GRADUATE BIOSTATISTICS SERIES

rgtlab Curriculum Project



Reproducible Research for the Health Sciences

A graduate textbook

First Edition · 2026

rgtlab

Welcome

This is the online version of **Reproducible Research for the Health Sciences** by the rgtlab curriculum team, a graduate course textbook.

In this book we take a position that a health scientist will recognize from their own methodological training. Computational reproducibility is not a property of any single file. It is a property of the relationships among the operating system, the language runtime, the package set, the session configuration, the source code, and the data. A result reproduces when those relationships are captured well enough that another person, on another machine, at a later time, obtains the same answer. The mature tools of the R ecosystem each capture one or two of these relationships. The unsolved problem, and the subject of this book, is composing them into a single workflow that a working analyst can actually adopt.

We organize the book around three ideas that recur in every chapter. The first is a ladder of reproducibility levels. It runs from a merely locatable analysis (L0) through pinned packages (L1) and a pinned environment (L2) to a verified analysis whose recorded outputs have been shown to regenerate (L3). The second is a distinction between capturing reproducibility and validating it: pinning an environment captures a level, whereas continuous integration and verification confirm it. The third is the research compendium, a single directory that bundles data, code, and manuscript inside a structure that any reader can navigate and any machine can rebuild.

The concrete vehicle throughout is `zzcollab`, an open-source R framework built around the levels ladder. It composes the standard tools (`renv` or Nix for packages, Docker or a runtime alternative for the environment, a research-compendium package structure, continuous integration, testing, and data-integrity checking) into a single system. We teach the concepts first and the tool second. A reader who finishes the book should be able to reason about reproducibility with any toolchain, and to operate this one.

What the reader is assumed to know

- R at a working level: writing functions and scripts, installing packages, and rendering an R Markdown or Quarto document.
- Comfort at the command line: running a shell command, editing a text file, and using `git` at a basic level.
- No prior experience with Docker, containers, continuous integration, or Nix is assumed. Each is built up from first principles.

The reader we have in mind is a graduate student or working analyst in public health, biostatistics, epidemiology, or a related quantitative health science. Such a reader produces analyses that others will need to trust, audit, and rebuild.

License

This book is licensed under Creative Commons Attribution 4.0 International. You may share and adapt it for any purpose, including teaching, provided you give attribution.

The code samples printed in this book are licensed under the MIT License. The repository code (package functions and build scripts) is licensed under GPL-3.

Reproducible Research for the Health Sciences

Compendia, containers, and computational verification in R.

Draft, November 13, 2026. Content may change.

Copyright

Reproducible Research for the Health Sciences by the rgtlab curriculum team is licensed under a Creative Commons Attribution 4.0 International License. You may share and adapt this work for any purpose, including excerpting chapters for a course, provided you give attribution.

The code samples printed in this book are licensed under the MIT License. The repository code (package functions and build scripts) is licensed under GPL-3.

To cite this book, please use:

> rgtlab curriculum team (2026). *Reproducible Research for the Health Sciences: Compendia, Containers, and Computational Verification in R*. Available at https://reproducibility.rgtlab.org.

Table of contents

What the reader is assumed to know . . . 2
License . . . 2

Copyright **5**

Preface **13**
The organizing idea . . . 13
The tool, and why it is here . . . 14
What this book covers . . . 14
What this book does not cover . . . 15
Reading it in one quarter . . . 15
Chapter template . . . 16

Conventions **17**
Code . . . 17
The chapter card . . . 17
Where a chapter sits on the stack . . . 18
The folded Check . . . 18
Callouts . . . 19
Habits . . . 20
The two datasets . . . 20
The one-quarter core path . . . 21
Cross-references . . . 22

1. Introduction: reproducibility as a computational problem **23**
1.1. A result that ran only once . . . 23
1.2. What reproducibility means . . . 24
1.3. The evidence that it is not happening . . . 25
1.4. Why the failures are computational, not statistical . . . 27
1.5. The tools, and the framework this book uses . . . 28
1.6. The plan of this book . . . 29

1.7. How to use this book 30

I. Foundations **33**

2. The Reproducibility Crisis **35**
2.1. Orientation 37
2.2. Where this sits on the stack 38
2.3. The analyst's contribution 39
2.4. The evidence that re-execution fails 41
2.5. Three words that are not synonyms 42
2.6. Why the failures are computational, not statistical 46
2.7. A reproducible analysis can still be wrong 48
2.8. Worked example 49
2.9. The cohort, continued 51
2.10. Collaborating with an LLM 53
2.10.1. What to delegate to a coding agent 55
2.10.2. Disclosing and recording model use 56
2.11. Exercises 57
2.12. Solutions 58
2.13. Cheat sheet 66
2.14. Further reading 67

3. Levels of Reproducibility **69**
3.1. Orientation 71
3.2. The analyst's contribution 72
3.3. The stack a result stands on 74
3.4. The ladder: capture sets the level, validation confirms it . . 78
3.5. Reproduced, replicated, and a warning about the words . . 83
3.6. Two numbering schemes, and which this book uses 85
3.7. The ladder grades every archetype 86
3.8. No framework pins the whole stack 87
3.9. Worked example 88
3.10. The cohort, continued 89
3.11. Collaborating with an LLM 91
3.12. Exercises 94
3.13. Solutions 95
3.14. Cheat sheet 104
3.15. Further reading 104

Draft, November 13, 2026. Content may change.

4. Version Control with Git and GitHub **107**
4.1. Where this sits on the stack . . . 109
4.2. Orientation . . . 110
4.3. The analyst's contribution . . . 112
4.4. What version control records, and what it cures . . . 114
4.5. The git model: repository, commit, history . . . 115
4.6. Reading the history: the log and the diff . . . 118
4.7. What must never be committed: `.gitignore` and PHI . . . 122
4.8. Branches, lightly . . . 129
4.9. Undoing things, and merge conflicts . . . 132
4.10. The remote: GitHub, GitLab, and collaboration . . . 136
4.11. What git does not do . . . 138
4.12. Worked example . . . 139
4.13. Collaborating with an LLM . . . 142
4.14. Exercises . . . 145
4.15. Solutions . . . 146
4.16. Cheat sheet . . . 157
4.17. Further reading . . . 158

5. The Research Compendium **161**
5.1. Where this sits on the stack . . . 163
5.2. Orientation . . . 164
5.3. The analyst's contribution . . . 166
5.4. The compendium as the unit of reproducible work . . . 167
5.5. The R package as the compendium container . . . 168
5.6. Building and maintaining the package skeleton . . . 170
5.7. The Five Pillars . . . 172
5.8. The directory layout, concretely . . . 174
5.9. The README and the data dictionary . . . 175
5.9.1. The README's computational requirements . . . 176
5.10. Project-oriented paths: `here`, not `setwd()` . . . 178
5.11. The five archetypes . . . 179
5.11.1. 1. Data analysis (a cohort study) . . . 180
5.11.2. 2. Scholarly manuscript (a comorbidity paper) . . . 180
5.11.3. 3. R package (a summary-table package) . . . 181
5.11.4. 4. Simulation study (a power analysis) . . . 181
5.11.5. 5. Blog (a teaching notebook) . . . 182
5.12. How each archetype treats the chapters ahead . . . 188

5.13. Progressive disclosure . . . 191
5.14. Worked example . . . 192
5.15. The cohort, continued . . . 194
5.16. Collaborating with an LLM . . . 196
5.17. Exercises . . . 198
5.18. Solutions . . . 199
5.19. Cheat sheet . . . 208
5.20. Further reading . . . 209

II. Capturing the Environment 211

6. Pinning Packages with renv 213
6.1. Where this sits on the stack . . . 214
6.2. Orientation . . . 215
6.3. The analyst's contribution . . . 217
6.4. From packrat to renv . . . 218
6.4.1. What the lockfile records . . . 219
6.4.2. The init, snapshot, restore workflow . . . 222
6.5. The dependency-triad invariant . . . 223
6.5.1. Scanning the documentation, not only the code . . . 229
6.5.2. The triad across archetypes . . . 230
6.6. Dated snapshots of a package repository . . . 231
6.7. The limits of renv . . . 232
6.8. Lighter and alternative approaches . . . 233
6.9. Worked example . . . 234
6.10. The cohort's own dependencies . . . 236
6.11. Collaborating with an LLM . . . 238
6.12. Exercises . . . 240
6.13. Solutions . . . 242
6.14. Cheat sheet . . . 250
6.15. Further reading . . . 251

7. Pinning the Environment with Docker 253
7.1. Where this sits on the stack . . . 255
7.2. Orientation . . . 256
7.3. The analyst's contribution . . . 257
7.4. What the lockfile leaves standing . . . 259
7.5. Images and containers . . . 260

Draft, November 13, 2026. Content may change.

7.6. The Rocker project . . . 262
7.7. Making the image itself reproducible . . . 263
7.8. Apple Silicon and multi-architecture images . . . 271
7.9. Environment-variable determinism . . . 273
7.10. Profiles as base images . . . 276
7.11. A first look at the two-layer team model . . . 278
7.12. Worked example . . . 278
7.13. The cohort, continued . . . 282
7.14. Collaborating with an LLM . . . 283
7.15. Exercises . . . 285
7.16. Solutions . . . 287
7.17. Cheat sheet . . . 298
7.18. Further reading . . . 299

8. Backends and Runtimes: Nix, Podman, and Apptainer 301
8.1. Where this sits on the stack . . . 303
8.2. Orientation . . . 304
8.3. The analyst's contribution . . . 305
8.4. Axis one: a different runtime over the same image . . . 307
8.4.1. Podman . . . 307
8.4.2. Apptainer . . . 308
8.5. Axis two: a different environment model without a container 310
8.5.1. Nix and the rix front-end . . . 310
8.5.2. Guix and the conda family . . . 312
8.6. Axis three: remote execution . . . 314
8.7. Choosing among the alternatives . . . 315
8.8. The cohort, continued . . . 318
8.9. Worked example . . . 319
8.10. An HPC worked example: Apptainer under Slurm . . . 321
8.11. Collaborating with an LLM . . . 323
8.12. Exercises . . . 325
8.13. Solutions . . . 327
8.14. Cheat sheet . . . 334
8.15. Further reading . . . 335

III. Validation **337**

9. Literate Programming and the Rendered Report **339**
9.1. Where this sits on the stack . . . 341
9.2. Orientation . . . 342
9.3. The analyst's contribution . . . 343
9.4. The oldest idea in the book . . . 345
9.5. From Sweave to Quarto . . . 346
9.6. The sealed render . . . 347
9.6.1. The seal applies to numbers in sentences, not only to figures . . . 350
9.7. The landmines that break the seal . . . 351
9.8. Freeze and cache: renders that skip the code . . . 354
9.9. HTML or PDF, and the toolchain each needs . . . 355
9.10. Why a rendered document will not hash the same twice . . . 356
9.11. Literate programming across the archetypes . . . 358
9.12. The write-as-you-go doctrine . . . 359
9.13. Worked example . . . 360
9.14. The cohort, continued . . . 362
9.15. Collaborating with an LLM . . . 363
9.16. Exercises . . . 366
9.17. Solutions . . . 367
9.18. Cheat sheet . . . 376
9.19. Further reading . . . 378

10. Automating the Analysis: Make and targets **379**
10.1. Where this sits on the stack . . . 381
10.2. Orientation . . . 383
10.3. The analyst's contribution . . . 384
10.4. The analysis as a dependency graph . . . 386
10.4.1. The graph of an actual analysis . . . 389
10.5. GNU Make: targets, prerequisites, and rules . . . 391
10.6. targets: a function-oriented pipeline for R . . . 394
10.7. What a pipeline does, and does not, add . . . 398
10.8. Pipelines across the archetypes . . . 400
10.9. Beyond R: Snakemake, Nextflow, and workflowr . . . 400
10.10. Worked example . . . 402
10.11. The cohort, continued . . . 404

Draft, November 13, 2026. Content may change.

10.12 A language model inside the pipeline . . . 406
10.13 Collaborating with an LLM . . . 408
10.14 Exercises . . . 411
10.15 Solutions . . . 413
10.16 Cheat sheet . . . 423
10.17 Further reading . . . 424

11. Testing and Code Quality 427
11.1. Where this sits on the stack . . . 428
11.2. Orientation . . . 430
11.3. The analyst's contribution . . . 431
11.4. The reproducibility paradox, stated precisely . . . 433
11.5. The analysis plan as an oracle . . . 434
11.6. tinytest: bare expectations . . . 435
11.7. The four kinds of test . . . 436
11.7.1. Unit tests of functions . . . 437
11.7.2. Integration tests of the pipeline . . . 437
11.7.3. Data-validation tests . . . 438
11.7.4. Reproducibility tests . . . 440
11.8. Monte Carlo error and parallel streams . . . 447
11.9. Test behavior, not implementation . . . 453
11.10 Code quality as a validation feature . . . 454
11.11 Validation in the regulated sense . . . 456
11.12 Reviewing analysis code . . . 458
11.13 Testing across the archetypes . . . 459
11.14 Worked example . . . 460
11.15 Collaborating with an LLM . . . 463
11.16 The cohort, continued . . . 465
11.17 Exercises . . . 469
11.18 Solutions . . . 470
11.19 Cheat sheet . . . 481
11.20 Further reading . . . 483

12. Continuous Integration and Verification 485
12.1. Where this sits on the stack . . . 487
12.2. Orientation . . . 488
12.3. The analyst's contribution . . . 490
12.4. The machine that is not yours . . . 492

Draft, November 13, 2026. Content may change.

12.5. What the checks are: the three gates 493
12.5.1. Where CI bugs hide: cross-boundary state 498
12.6. Secrets, tokens, and pinned actions 500
12.7. Two dialects: GitHub Actions and GitLab CI 502
12.8. Verification and level L3 505
12.9. Data integrity: the checksum manifest 509
12.10 Proving the framework itself: the capture test matrix 512
12.11 How continuous integration adapts to the archetype 514
12.12 Independent external verification 514
12.13 Worked example 516
12.14 Debugging a failed reproduction 519
12.15 Demonstrating a gate on the cohort compendium 521
12.16 Collaborating with an LLM 523
12.17 Exercises 526
12.18 Solutions 528
12.19 Cheat sheet 542
12.20 Further reading 543

IV. Collaboration and Stewardship 545

13. Collaboration, Archiving, and Stewardship 547
13.1. Where this sits on the stack 549
13.2. Orientation 550
13.3. The analyst's contribution 552
13.4. Sharing with a team: the two-layer model 554
13.4.1. The Dockerfile, not the image, is the guarantee . . . 556
13.4.2. Registries and the forge 557
13.5. Sharing with the world: archiving 558
13.5.1. Licensing, CITATION.cff, and Software Heritage . . 559
13.5.2. When the data are restricted 561
13.5.3. Working inside a trusted research environment . . . 564
13.5.4. Funder and journal obligations 566
13.6. Stewardship over time 570
13.7. Across the archetypes 571
13.8. Worked example 572
13.9. The cohort, continued 576
13.10 Collaborating with an LLM 579

Draft, November 13, 2026. Content may change.

13.11Exercises 581
13.12Solutions 583
13.13Cheat sheet 594
13.14Further reading 596
13.15In conclusion 596

References **599**

Appendices **609**

Shell Primer **609**
Where you are: `pwd`, `ls`, `cd` 609
File operations: `cp`, `mv`, `rm` 610
Redirection and pipes: `>`, `>>`, `|` 611
Environment variables and `PATH` 613
Permissions and the execute bit 614
A note on bash versus zsh 615

Setup Appendix: Docker and Git on Your Own Machine **617**
Apple Silicon (M-series Mac) 617
Windows with WSL2 618
Locked-down institutional laptops 619
Diagnosing a failed or unexpectedly slow build 620
Git preflight 622
When git refuses 624

Cheat-Sheet Index **627**
Foundations 627
Capturing the Environment 628
Validation 628
Collaboration and Stewardship 629

Glossary **631**

Credits **651**

Colophon **653**

Draft, November 13, 2026. Content may change.

List of Figures

1.1. Reported success rates in four studies, separated by what each measured. The left panel reports replication studies, in which new data was collected to test a prior finding. The right panel reports re-execution studies, in which deposited code was run again without any attempt to evaluate the science. The re-execution figures are those stated in the text of the studies as cited in this chapter, and are approximate where the source reports a bound rather than a point estimate. 27

2.1. The determinant stack of Chapter 3, with no layer captured or validated. This chapter diagnoses, in prose, why re-execution fails on two of these layers, the R runtime and the session configuration, but pins nothing and checks nothing pinned; the chapters that follow are the ones that mark this figure. 39

2.2. Reproducibility, replicability, and robustness distinguished by what is held fixed. The columns vary the data and the rows vary the analysis, so each cell names the claim supported when a consistent result survives that particular change. Only the shaded cell, in which both the data and the analysis are held fixed, is the subject of this book. The grid and the robustness cell follow the Turing Way (The Turing Way Community, 2022); the definitions of reproducibility and replicability are those of the National Academies (National Academies of Sciences, Engineering, and Medicine, 2019) adopted above. 45

Draft, November 13, 2026. Content may change.

3.1. The determinant stack, from the hardware at the base to the data at the top, colored by the rung of the ladder that first secures each layer. Package versions are secured at L1, the operating system and R runtime at L2, and the analysis code and data are merely locatable at L0. The session-configuration layer is only partially pinned at L2: the locale and the time zone are fixed, but the RNG kind, the contrasts, and the number of printed digits are not. Three further layers, the hardware, the numerical libraries, and intrinsic non-determinism, are pinned by no rung of the capture ladder. Every chapter that draws this stack draws it with the same function, as the Conventions page explains, so that no two chapters can disagree about what a layer is called or which rung secures it. 76

4.1. The determinant stack of Chapter 3, with the analysis-code layer marked as captured here. Version control is what first pins the code, lifting it from unlocatable to L0; the input-data layer immediately below is left unhighlighted on purpose, because this chapter spends as much effort keeping some data out of the history as it spends getting the code into it. 110

4.2. A branch and its merge, drawn as the commit graph git actually stores. The lower line is `main`; the upper line is a sensitivity-analysis branch created at commit B and merged back at M. Commits A through C exist once and are shared by both lines: the branch adds commits rather than copying them. The merge commit M is the only commit in the diagram with two parents, which is what records that the two lines rejoined and when. 132

5.1. The determinant stack of Chapter 3, with analysis code and input data marked as captured by this chapter. A research compendium does not pin a package version or an operating system; what it captures, for the first time, is that the code and the data are locatable, organized by convention, and separated into raw and derived, which is the L0 rung this chapter secures and the container the later chapters install L1 through L3 into. 164

Draft, November 13, 2026. Content may change.

5.2. Validation gates by archetype. The R-package check and the unit tests apply universally; the render and seeding gates apply selectively. Where a render gate is only optional, as for a data analysis or a simulation study without a report, the cell is left unmarked. The underlying reproducibility machinery, the Five Pillars, is identical across every row. . . 187

6.1. The determinant stack of Chapter 3, with package versions marked as captured here. Pinning this one layer with a committed `renv.lock` is the L0-to-L1 transition this chapter teaches; the R runtime, the operating system, and the system libraries beneath it remain unpinned until Chapter 7 adds the container. 215

6.2. The dependency triad as nested sets, with example packages placed in each region. Everything the code calls must be pinned, so a package outside the lockfile is the one broken position and is marked with a cross. The band between `DESCRIPTION` and the lockfile holds the transitive dependencies no analyst names directly, together with, in a compendium, the packages used only by analysis scripts. That second case is the archetype qualification discussed above: it is a violation in an R package and legitimate in a compendium. 227

7.1. The determinant stack of Chapter 3, with the operating system and system libraries, and the R runtime, marked as captured by this chapter. These are the two layers a lockfile cannot reach, because neither is an R package; pinning them is the L1-to-L2 step this chapter teaches. 256

7.2. One unchanged recipe built repeatedly over a year, under a tag and under a digest. The upper row builds on whatever the tag currently points at, so a maintainer rebuild in April and another in September silently change the ground the analysis stands on, giving three different environments from identical text. The lower row names a digest, builds for one fixed platform, and receives the same bytes at every build. The maintainer rebuilds are ordinary maintenance, not mistakes: the point is that a mutable pointer cannot carry an immutable claim. 266

Draft, November 13, 2026. Content may change.

8.1. The determinant stack of Chapter 3, with the operating system, system libraries, and R runtime marked as captured, the same two layers Chapter 7 captures with Docker. This chapter reaches the identical pinned-environment rung, L2, by Podman, Apptainer, Nix, conda, or a remote service instead of a Docker build: the same rung, by a different route, not a new capability. 304

9.1. The determinant stack of Chapter 3, with analysis code marked as validated rather than captured. A sealed render pins no new layer of the stack; it confirms that the numbers and figures a reader sees are the ones the already-committed code actually produces, which is the capture-versus-validation distinction this chapter applies to the rendered document itself. 341

9.2. The reversal of an association upon pooling. Within each synthetic species bill depth rises with bill length (solid lines); across the pooled sample the fitted line falls (dashed). Because the figure is computed when the book is built, the described reversal cannot drift from the code that produces it. 349

10.1. The determinant stack of Chapter 3, with the analysis-code and input-data layers marked as validated, not captured, by a pipeline tool. Make and `targets` confirm that a recorded output is current with respect to the code and data that produced it; they do not pin a package version, an operating system, or an R runtime, and they say nothing about whether the code or the data are correct, only whether the output in hand still reflects them. 382

10.2. The analysis of the orientation as a directed acyclic graph. Each node is a step that produces an output; each arrow records that the step it points to consumes what the previous step produced. A change to the raw data invalidates every node downstream of it (all four to its right); a change to the plotting code invalidates only the figures and the report. A pipeline tool reads this graph to rebuild exactly the stale nodes and skip the rest. 388

Draft, November 13, 2026. Content may change.

11.1. The determinant stack of Chapter 3, with analysis code and input data marked as validated rather than captured. This chapter pins no new layer; a unit test, a data-validation test, and a baseline test each confirm that the code and the data already captured by earlier chapters still produce the answer they are supposed to, which is the capture-versus-validation distinction the chapter develops. 429

11.2. Four representative defects against the four kinds of test, with a fifth row for re-execution alone. A filled cell marks the kind of test most likely to catch that defect. The matrix is close to diagonal, which is the point: the kinds are complements rather than alternatives, and a suite missing one kind is blind to a whole class of defect. The bottom row is empty because every defect shown is deterministic, so re-running the analysis reproduces it faithfully. 452

12.1. The determinant stack of Chapter 3, with package versions, analysis code, and input data marked as validated rather than captured. CI pins no new layer: the dependency-manifest gate confirms packages are still pinned, `R CMD check` and the render gate confirm the analysis code still runs, and the checksum manifest confirms the input data is unchanged, which applies the capture-versus-validation distinction to the three gates of this chapter. 488

13.1. The determinant stack of Chapter 3, with input data marked as captured here and the package-version and operating-system layers marked as validated. This chapter pins no new byte: the packages are captured in Chapter 6 and the operating system in Chapter 7. What it adds to the data layer is durability, lifting it from merely locatable on a forge to findable under a persistent identifier, which is why input data is captured rather than validated here; what it adds to the package and system layers is the stewardship check, using an archived image or a date-based reconstruction, that those earlier pins still resolve, which is why they are validated rather than captured again. 550

Draft, November 13, 2026. Content may change.

List of Tables

2.1. The same seed yields the same draws only under the same RNG kind. Each row shows the first six values of a shuffled sequence of the integers one through ten, drawn under seed 123. 50

3.1. The reproducibility ladder used throughout this book. Each rung names what it pins, the primary artifact that marks it, and what reaching it means for a reader of health research. 80
3.1. The reproducibility ladder used throughout this book. Each rung names what it pins, the primary artifact that marks it, and what reaching it means for a reader of health research. 81

4.1. A synthetic commit history for the first weeks of a cohort analysis, read as a record of decisions. Each row is one atomic, message-bearing commit; together they record not only what changed but, in the message, why. Note the reviewer-driven change to the outlier threshold, the kind of decision an auditor is entitled to trace. 121

5.1. The five research-compendium archetypes. The reproducibility machinery is identical across all five; only the directory emphasis, the presence of a rendered report, and the gates that apply on top of the universal R CMD check differ. . . . 184
5.2. How each archetype treats the topics of the chapters ahead. The reproducibility machinery under every cell is the same; the table records only where the emphasis falls. 189

6.1. A synthetic lockfile diff between two snapshots of one project. The changed column flags every package whose pinned version moved between the earlier and later snapshot. 235

Draft, November 13, 2026. Content may change.

6.2. Packages `renv::dependencies()` finds referenced in the two files that produce the running cohort, scanned directly rather than asserted. `simulate_cohort.R` contributes no rows because it calls base R functions (`rnorm`, `sample`, `as.Date`) without `library()` or a `::` prefix, so the scan finds no package name to report. 237

7.1. The same numbers written under a period decimal mark and a comma decimal mark, as a spreadsheet export might write them, and what a naive period-assuming cleaning step reads back from each. The comma-convention values are silently misread, with no error raised. 274

8.1. Backends and runtimes for reaching the pinned environment, by axis 317

10.1. The retinopathy analysis expressed as a dependency graph, with the cost of each step measured rather than asserted. The timings are from one run on one machine and will differ on another, and repeated runs on the same machine have varied by several-fold; even the ratios between steps should be read as illustrative, not as a stable property of the analysis. Note that the expensive step sits at the end, so an edit to the plotting code that forced a full rebuild would waste almost all of the total. 390

10.2. The two steps `load_cohort()` and `clean_cohort()` perform on the running cohort compendium, with elapsed time measured on one run on the machine that built this chapter; repeated runs vary and should not be read as a stable property of either function. 405

11.1. Eyes and events by treatment arm in the diabetic retinopathy trial (`survival::diabetic`), the real dataset this book computes on. Each count is verifiable by hand from the 394 records, which is what makes the table usable as a test oracle rather than merely as a description. 439

Draft, November 13, 2026. Content may change.

11.2. A baseline test across sessions. The bound recorded under the pre-3.6.0 sampler is reproduced exactly by a rerun with the same seed and sampler, and is not reproduced seed-exactly by a rerun with the same seed under the current default sampler, so the comparison at tolerance 1e-8 passes once and fails once. A same-session check would have passed in both cases. 444

11.2. A baseline test across sessions. The bound recorded under the pre-3.6.0 sampler is reproduced exactly by a rerun with the same seed and sampler, and is not reproduced seed-exactly by a rerun with the same seed under the current default sampler, so the comparison at tolerance 1e-8 passes once and fails once. A same-session check would have passed in both cases. 445

11.3. A correct function and a buggy variant checked against three inputs with hand-computed means. The buggy variant returns the same wrong value on every input and would return it on every machine and every rerun: it is perfectly reproducible and still wrong. 462

12.1. A single altered value changes the digest of a data file. The three CSV files are hashed by their bytes on disk. The second holds the same values as the first, written from a plain data frame rather than a tibble, and its digest matches. The third differs in exactly one cell, and its SHA-256 digest differs from the first character, so a checksum manifest detects the change. 511

13.1. Code-availability policies of three journals, as stated on each journal's author pages on 8 November 2026. 569

13.2. The renv union model over one quarter of team collaboration. Each contributor installs packages independently; the project lockfile records the cumulative union, which the shared image is periodically rebuilt from. Because it is a union rather than a running total, a package a contributor installs that another already has does not enlarge it, so the union can grow by fewer than the number of packages added. The union size never shrinks, so every contributor who rebuilds inherits the full set. 573

Draft, November 13, 2026. Content may change.

1. Git refusals a new user meets, indexed by the message git or the forge prints. Messages from git 2.56.0, run 2026-11-08; the two forge rows were not reproduced. 624
1. The rendering environment of the last execution of this page, read from the live session, beside the versions the committed lockfile records. 653
1. The rendering environment of the last execution of this page, read from the live session, beside the versions the committed lockfile records. 654

Draft, November 13, 2026. Content may change.

Preface

This book grew out of a recurring and dispiriting experience: an analysis that ran perfectly on the machine where it was written, and nowhere else. The script that produced last year's figures now errors on a colleague's laptop. The supplement that accompanied a published paper will not knit, because a package it silently depended on has moved on. The reviewer who asks, reasonably, for the code receives a folder that does everything except reproduce the result. None of these failures is a failure of statistics. Each is a failure of computational reproducibility, and each was avoidable.

The health sciences have absorbed the lesson that a result must be reproducible in principle, and journals now ask for the code. What the field has not yet absorbed is that making a computational result reproducible in practice is a technical discipline in its own right. It has its own tools, its own failure modes, and its own standards of evidence. In this book we teach that discipline to an audience that already has the quantitative training to appreciate why it matters.

The organizing idea

The organizing idea of the book is that reproducibility is layered, and that honesty about which layer one has reached is itself a scientific virtue. We develop a ladder of four levels. A locatable analysis (L0) is under version control and can be found, but nothing about its computational environment is fixed. A pinned analysis (L1) records the exact versions of the packages it used. A pinned-environment analysis (L2) fixes, in addition, the operating system, the language runtime, and the system libraries, most often inside a container. A verified analysis (L3) has gone one step further: its recorded computational outputs have been regenerated from scratch and shown to match. In the chapters that follow we teach how to reach each level and

how to tell which one an analysis has actually reached. Just as important, we consider when a lower level is the honest and sufficient choice.

Running underneath the ladder is a distinction we return to constantly: the difference between capturing reproducibility and validating it. Pinning packages and pinning the environment are acts of capture; they fix the inputs and thereby set the level. Continuous integration, dependency checking, and verification are acts of validation; they confirm the level without capturing anything new. Conflating the two is the source of a great deal of false confidence.

The tool, and why it is here

The book uses `zzcollab`, an R framework for building reproducible research compendia, as its concrete apparatus. The choice is deliberate and, we hope, honestly motivated. Reproducibility is learned by building something that reproduces, and that requires a real toolchain. `zzcollab` composes the established R tools into one workflow organized by the levels ladder. This allows us to teach the concepts and demonstrate them in the same breath. A reader who prefers a different toolchain will find that the concepts transfer without loss. That is, the ladder, the compendium, and the capture-validation distinction are properties of the problem, not of any one program.

What this book covers

Following the introduction, the remaining twelve chapters are arranged in four parts, which we outline here.

1. **Foundations.** The reproducibility crisis and its specifically computational character; a framework of reproducibility levels and the determinants that set them; version control with git as the foundation of a locatable analysis; and the research compendium as the unit of reproducible work.

Draft, November 13, 2026. Content may change.

2. **Capturing the environment.** Pinning packages with a lockfile; pinning the operating system and runtime with a container; and the alternatives, from Nix to Podman to the high-performance-computing runtime Apptainer.
3. **Validation.** Literate programming and the rendered report; automating the analysis with make and targets; testing and code quality; and continuous integration and computational verification, the acts that turn a claimed level into a confirmed one.
4. **Collaboration and stewardship.** Sharing a compendium with a team and with the world: the team environment model, package registries, archival deposition, and the FAIR principles.

What this book does not cover

The book is not a statistics text. We assume the reader can already conduct an analysis, and we concern ourselves only with making that analysis reproducible. It is not a comprehensive manual for any single tool; where a tool's own documentation is the right reference, we point to it. Nor is it a treatise on replicability in the scientific sense, the question of an independent study reaching the same conclusion on new data. That is a vital matter, but a different one. We are careful throughout to keep the computational question of re-execution distinct from it.

Nor is the book a manual of computer-system validation for regulated work. Where we touch 21 CFR Part 11, qualification, or validation in the regulatory sense, we do so to keep the vocabulary straight and to mark where a regulated setting asks for more. A reader preparing analyses for submission should treat those passages as orientation only. The governing documents are the regulations themselves, the sponsor's standard operating procedures, and guidance such as the R Validation Hub's (Nicholls et al., 2020).

Reading it in one quarter

We have written the book for a one-quarter, ten-week course, but not every chapter fits that schedule. The Conventions page marks a core path through

the thirteen chapters. It also names the material a ten-week offering has no room for, namely Backends and Runtimes in full and the latter half of Automating the Analysis. An instructor teaching to that pace will find a week-by-week table built from the same core path in the instructor guide, which is distributed separately to course instructors and is not part of this book. A reader with more time, or working alone, can simply read the book in order. Nothing marked optional is weaker material, only material a shorter term cannot fit.

Chapter template

Each content chapter follows the same sequence. It opens with learning objectives and an orientation that motivates the chapter from a concrete research problem. It then identifies the analyst's contribution that no tool automates, and proceeds through content sections with worked R and shell examples. It closes with a section on collaborating with a language model, exercises, and further reading. The pattern repeats deliberately, so that by the third chapter the reader knows where to find each component.

Draft, November 13, 2026. Content may change.

Conventions

On this page we summarize the visual and structural conventions used throughout the book.

Code

Two kinds of code appear. R code is shown in syntax-highlighted blocks; its output is prefixed with `#>` to make the boundary between input and output explicit:

```
mean(c(1, 2, 3, 4, 5))
#> [1] 3
```

Shell commands, which recur constantly in this book, are shown with a `$` prompt that is not part of the command:

```
$ zzc status
```

Inline code is in `monospace`. File and directory names (`renv.lock`, `analysis/`) and command names (`make`, `zzc`) are also set in monospace.

The chapter card

Each numbered chapter opens with a card. The card gives an estimated reading time and names the earlier chapters and the statistical knowledge the chapter assumes. The time is computed, not guessed: the prose words before the exercises, plus ten words for each line of code, read at 200 words

a minute and rounded to five minutes. It excludes the exercises, whose time depends on the reader. A test fails when a card drifts more than five minutes from the rule. It also lists three to five objectives, phrased as questions the reader should be able to answer by the end, and a short outline of what follows. A reader who already knows the material can check the objectives and skip ahead.

Where a chapter sits on the stack

A computational result rests on nine layers, from the hardware at the base to intrinsic non-determinism at the top (Chapter 3 works through all nine). Most chapters open with a figure placing that chapter on the stack. In that figure the layers the chapter **captures** (pins for the first time, or pins more tightly than an earlier chapter did) are drawn in full color. The layers it **validates** (checks without itself pinning anything new) are outlined. Every such figure is a call to the same function, `plot_determinant_stack()`, and its alt text is computed from the same call that draws the figure, so the two cannot describe different layers.

The folded Check

Most numbered sections end with a folded Check: a short question testing comprehension of the section just read, with the answer folded inside it so that attempting the question is not optional. Expand the box to read the question, attempt an answer, and only then expand the Answer nested inside it. On paper a box cannot be folded, so the printed edition numbers the checks in each chapter and collects their answers at the end of the chapter, under ‘Answers to the checks’, with page references both ways. The example below keeps its answer in place.

Check your understanding: example

A short question testing comprehension of the just-read material.

Answer

Draft, November 13, 2026. Content may change.

The worked answer, with the reasoning that leads to it rather than the conclusion alone.

Callouts

Three kinds of callout appear, each with one job.

Verification

How to see that a claim actually holds, and, just as important, how to make it fail on purpose. A gate that cannot fail is not a gate.

⚠ Pitfall

A specific way the preceding technique goes wrong in practice, not a generic warning.

❗ Restricted data

Where protected, identifiable, or data-use-agreement restricted data may and may not go: which endpoints, which tools, which repositories. This callout appears wherever a technique could tempt a reader into committing, depositing, or pasting such data.

A fourth, smaller device, the honest level, closes most chapters: one sentence naming the rung the chapter's own practice reaches and what it still leaves exposed.

❗ The honest level

One sentence, specific to this chapter, not the whole ladder.

Draft, November 13, 2026. Content may change.

Habits

Seven numbered habits recur across the book, introduced in full in Chapter 1 and recalled in the closing chapter. A rule of thumb in a chapter cites its habit by number, for example 'habit 4', rather than restating it.

The two datasets

We use two datasets in the book, and each has a different job.

`survival::diabetic` is real. It holds 394 records from a randomized trial of laser photocoagulation for diabetic retinopathy, with one row per eye and two eyes per patient. It ships with every R installation, because `survival` is a recommended package, so no chapter that uses it adds a dependency the reader must install or the lockfile must pin. It carries the computation: where the book fits a model, draws a figure, or reports a number, that number comes from real data and the reader can regenerate it.

The incident-diabetes cohort is synthetic, and is written by `analysis/scripts/simulate_cohort.R` from a fixed seed. It carries two threads the real data cannot. The first is confidentiality: the extract holds record numbers, names, and dates of birth, so the chapter on version control can show a file that must never enter a repository's history. No public dataset contains protected information, by construction, so that lesson requires an invented one. The second is data validation. The extract carries deliberate defects (two date formats in one column, a duplicated record, an implausible age, an inconsistently coded sex, and missing outcomes), so that the tests in Chapter 11 have something to catch. Clean data would have to be broken on purpose in any case.

The synthetic choice is not a compromise, and we would make it again with real data available. A committed generator with a fixed seed is *more* reproducible than a downloaded extract. It has no external dependency, no link rot, and no access control, and a reader can regenerate it byte for byte in a second. What it gives up is friction. Invented messiness is tidier and stranger than the messiness of an actual registry export, and a reader who has only ever met generated data will be surprised by their first real one.

Draft, November 13, 2026. Content may change.

The cohort compendium is still taking shape across the book's chapters, and not every chapter uses it yet. A reader who notices a chapter working from a different example is therefore seeing that work in progress, not a different design.

The one-quarter core path

The book is written for a one-quarter, ten-week course. The table below marks a core path through it; a chapter marked optional is not weaker material, only material a ten-week schedule has no room for. An instructor with more time, or a reader working alone, can take the whole book in order.

Chapter	Status
Introduction	Core
The Reproducibility Crisis	Core
Levels of Reproducibility	Core
Version Control with Git and GitHub	Core
The Research Compendium	Core
Pinning Packages with renv	Core
Pinning the Environment with Docker	Core
Backends and Runtimes: Nix, Podman, and Apptainer	Optional
Literate Programming and the Rendered Report	Core
Automating the Analysis: Make and targets	Core for the Make section; `targets`, Snakemake, Nextflow, and workflowr are optional
Testing and Code Quality	Core
Continuous Integration and Verification	Core
Collaboration, Archiving, and Stewardship	Core

Draft, November 13, 2026. Content may change.

Cross-references

Sections, figures, and tables are referenced by their Quarto label (`@sec-levels`, `@fig-ladder`, `@tbl-tools`). Literature is cited by `@bibkey` and collected in the References.

Draft, November 13, 2026. Content may change.

1. Introduction: reproducibility as a computational problem

An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures.

Buckheit and Donoho, paraphrasing Jon Claerbout, *WaveLab and Reproducible Research* (1995)

1.1. A result that ran only once

Every quantitative health scientist has met the failure this book exists to prevent. An analysis runs cleanly on the machine where it was written and produces a table, a figure, a p-value. A year later, on a new laptop, or a colleague's, or a reviewer's, the same code errors. Worse, it may run to completion and return a different number. The statistics were sound. What failed was everything around the statistics: a package that changed beneath the code, a default that shifted between releases of R, a system library present on one machine and absent on another. The result had been reproducible exactly once, on one machine, at one moment, and no one noticed until the moment had passed.

This book is about closing that gap. Its subject is computational reproducibility, by which we mean the property that a computational result can be regenerated, within a stated tolerance, by another person on another machine at a later time. That is a narrower claim than scientific replication, and we shall be careful to keep the two apart throughout. It is nonetheless a claim that a great deal of published health-sciences research cannot currently support. Unlike replication, it is also a claim that can be secured by technical means available today.

Draft, November 13, 2026. Content may change.

We begin by fixing the vocabulary of the field and then review the evidence that computational reproducibility is often not achieved. We then consider why the failures are computational in character, and introduce the tools and the framework that the book uses. Lastly, we set out the plan of the book and how it is meant to be used.

1.2. What reproducibility means

The vocabulary of this field is contested, and the reader should fix the terms before wading into the literature. We follow the distinction adopted by the National Academies in their 2019 consensus report (National Academies of Sciences, Engineering, and Medicine, 2019). *Reproducibility* is obtaining consistent computational results using the same input data, computational steps, methods, code, and conditions of analysis. That is, it is a statement about re-execution. *Replicability* is obtaining consistent results across studies aimed at answering the same question, each with its own data. It is a statement about whether a scientific finding holds on new data. The two are independent. A result can be perfectly reproducible and entirely wrong, if the code faithfully regenerates a mistaken number every time. Conversely, a finding can replicate across laboratories while no single analysis in the set is computationally reproducible. Leek and Peng press the point that a reproducible analysis can still be wrong, and that reproducibility is therefore a floor rather than a ceiling (Leek & Peng, 2015). This book concerns itself almost entirely with the first of the two terms, and we say so plainly wherever the boundary is in doubt.

Reproducibility so defined is a spectrum. Peng (2011) arranged computational work along a scale from the merely published, through code available, code and data available, and linked and executable, to fully replicable. He argued that ‘reproducibility has the potential to serve as a minimum standard for judging scientific claims when full independent replication of a study is not possible’. We take from Peng the idea that reproducibility comes in degrees, and that honesty about which degree one has reached is itself a scientific virtue. We shall develop that idea into the levels ladder of Chapter 3.

Draft, November 13, 2026. Content may change.

1.3. The evidence that it is not happening

The case for taking this seriously is empirical, and it is grim. In psychology, the Open Science Collaboration's teams judged only 39 of 100 replications successful (Open Science Collaboration, 2015). In preclinical oncology, Begley & Ellis (2012) reproduced only 6 of 53 landmark findings. These are replication failures, at the scientific end of the spectrum, and their causes are many. However, a growing body of evidence isolates the specifically computational failures. These are, if anything, more damning, because they are more avoidable.

Trisovic et al. (2022) re-ran more than nine thousand R files deposited in the Harvard Dataverse. They found that roughly one in four executed without error, and that was before asking whether the output matched. Saju et al. (2025), examining R code supplements on the Open Science Framework, found that only about a quarter of the R scripts ran without error, even after an automated pipeline rebuilt their environments. They also found that nearly every project carried no description of its dependencies at all. These analyses could not be made to run a second time, so the question of whether they reached the right conclusion never arose. The scholarship, in Buckheit and Donoho's paraphrase of Claerbout, was advertised but never delivered.

Setting the four studies beside one another, as in Figure 1.1, makes the shape of the problem visible. The replication studies and the re-execution studies measure different things, and for that reason the figure keeps them in separate panels. Nevertheless, the rates are comparably low in both. The right-hand panel differs in one respect: it counts failures that better engineering would fix and that no amount of additional data collection would.

```
library(ggplot2)
library(tibble)

studies <- tibble(
  study = c('Open Science\nCollaboration (2015)',
            'Begley and\nEllis (2012)',
            'Trisovic et al.\n(2022)',
            'Saju et al.\n(2025)'),
```


Draft, November 13, 2026. Content may change.

```
  rate = c(39, 11, 26, 26),
  kind = factor(
    c('Replication: new data collected',
      'Replication: new data collected',
      'Re-execution: deposited code re-run',
      'Re-execution: deposited code re-run'),
    levels = c('Replication: new data collected',
               'Re-execution: deposited code re-run')
  )
)

ggplot(studies, aes(x = reorder(study, rate), y = rate)) +
  geom_col(aes(fill = kind), width = 0.62, show.legend =
   ↪  FALSE) +
  geom_text(aes(label = paste0(rate, '%')),
            hjust = -0.18, size = 3.4) +
  coord_flip(ylim = c(0, 48)) +
  facet_wrap(~kind, scales = 'free_y', ncol = 2) +
  scale_fill_manual(values = c(
    'Replication: new data collected' = '#4c72b0',
    'Re-execution: deposited code re-run' = '#14746f'
  )) +
  labs(x = NULL, y = 'Reported success rate (percent)') +
  theme_minimal(base_size = 10) +
  theme(
    panel.grid.major.y = element_blank(),
    strip.text = element_text(face = 'bold', size = 8.5)
  )
```


Draft, November 13, 2026. Content may change.

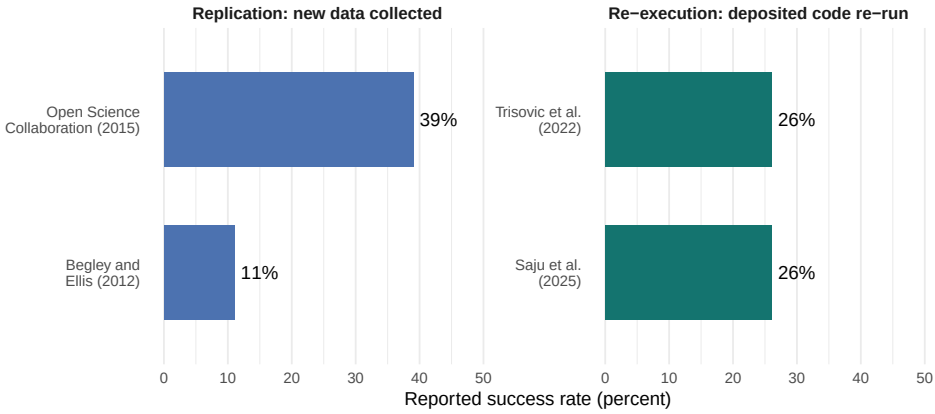


Figure 1.1.: Reported success rates in four studies, separated by what each measured. The left panel reports replication studies, in which new data was collected to test a prior finding. The right panel reports re-execution studies, in which deposited code was run again without any attempt to evaluate the science. The re-execution figures are those stated in the text of the studies as cited in this chapter, and are approximate where the source reports a bound rather than a point estimate.

The figure tells us how often replication and re-execution fail. It does not tell us why, and that question is taken up next.

1.4. Why the failures are computational, not statistical

The reasons re-execution fails point directly at the remedy. Consider two changes to R itself, each silent and each capable of altering a published result. In R 3.6.0 the default algorithm that `sample()` uses to turn a seed into draws changed. Code that set a seed and drew a sample therefore obtained different draws before and after the release, with no error and no warning. In R 4.0.0 the `stringsAsFactors` default flipped from `TRUE` to `FALSE`. Code reading a data frame then silently produced character columns where it once produced factors. Any code that depended on factor-specific

Draft, November 13, 2026. Content may change.

behavior, such as the ordering of levels or a function that requires a factor, behaved differently as a result. Neither change was a bug; both were improvements. Both would nonetheless break the exact reproduction of an analysis written under the old default, and neither would announce itself.

These are instances of a general phenomenon. A computational result depends not only on the code and the data but on a stack of determinants beneath them. The stack includes the operating system and its system libraries, the numerical libraries that implement linear algebra, the language runtime, the exact versions of every package, and the session configuration. That configuration includes the random-number kind, the contrasts, the number of printed digits, and the locale and time zone. Underneath all of it sits the hardware. A published script pins almost none of these. It records the analysis logic and, if the reader is fortunate, the data. The remaining layers are left to whatever the next machine happens to provide. Reproducibility fails in the gap between what the script fixes and what the result depends on, and the task of a reproducibility toolchain is to close that gap, layer by layer. But how far down the stack has a given effort actually reached? That is the organizing question of this book.

1.5. The tools, and the framework this book uses

The R ecosystem has, over two decades, produced mature tools for nearly every layer of that stack. Literate programming, through `knitr`, R Markdown, and now Quarto (Allaire et al., 2024), binds prose to the code that produces it. `renv` (Ushey & Wickham, 2026) pins package versions in a lockfile. Docker, through Rocker's R-specific base images (Boettiger, 2015; Nüst, Eddelbuettel, et al., 2020), pins the operating system and its libraries. Continuous integration re-runs the whole compendium on a clean machine to confirm that the pinned layers actually suffice. Alternatives exist at every layer: Nix and its R front-end `rix` in place of `renv` (Dolstra et al., 2004; Rodrigues & Baumann, 2024), Podman or Apptainer in place of Docker (Kurtzer et al., 2017), GitLab in place of GitHub. We introduce each tool where the chapter ahead needs it, not in one early catalog. Package pinning appears in Chapter 6, and the environment and its alternatives in Chapter 7 and Chapter 8. Chapter 5 treats the compendium structure that Gentleman & Temple Lang (2007) and Marwick et al. (2018)'s `rrtools` established.

Draft, November 13, 2026. Content may change.

Finally, Chapter 10, Chapter 11, and Chapter 12 treat the workflow, quality, and validation layer.

The difficulty lies in composing this tooling by hand. `renv` says nothing about the operating system, and Docker says nothing about which package versions are installed inside it. Neither says anything about session configuration. A researcher who wants a genuinely reproducible analysis must therefore assemble all of it correctly, for every project, by hand. The running example of this book uses `zzcollab`, a framework that assembles the triad (`renv` for packages, Docker for the environment, continuous integration for validation) into one workflow. A biostatistician thereby obtains the composition without building it from scratch. Each chapter ahead develops the part of `zzcollab` relevant to its own layer, against the concept it has just introduced.

Like every tool this book covers, `zzcollab` is held to the standard it teaches. By its own accounting it firmly pins a part of the determinant stack, namely the environment, the packages, the source, and the locale and time zone. It leaves the rest (the numerical libraries, the random-number discipline, the graphics device, intrinsic non-determinism) to the base image or to convention. Reproducibility tooling in general has an adoption problem more than a capability problem. Saju et al. (2025) found that most deposited code supplements carry no dependency description at all, even though mature, free, documented tools for that exact problem have existed for years. The task of this book is therefore to make existing capability teachable and routine. It is also to state plainly, at each level of the ladder, what a given effort has secured and what remains exposed, including its own.

1.6. The plan of this book

The book is organized in four parts, and three ideas recur in every chapter. The first is the ladder of reproducibility levels (L0 locatable, L1 pinned packages, L2 pinned environment, L3 verified), which we introduce in Chapter 3 and use thereafter as the common scale. The second is the distinction between capturing reproducibility and validating it. That is, pinning is capture, while continuous integration and verification are validation. The third is the research compendium as the unit of reproducible work, introduced in

Draft, November 13, 2026. Content may change.

Chapter 5. The book is written for a one-quarter, ten-week course. Not every chapter fits in ten weeks, and an instructor teaching on that schedule should expect to treat some material as optional. The four parts proceed as follows.

- **Part I, Foundations.** The reproducibility crisis and its computational character (Chapter 2); the levels of reproducibility and the determinants that set them (Chapter 3); version control with git as the foundation of a locatable analysis (Chapter 4); and the research compendium (Chapter 5).
- **Part II, Capturing the environment.** Pinning packages with a lockfile (Chapter 6); pinning the operating system and runtime with a container (Chapter 7); and the alternatives, from Nix to Podman to Apptainer (Chapter 8).
- **Part III, Validation.** The practices that let a pinned analysis be re-run and checked: literate programming and the rendered report (Chapter 9); automating the analysis with make and targets (Chapter 10); testing and code quality (Chapter 11); and continuous integration and verification (Chapter 12).
- **Part IV, Collaboration and stewardship.** Sharing a compendium with a team and with the world (Chapter 13).

1.7. How to use this book

The book is meant to be worked, not only read. Each chapter develops a concept and then builds it, in R and at the shell, inside a running example compendium. The reader who follows along will finish with a reproducible project of their own. More durably, the reader will acquire the judgment to reason about reproducibility with any toolchain, and to state without flinching the level a given analysis has actually reached.

Seven habits recur across the chapters that follow. A chapter's own rules of thumb will cite a habit by number rather than restate it. The closing chapter recalls them once the argument of the book is complete.

Draft, November 13, 2026. Content may change.

Seven habits, and a motto

Claim only the level you have verified.

1. Commit before you compute.
2. Pin what you ran, not what is current.
3. Never commit, deposit, or paste restricted data.
4. Make every gate able to fail, and watch it fail once.
5. Compare against a recorded baseline, with a stated tolerance.
6. Rebuild from clean, on a machine that is not yours.
7. Write the honest level down.

Draft, November 13, 2026. Content may change.

Part I.

Foundations

Draft, November 13, 2026. Content may change.

2. The Reproducibility Crisis

More than 70% of researchers have tried and failed to reproduce another scientist's experiments, and more than half have failed to reproduce their own experiments.

Monya Baker, *1,500 Scientists Lift the Lid on Reproducibility* (2016)

By the end of this chapter you will be able to look at a re-execution failure like the auditor's below and say, concretely, whether the fault is statistical or computational. You will also be able to name the specific silent default most likely responsible, and to place the original analysis on the ladder this book develops in full in Chapter 3.

Before you start

Reading time. About 35 minutes.
Prerequisites. Chapter 1 (Chapter 1), for the book's first, informal pass at the determinant stack and the silent-default examples this chapter now develops in full; no other chapter is assumed. Statistically, nothing beyond reading a regression output, a hazard ratio, or an odds ratio with its confidence interval is required. No R fluency beyond reading a short script is assumed; the chapter's own code chunks are simple enough to follow without having written R before.
By the end of this chapter you will be able to answer:

- Can you summarize the empirical evidence that a large fraction of published health-sciences research cannot be regenerated, and distinguish the studies that measure replication from those that measure computational re-execution?
- Can you state the distinction the National Academies draw between reproducibility and replicability, name the third term the wider literature adds, and say why the terminology is contested?
- Can you explain why many re-execution failures are computa-

tional rather than statistical, naming at least one silent default change in R as a concrete case?
- Can you describe, in first outline, the stack of determinants beneath a computational result, and locate where in that stack a published script leaves the outcome unpinned?
- Can you argue why reproducibility is a floor rather than a ceiling, and give an example of a reproducible analysis that is nonetheless wrong?

Outline.

- Orientation: an auditor's regenerated analysis that moved in the third decimal place.
- Where this sits on the stack: which layers this chapter diagnoses.
- The analyst's contribution: three judgments no tool automates.
- The evidence that re-execution fails: replication studies against re-execution studies.
- Three words that are not synonyms: reproducibility, replicability, robustness.
- Why the failures are computational, not statistical: two silent defaults in R itself.
- A reproducible analysis can still be wrong.
- Worked example: the same seed, two samplers.
- The cohort, continued: a silent default made explicit in `load_cohort()`.

Check 1: from earlier chapters

A colleague re-runs a cohort script deposited in 2019, written under R 3.6, on a laptop with a current release of R. The script finishes without error, but `read.csv()` now returns `sex` as a character column where the original run had a factor. The colleague reports that the finding 'did not replicate'. Using Chapter 1, is 'replicate' the right word? Which layer of the determinant stack is at fault, and which tool would have pinned it?
The answer is on page 63.

Draft, November 13, 2026. Content may change.

2.1. Orientation

Consider a situation that every quantitative health scientist eventually meets. A biostatistician on a clinical-trial team runs the primary analysis and obtains the treatment effect and its confidence interval. The biostatistician exports the table that will anchor the regulatory submission and files the script. Two years later, during an audit, an inspector asks that the analysis be regenerated from the archived code and data. On a fresh machine the script errors on the first `library()` call. Once the missing package is located and installed, the analysis runs. The point estimate has moved in the third decimal place, however, and one confidence interval no longer excludes the null. Nothing in the statistical reasoning was wrong. What failed was everything around the statistics: a package that changed beneath the code, a default that shifted between releases of R, a system library present on the first machine and absent on the second.

This book exists to prevent that failure, and in this chapter we shall establish that the failure is real, common, and specifically computational. In a health science, a result that cannot be regenerated cannot be audited. An analysis that a regulator cannot reproduce is one a regulator may decline to trust. A clinical recommendation built on a number that no one can recompute rests on faith rather than evidence.

Throughout the book we place an analysis on a four-rung ladder, and we shall begin to use its vocabulary at once. An analysis is L0, locatable, when it is under version control and its source can be found, but nothing computational is pinned. It is L1, pinned packages, when the exact version of every package is fixed by a lockfile. It is L2, pinned environment, when the operating system, the language runtime, and the system libraries are fixed as well, usually inside a container. It is L3, verified, when the recorded computational outputs have been shown to regenerate within a stated tolerance from a clean rebuild, checked by a baseline test. The auditor's failure above is the failure of an analysis that reached, at best, L0: the code could be found, and nothing else about the computation had been fixed. The ladder is developed in full in Chapter 3; here we use it only to name what was missing.

The chapter proceeds as follows. After locating the chapter on the determinant stack and setting out the judgments that belong to the analyst, we

Draft, November 13, 2026. Content may change.

review the evidence that re-execution fails, separating replication studies from re-execution studies. We then fix three terms that are often confused: reproducibility, replicability, and robustness. Next we show, using two silent defaults in R itself, why the failures are computational and not statistical. We also argue that a reproducible analysis can still be wrong. Lastly, a worked example compares the same seed under two samplers, and the running cohort makes one silent default explicit in its loading function.

Check 2: naming the rung

The auditor's script ran, after installing a missing package, but returned a moved estimate. Using only the L0-L3 vocabulary introduced above, what is the most that can honestly be said about the rung this analysis had reached before the audit?
The answer is on page 64.

2.2. Where this sits on the stack

This chapter diagnoses rather than pins, and since Chapter 3 has not yet drawn the distinction between capturing a layer and validating one, the figure below claims neither. The prose names the layers of the determinant stack most responsible for the re-execution failures the evidence below documents. These are the R runtime itself and the session configuration riding on top of it. The RNG-kind comparison below and the `stringsAsFactors` comparison in 'The cohort, continued' each show an unpinned layer behaving exactly as badly as the chapter claims. No lockfile, no container, and no baseline test appears until later chapters, so Figure 2.1 highlights nothing. Its point is the blank stack itself, which Chapter 3 gives a name to and every later chapter's figure will mark.

```
plot_determinant_stack(
  title = 'A diagnosis, not yet a capture or a validation')
```


Draft, November 13, 2026. Content may change.

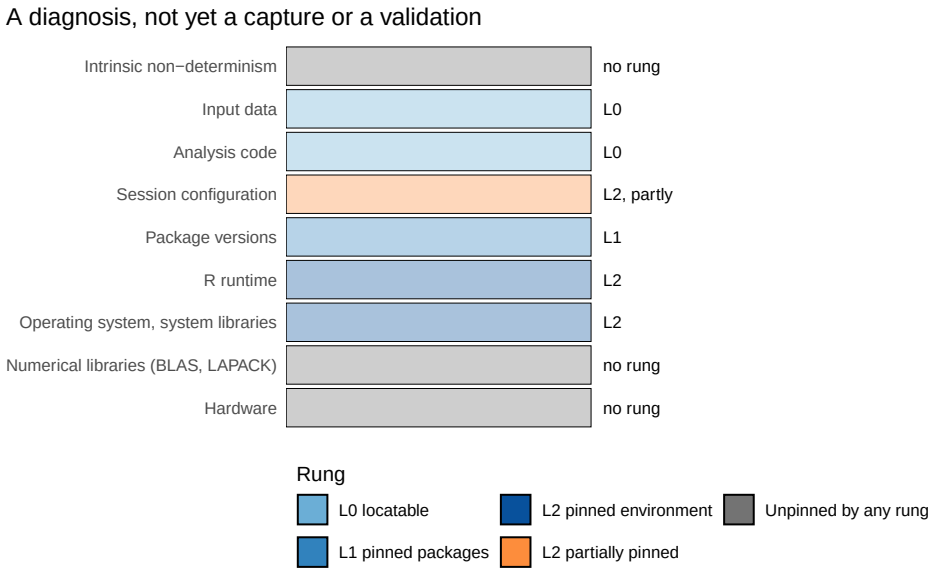


Figure 2.1.: The determinant stack of Chapter 3, with no layer captured or validated. This chapter diagnoses, in prose, why re-execution fails on two of these layers, the R runtime and the session configuration, but pins nothing and checks nothing pinned; the chapters that follow are the ones that mark this figure.

2.3. The analyst's contribution

Before turning to the evidence, we note three judgments at the center of this chapter that belong to the analyst and are automated by no tool.

1. **Naming the failure mode.** When a result will not regenerate, or regenerates differently, the analyst must decide what kind of discrepancy it is. It may be statistical (the method was inappropriate, the inference was misdrawn) or computational (the code is sound but its environment has shifted). The remedy differs entirely by case.

2. **Matching the claim to the level reached.** A reproducibility claim is a scientific claim and must be calibrated to evidence. The analyst decides, and states honestly, which rung of the ladder an analysis has

Draft, November 13, 2026. Content may change.

actually reached, and resists the temptation to assert a level higher than the one secured.

3. **Deciding what must be pinned.** Not every determinant matters equally for every analysis. A simulation study lives or dies by its random-number configuration; a geospatial pipeline depends on system libraries that a survey analysis never touches. Judging which layers of the stack are load-bearing for the analysis at hand is a decision the analyst makes before any tool is invoked.

i The vocabulary of this chapter

New to these terms? Each is defined again on first use.

- **Computational reproducibility.** Obtaining the same numerical results from the same data and the same code, on another machine or at a later time.
- **Replicability.** Obtaining consistent findings from a new study, with new data, aimed at the same question.
- **Robustness.** Reaching the same qualitative conclusion from the same data under different but defensible analytical choices.
- **Determinant stack.** The layered set of things a computational result depends on, from the hardware and operating system, through the system and numerical libraries, the language runtime, and the package versions, up to the session configuration and the code itself.
- **Silent default.** A behavior that software applies unless told otherwise, and that can change between software versions without any error or warning to the analyst.
- **Seed and RNG kind.** The seed (`set.seed`) is the fixed starting point that makes a random draw repeatable; the RNG kind is the generator algorithm that turns that seed into numbers, and its default has changed across R versions.
- **Package churn (code rot).** The gradual breakage of once-working code as the packages beneath it are updated over time.
- **Research compendium.** The self-describing bundle of code, data, and text that travels as the unit of a reproducible analysis, developed in Chapter 5.

Draft, November 13, 2026. Content may change.

Check 3: whose judgment

A reviewer complains that a manuscript claims ‘L2 reproducibility’ without saying whether the random-number kind was fixed. Which of the three analyst judgments above does this complaint invoke?
The answer is on page 64.

2.4. The evidence that re-execution fails

We begin with the large replication studies, because they set the scale of the problem. We then turn to the narrower and, for our purposes, more damning studies that isolate the computational failures.

In psychology, the Open Science Collaboration re-ran one hundred published studies. Its teams judged the original finding replicated in only thirty-nine of them, and only thirty-six percent of the replications reached statistical significance (Open Science Collaboration, 2015). In preclinical oncology, Begley and Ellis, working from within industry, reported that only six of fifty-three landmark findings could be reproduced, a success rate close to one in ten (Begley & Ellis, 2012). These figures were not, at first, universally believed. But the suspicion that the published literature was less solid than it appeared already had a theoretical footing. Ioannidis had argued, given the prevailing rates of low statistical power, analytical flexibility, and publication bias, that most published research findings are false (Ioannidis, 2005). When Baker later surveyed more than fifteen hundred researchers for Nature, a majority agreed that science was facing a reproducibility crisis. More than seventy percent reported that they had tried and failed to reproduce another scientist’s experiments (Baker, 2016).

These are, for the most part, failures of replication, at the scientific end of a spectrum we shall shortly define, and their causes are many. But a growing body of work isolates the specifically computational failures. Here the question is not whether a finding holds up in a new sample, but merely whether the original analysis can be made to run a second time. The evidence there is starker, because the failures are more avoidable. Trisovic and colleagues re-ran more than nine thousand R files deposited alongside published articles in the Harvard Dataverse. Roughly one in four executed

Draft, November 13, 2026. Content may change.

without error, before any question of whether the output matched the published result (Trisovic et al., 2022). Saju and colleagues examined R code supplements archived on the Open Science Framework. Only about a quarter of the R scripts ran without error, even in environments the authors rebuilt, and only one project in six ran in full. Close to ninety-nine percent of the projects carried no formal description of their dependencies at all (Saju et al., 2025).

We see, then, that the replication studies tell us that scientific findings are fragile. The re-execution studies tell us something more specific and more remediable, namely that a large fraction of published analyses cannot be regenerated even in principle. The reason is that the computational scaffolding needed to run them was never captured. The tooling in this book addresses the second failure.

Check 4: two kinds of failure

A colleague re-runs the archived code from a published cohort study, obtains a number that differs from the printed one, and concludes that the paper's finding does not replicate. Why is that conclusion premature, and which of the studies cited above speaks most directly to the colleague's situation?
The answer is on page 64.

2.5. Three words that are not synonyms

Before proceeding we need to fix our terms. The vocabulary of this field is genuinely contested, and a reader who does not fix it will be confused by the literature. Goodman and colleagues surveyed the usage and found that the same words carry opposite meanings in different research communities: one field's 'reproducibility' is another field's 'replication' (Goodman et al., 2016). We adopt the distinctions of the National Academies, whose 2019 consensus report offers the settlement most widely followed in the health sciences (National Academies of Sciences, Engineering, and Medicine, 2019).

Reproducibility is obtaining consistent computational results using the same input data, the same computational steps, methods, and code, and the same conditions of analysis. It is a statement about re-execution, and it is

Draft, November 13, 2026. Content may change.

the subject of this book. *Replicability* is obtaining consistent results across studies aimed at the same question, each collecting its own data. It is a statement about the durability of a scientific finding. To these two the wider literature, notably the Turing Way (The Turing Way Community, 2022), adds a third, *robustness*, obtaining consistent conclusions from the same data when the analytical choices are varied within the range a competent analyst would defend. The three are independent. An analysis can be perfectly reproducible and entirely wrong, faithfully regenerating the same mistaken number on every machine. A finding can replicate across laboratories while no single analysis in the set is computationally reproducible. And a result can be reproducible yet fragile to a defensible change in model specification.

How, then, are the three terms to be kept apart? They sort cleanly once we notice that each answers the same question: what was held fixed, the data or the analysis. Figure 2.2 lays the possibilities out as a grid, a presentation the Turing Way has made the common one (The Turing Way Community, 2022). The fourth cell, in which both the data and the analysis change, is generalizability. We name it here for completeness and do not treat it.

```
library(ggplot2)
library(tibble)

grid <- tibble(
  data     = rep(c('Same data', 'New data'), each = 2),
  analysis = rep(c('Same analysis', 'Varied analysis'), 2),
  term = c('Reproducibility',
           'Robustness',
           'Replicability',
           'Generalizability'),
  gloss = c('the same code re-run',
            'defensible analytic\nchoices varied',
            'a new study of the\nsame question',
            'not treated in\nthis book'),
  focus = c(TRUE, FALSE, FALSE, FALSE)
)

grid$data <- factor(grid$data, levels = c('Same data', 'New
 ↪  data'))
```


Draft, November 13, 2026. Content may change.

```
grid$analysis <- factor(grid$analysis,
                        levels = c('Varied analysis', 'Same
                         ↪  analysis'))

ggplot(grid, aes(x = data, y = analysis)) +
  geom_tile(aes(fill = focus), color = 'white', linewidth =
   ↪  2) +
  geom_text(aes(label = term), vjust = -0.6,
            fontface = 'bold', size = 3.9) +
  geom_text(aes(label = gloss), vjust = 1.25, size = 3,
            lineheight = 0.95, color = 'grey25') +
  scale_fill_manual(values = c('TRUE' = '#cfe6e3',
                               'FALSE' = '#f2f2f2'),
                    guide = 'none') +
  labs(x = NULL, y = NULL) +
  theme_minimal(base_size = 10) +
  theme(
    panel.grid = element_blank(),
    axis.text = element_text(face = 'bold', size = 9.5)
  )
```


Draft, November 13, 2026. Content may change.

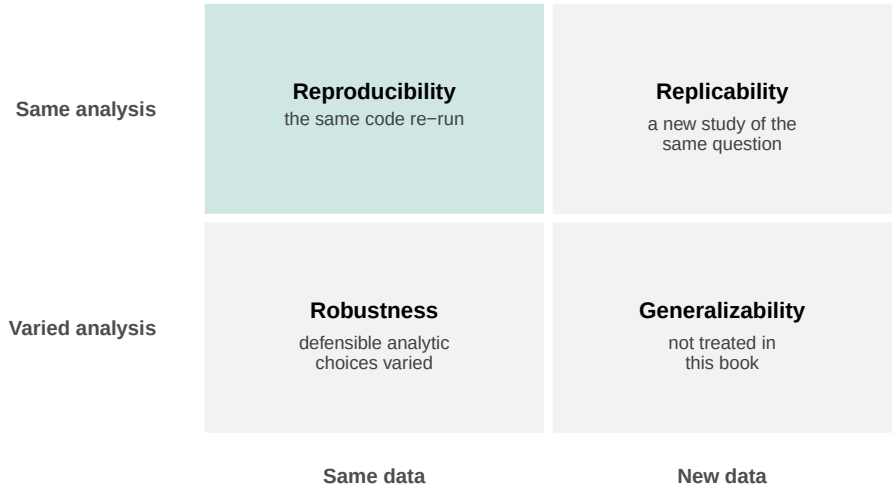


Figure 2.2.: Reproducibility, replicability, and robustness distinguished by what is held fixed. The columns vary the data and the rows vary the analysis, so each cell names the claim supported when a consistent result survives that particular change. Only the shaded cell, in which both the data and the analysis are held fixed, is the subject of this book. The grid and the robustness cell follow the Turing Way (The Turing Way Community, 2022); the definitions of reproducibility and replicability are those of the National Academies (National Academies of Sciences, Engineering, and Medicine, 2019) adopted above.

The idea that computational work should be reproducible in the narrow, re-execution sense is older than the current crisis. It is usually traced to Claerbout and his colleagues in geophysics. In the early 1990s they built their books and reports so that every figure could be rebuilt from the archived code and data (Buckheit & Donoho, 1995; Claerbout & Karrenbach, 1992). Their standard was demanding. The phrasing that has become the field's motto is Buckheit and Donoho's paraphrase of it (Buckheit & Donoho, 1995; Claerbout & Karrenbach, 1992): the published article is merely the advertisement of the scholarship, and the scholarship itself is the full computational apparatus that produced the figures.

Reproducibility so understood is a spectrum. Peng arranged computational

Draft, November 13, 2026. Content may change.

work along a scale, from the merely published, through code available, code and data available, and linked and executable, to fully replicable. He argued that reproducibility can serve as a minimum standard for judging a scientific claim when full independent replication is not possible (Peng, 2011; Peng et al., 2006). We take from Peng the idea that reproducibility comes in degrees, and that honesty about which degree one has reached is itself a scientific virtue. We shall develop that idea into the levels ladder of Chapter 3.

Check 5: three words, one study

A cardiology registry reports an effect. A second registry collects its own patients and finds the same effect under a slightly different covariate adjustment. A third team cannot regenerate the first registry's published table from its deposited code. Name which of reproducibility, replicability, and robustness each observation bears on.
The answer is on page 64.

2.6. Why the failures are computational, not statistical

We now ask why re-execution fails, and the reasons point directly at the remedy. Consider two changes to R itself, each silent, each an improvement, and each capable of altering a published result without any error or warning to announce it.

In R 3.6.0, released in 2019, the default algorithm that turns a seed into a random sample was corrected. The old sampler introduced a slight bias in `sample()`; the new one removed it. The consequence for reproducibility is direct. Code that set a seed and drew a sample obtained one sequence of draws before the release and a different sequence after it, with identical code and an identical seed (R Core Team, 2019). Every bootstrap resample, every cross-validation fold assignment, and every permutation test built on `sample()`, directly or through a function that calls it, draws differently under the new default than under the old one. A simulation that draws from `rnorm()`, `rpois()`, or another generator that does not route through

Draft, November 13, 2026. Content may change.

`sample()` is unaffected; the 2019 correction touched one generator, not every source of randomness in R.

Then, in R 4.0.0, released in 2020, the `stringsAsFactors` default flipped from `TRUE` to `FALSE`. Reading a data frame now silently produced character columns where it once produced factors. This affected any code that depended on factor-specific behavior, such as the ordering of levels or a function that requires a factor (R Core Team, 2020). Neither change was a bug; both were deliberate improvements. Both would nonetheless break the exact reproduction of an analysis written under the old default, and neither would say a word.

To these language-level shifts we must add the steadier erosion of package churn, the phenomenon by which working code rots as the packages beneath it are updated. A modern analysis may depend on dozens or hundreds of packages, each with its own release history and its own breaking changes. Much of this churn, along with missing files and hard-coded paths, is what Trisovic and colleagues measured when they found that most deposited R scripts did not run when re-executed, often years after the original deposit (Trisovic et al., 2022).

These are instances of a general phenomenon that we shall develop fully in Chapter 3. A computational result depends not only on the code and the data but on a stack of determinants beneath them. The stack includes the operating system and its system libraries, the numerical libraries that implement linear algebra, the language runtime, and the exact versions of every package. It also includes the session configuration (the random-number kind, the contrasts, the number of printed digits, the locale and time zone) and, underneath all of it, the hardware. A published script pins almost none of these. It records the analysis logic and, if the reader is fortunate, the data. The remaining layers are left to whatever the next machine happens to provide. Reproducibility fails in the gap between what the script fixes and what the result depends on. The organizing question of this book is how far down that stack a given effort has actually reached.

This diagnosis, that the failures are computational rather than statistical, is what makes the problem tractable. A statistical error requires a scientific judgment to correct; a computational failure requires only that the determinant stack be captured, layer by layer, so that the same computation can

Draft, November 13, 2026. Content may change.

be run again. Computational hygiene of that kind can be taught, tooled, and enforced.

The same exposure applies to every kind of computational output, including outputs with no polished manuscript. A research compendium comes in several archetypes, developed in full in Chapter 5: a data-analysis project, a scholarly manuscript, an R package, a simulation study, and a blog of many rendered posts. Each varies the directory layout, whether a rendered report exists at all, and which validation gates apply. However, all share the same determinant stack, and so all are exposed to the same silent shifts. An R package with no report to render can still be broken by a changed `sample.kind`. A simulation study that resamples, permutes, or randomizes through `sample()` is more exposed to the RNG change than any manuscript, since its whole output is a function of those draws. And a teaching blog that renders forty posts inherits the churn of every package any post loads.

Check 6: locating the failure

An analyst reports that a script producing a principal-components plot gives visibly different loadings on a colleague's laptop, although both installed the same package versions from the same lockfile. Setting a seed changes nothing. Which layer of the determinant stack is the most likely culprit, and why does the lockfile not protect against it? *The answer is on page 65.*

2.7. A reproducible analysis can still be wrong

Nothing in this book claims that a reproducible analysis is a correct one. The point is pressed by Leek and Peng, who observe that reproducibility guarantees only that a computation can be regenerated, not that the computation was the right one to perform (Leek & Peng, 2015). A script that faithfully regenerates a mistaken p-value on every machine, forever, is perfectly reproducible and perfectly wrong.

It follows that reproducibility is a floor and not a ceiling. It is the minimum standard beneath which a computational claim cannot be assessed at all, because a result no one can regenerate is a result no one can check. Above

Draft, November 13, 2026. Content may change.

that floor sit all the harder questions, of study design, of confounding, of model choice, of inference, that reproducibility does not touch. We insist on the floor precisely so that the harder questions become answerable. That is, an analysis that can be re-run is an analysis whose statistical choices can be inspected, criticized, and, where necessary, corrected.

Check 7: floor, not ceiling

A colleague argues that because their analysis is containerized, pinned, and verified against a baseline (L3), it must also be correct. What is wrong with the inference, and which cited argument refutes it directly? *The answer is on page 65.*

! The honest level

This chapter's own practice reaches no rung at all. It diagnoses the R-runtime and session-configuration layers in its worked demonstrations without capturing either. The claim it can honestly make is therefore diagnostic, not a placement on the ladder that Chapter 3 defines.

2.8. Worked example

To make the RNG change concrete, we reproduce it here in a few lines of base R. The generator that turns a seed into a sample is selectable through `RNGkind()`. R still ships both the pre-3.6.0 sampler, named `Rounding`, and the current one, named `Rejection`. We can therefore hold the seed fixed at a single value and vary only the sampler. We then observe directly that the same seed does not give the same draws when the RNG kind differs. The function below sets the sampler, sets the seed, and shuffles the integers one through ten. We call it under each sampler and restore the session's original generator afterward, so that nothing downstream is disturbed.

```
draw_sample <- function(sample_kind) {
  suppressWarnings({
    RNGkind(sample.kind = sample_kind)
    set.seed(123)
```


Draft, November 13, 2026. Content may change.

```
    sample(1:10)
  })
}

original_rng <- RNGkind()

rounding_draw  <- draw_sample('Rounding')
rejection_draw <- draw_sample('Rejection')

suppressWarnings(RNGkind(
  kind        = original_rng[1],
  normal.kind = original_rng[2],
  sample.kind = original_rng[3]
))

comparison <- data.frame(
  sampler = c('Rounding (R < 3.6.0)',
              'Rejection (R >= 3.6.0)'),
  draws = c(
    paste(head(rounding_draw),  collapse = ', '),
    paste(head(rejection_draw), collapse = ', ')
  )
)

knitr::kable(
  comparison,
  col.names = c('RNG sample kind', 'First six draws'),
  align = 'l'
)
```

Table 2.1.: The same seed yields the same draws only under the same RNG kind. Each row shows the first six values of a shuffled sequence of the integers one through ten, drawn under seed 123.

RNG sample kind	First six draws
Rounding (R < 3.6.0)	3, 8, 4, 7, 6, 1
Rejection (R >= 3.6.0)	3, 10, 2, 8, 6, 9

Draft, November 13, 2026. Content may change.

The two rows of Table 2.1 share a seed and share a line of code, `set.seed(123); sample(1:10)`, and yet they diverge almost immediately, coinciding only sporadically thereafter. An analyst who set a seed under R 3.5, confident that the seed made the analysis reproducible, would find the draws silently changed on upgrading to R 3.6 or later. The draws depend on the seed together with the RNG kind. This is the determinant stack in miniature: the code fixed one layer, the seed, and left another, the sampler, to whatever the running version of R supplied. Two things set the sampler, and they are fixed at different points. The R release sets the *default* kind, so the default belongs to the R runtime, and pinning the R version, which we reach at L2 with a container, fixes it. Code or a project `.Rprofile` can *override* that default, however, and an override is session configuration, fixed only by an explicit `RNGkind()` call in the analysis code. Recording the RNG kind alongside the seed makes either gap visible in the interim.

Verification

To see this hold on your own machine, run the `draw_sample()` chunk above as written and confirm the two rows of Table 2.1 disagree. To make the claim fail on purpose, delete the `RNGkind(sample.kind = ...)` call from `draw_sample()` so both calls inherit whatever sampler the session already has. The two rows will then agree. That agreement is the symptom of the same hazard, since both draws are now silently using one unrecorded sampler rather than two declared ones.

Check 8: the seed was never enough

Table 2.1 holds the seed and the code fixed, and varies only the RNG kind. Which term in the chapter's own vocabulary names what the seed, by itself, fails to pin?
The answer is on page 65.

2.9. The cohort, continued

The worked example above manufactures the R 3.6.0 sampler change on purpose, by selecting both samplers explicitly. This book's own running cohort compendium already carries a smaller, quieter instance of the same

Draft, November 13, 2026. Content may change.

hazard, the `stringsAsFactors` default of R 4.0.0. The file `R/cohort.R` closes it the way this chapter has argued an analyst should, by setting the argument explicitly rather than trusting whatever the installed R happens to default to. The extract itself is not in the repository, only the script that generates it, so a reader with a fresh clone first runs `Rscript analysis/scripts/simulate_cohort.R` from the top directory to write `analysis/data/raw_data/cohort_raw.csv`.

```
raw <- load_cohort('../data/raw_data/cohort_raw.csv')
class(raw$sex)
#> [1] "character"

legacy <-
 ↪  utils::read.csv('../data/raw_data/cohort_raw.csv',
                            stringsAsFactors = TRUE)
class(legacy$sex)
#> [1] "factor"
```

`load_cohort()` calls `read.csv()` with `stringsAsFactors = FALSE` written into the function itself, so `raw$sex` is a character column regardless of which R version runs it. Calling `read.csv()` on the identical file with the pre-4.0.0 default made explicit, `stringsAsFactors = TRUE`, returns the same column as a factor instead. Every function downstream that cares about the difference (an `ifelse()` that compares against a string, a `table()` that orders levels) would see two different objects from the one file. Which object it saw would depend on nothing but which R release happened to be installed when the bare call ran. `load_cohort()` pins the choice in the code. That is a capture of the analysis-code layer, not of the R-runtime layer itself, which is why this comparison still counts as a diagnosis and not as the chapter's first capture.

> Check 9: which layer is closed
>
> `load_cohort()` hard-codes `stringsAsFactors = FALSE`. Does this pin the R-runtime layer, and if not, what does it pin?
> *The answer is on page 65.*

Draft, November 13, 2026. Content may change.

2.10. Collaborating with an LLM

We turn finally to the use of a large language model as an assistant. A model is a capable assistant for the material of this chapter, provided the analyst treats its output as a draft to be verified rather than an authority to be trusted.

! Restricted data

One rule comes first (habit 3). Never paste or upload protected health information, identifiable records, or data held under a data use agreement, nor outputs derived from them, into a model or a coding agent unless three conditions all hold. The institution has approved that endpoint for that class of data. The IRB protocol and the consent permit the processing. The data use agreement, or the provider's terms, permits it. If any of the three is unclear, the answer is no.

The third condition is the one most often missed. Institutional approval of a tool does not override the terms under which a data provider released the data. NIH, for example, treats sharing controlled-access data through prompts with public generative AI tools, which it glosses as third-party tools, as a breach of the data use certification. It treats a model trained on such data as a data derivative, to be shared only with other approved users and not kept after the project closes (National Institutes of Health, 2025a).

Describe the structure instead: the column names and types, or a small synthetic example generated from the data dictionary, never edited from real rows. A coding agent sees whatever files are in its working tree, so run it in a checkout or container where the real data is not mounted.

An agent also acts, so confine it to a branch and keep merges, pushes, and deposits for yourself (Section 2.10.1).

Three uses recur, and each carries a specific verification burden.

Prompt. 'Explain the difference between reproducibility, replicability, and robustness, and give a health-sciences example of each.'

Watch for. Check whether the answer conflates the first two terms, or adopts the reversed convention that some fields use; both usages are common in

the literature. Check also that each example carries the label it deserves, since a plausible example can still be mislabeled.

Verification. Check the definitions against the National Academies framing (National Academies of Sciences, Engineering, and Medicine, 2019) adopted in this chapter, and confirm that the example labeled 'reproducibility' concerns re-executing the same code on the same data, not re-collecting data. Do not accept the term assignment without this check.

A better prompt. 'Using the three definitions pasted below, which follow the National Academies report, give one health-sciences example each of a failure of reproducibility, of replicability, and of robustness. For each, state what was held fixed (code, data, analysis) and what changed. Do not use any other convention. After each example, quote the sentence of the definition it satisfies, so that I can check the label.'

Prompt. 'My results changed after I upgraded R. What could have caused this and how do I diagnose it?'

Watch for. The model may propose confident but untested causes, or invent version numbers and release dates for the change it blames. A chat model has no access to the session and cannot know which layer actually moved. A coding agent that can run R in the project can inspect `sessionInfo()` itself. However, it may still report a cause it has not tested, and it can change the environment while investigating it, for example by reinstalling a package or updating the lockfile.

Verification. Treat every proposed cause as a hypothesis to test against your own `sessionInfo()` and the actual release notes, not as a diagnosis. Confirm the specific mechanism (for the RNG change, by the demonstration above; for `stringsAsFactors`, by inspecting a column's class) before acting on it. If an agent did the investigating, run `git diff` on `renv.lock` and the code afterward, so that a repair it made silently is seen before it is committed. The lockfile is a protected path, and the decision to change it stays with you (Section 2.10.1).

Prompt. 'Write a paragraph for my methods section stating that my analysis is fully reproducible.'

Watch for. Check whether the draft asserts a level of reproducibility that the project has not reached. The model has not audited the compendium;

Draft, November 13, 2026. Content may change.

it has only the request. A confident claim of full reproducibility is exactly the overstatement the honest-level caution warns against.

Verification. Establish which rung of the ladder the analysis has genuinely reached, and edit the paragraph to claim that rung and no higher. The claim is your responsibility, not the model's.

2.10.1. What to delegate to a coding agent

A coding agent does not only read; it runs commands, edits files, commits, and pushes if its credentials allow. Settle the division of labor before it starts.

The agent may scaffold, edit code on a branch, run the tests, and open a pull request (Chapter 4); the analyst reviews and merges. Three cautions govern that review. First, a self-report is not an observation. 'The tests pass' is a claim, so read the run log yourself. Second, an agent asked to make a failure go away can change the check instead of the code: an expected value, a baseline, a tolerance, or a CI step allowed to fail. Treat as protected the paths that hold checks and inputs, in this book's layout `inst/tinytest/`, `inst/baseline/`, `renv.lock`, `.github/workflows/`, and `analysis/data/raw_data/`, and read every diff that touches them line by line. Third, irreversible actions stay with the analyst: pushing to a registry, minting a DOI, depositing, rewriting history, force-pushing, and merging. Withhold the credentials that would let the agent take them.

An agent can also be steered by what it reads. A downloaded README, an issue comment, a CI log, or a data file can carry instructions that the agent follows as if they were yours. OWASP lists this risk, prompt injection, first among the risks of LLM applications and states that no fool-proof prevention is known (OWASP GenAI Security Project, 2025a). Its entry on excessive agency recommends least privilege and human approval of high-impact actions (OWASP GenAI Security Project, 2025b), which is the division above.

The data boundary has to be the filesystem. An agent sees whatever its working tree holds, and an ignore setting is a request the agent is expected to honor, not a control the operating system enforces. Run the agent where

Draft, November 13, 2026. Content may change.

the real data are not mounted, against synthetic rows generated from the data dictionary.

2.10.2. Disclosing and recording model use

If a large language model contributed to the code, prose, or analysis behind a paper, disclose that use. The International Committee of Medical Journal Editors bars AI tools from authorship, because a tool cannot be responsible for the accuracy, integrity, and originality of the work, and places that responsibility on the human authors (International Committee of Medical Journal Editors, 2026). It also asks authors to describe their use of AI at submission, in the cover letter and in the appropriate section of the manuscript. Journals differ about which section that is. Place the statement where the target journal's instructions say, and name the use in the cover letter as well.

A usable statement has the parts of a package citation:

> `<tool name>`, `<version or model identifier>`, `<provider>`, used `<dates>` to `<purpose>` on `<files or sections>`; output reviewed and verified by `<author>`.

A statement that says only 'written with AI assistance' gives a future reader no way to audit what was used. Give the identifier exactly as the interface or the API reported it, as 'R 4.4.1, glmnet 4.1-8' gives a package, not an approximate version. Unlike a package version, however, a model's output is not a fixed, reproducible artifact. Model behavior drifts across versions, and providers retire and replace model identifiers over time. Even repeated calls to one unchanged version are not guaranteed to return identical output. The text a model produced in January may therefore not be the text it would produce from the same prompt in June, or even twice in the same afternoon. The identifier narrows the search, the way a `sessionInfo()` snapshot narrows an R version mismatch, but it cannot regenerate the text. Keep the prompts, and the transcripts where the interface allows, as files in the compendium.

The compendium is where this provenance belongs, as it is for every other input. Commit a file `LLM-USE.md`, or a section of the README, with one row per use: the date, the tool and its identifier, the purpose, the affected

Draft, November 13, 2026. Content may change.

files, the path to the prompt file, and the verification performed. A commit that an agent made should say so, in the commit message or in a trailer line used consistently, such as `Assisted-by: <tool name>`, so that the history shows which changes a person wrote.

Funders set their own rules. NIH will not treat an application substantially developed by AI as the applicant's original idea (National Institutes of Health, 2025b), and it prohibits its peer reviewers from using generative AI to analyze applications or write critiques (National Institutes of Health, 2023).

2.11. Exercises

1. Locate a published article in your own field that provides code and data. Attempt to run the code on your machine without modification. Record what happened at each step, and classify any failure you meet as computational or statistical. On which rung of the ladder (L0 to L3) would you place the deposited analysis, and why?

2. The chapter distinguishes reproducibility, replicability, and robustness. For a study you know well, write one sentence describing what it would mean for that study to fail on each of the three dimensions separately. Which failure would be most damaging to the study's clinical conclusion?

3. Modify the worked-example chunk to draw a bootstrap resample of a small synthetic vector rather than a shuffle, again comparing the two RNG kinds under a fixed seed. Does the choice of sampler still change the result? Explain what this implies for a bootstrap confidence interval computed under the two R versions.

4. Consider the archetypes named in the chapter (a data-analysis project, a manuscript, an R package, a simulation study, and a blog). For each, state which single determinant you would expect to be most load-bearing for its reproducibility, and justify the choice in one sentence.

5. Reproducibility is described as a floor rather than a ceiling. Construct a short, concrete example of an analysis that is fully reproducible (L3)

Draft, November 13, 2026. Content may change.

and yet scientifically wrong, and explain what a reader would need beyond reproducibility to detect the error.

6. Read the abstract of the Open Science Collaboration report (Open Science Collaboration, 2015) and of the Trisovic study (Trisovic et al., 2022). In two or three sentences, explain why the two papers measure different things, and which one is the more relevant evidence for the argument of this book.

7. Run the two loading calls in 'The cohort, continued' above on `analysis/data/raw_data/cohort_raw.csv`. Report two values a grading script could check without fitting any model: the class `load_cohort()` gives `raw$sex`, and the class the same file gives `legacy$sex` when read with `stringsAsFactors = TRUE`. State in one sentence which determinant-stack layer the difference between the two calls demonstrates.

8. This exercise returns to the informal, first-pass account of silently shifting R defaults given in Chapter 1 (Chapter 1). That chapter names the determinant stack informally and mentions the `stringsAsFactors` flip, but does not say which layer of the stack the flip occupies. Using this chapter's own vocabulary, state which layer that change occupies, and explain in one sentence why naming the layer, rather than merely noting that a default changed, is the addition this chapter makes.

2.12. Solutions

Attempt an exercise before expanding its solution. Several exercises in this chapter ask you to work with an article or a study of your own choosing, so they have no single correct answer. For those the box says what a good answer establishes, and names the failure modes worth looking for.

Exercise 1

No single correct answer, since it depends on the article chosen. What a good answer establishes is a classification and a placement, both defended.

Draft, November 13, 2026. Content may change.

On the classification, the discipline is to ask what kind of thing went wrong. A missing package, a package that installs but at a newer version, a hard-coded path, a file that was never deposited, a script that assumes a working directory: these are computational failures. They are the ones this book addresses. A number that regenerates exactly but that you believe was the wrong number to compute is a statistical disagreement, and no amount of environment capture touches it. You will most likely find that your failures are overwhelmingly of the first kind, which is the point of the exercise and matches what Trisovic and colleagues measured at scale (Trisovic et al., 2022).
On the placement, the honest default for a deposited analysis is L0. Code and data that can be located and downloaded are locatable, and that is what most deposits supply. A deposit that also carries a lockfile is L1, and one carrying a container recipe is L2. A deposit carrying evidence that the outputs were regenerated from a clean rebuild and matched recorded values within a stated tolerance is L3. The rung is set by what the authors deposited, not by whether the code happened to run on your machine. An analysis can be L0 and still run, because you were lucky. It can also be L1 and still fail, because the lockfile pinned packages and something below them moved.

Exercise 2

No single correct answer, since it depends on the study. What a good answer establishes is that the three dimensions are independent, which is easiest to see by writing three sentences that could each be true while the other two are false.
A serviceable template. *Reproducibility fails* if a reader with the authors' own data and code cannot regenerate the published numbers. *Replicability fails* if a new cohort, collecting its own data to answer the same question, does not find the effect. *Robustness fails* if the effect disappears when a competent analyst makes a different but defensible modeling choice: a different adjustment set, a different handling of missing data, a different cut point.
On which failure is most damaging to the clinical conclusion, the defensible answer is replication. A reproducibility failure impugns the record but leaves open whether the finding is true. A robustness

Draft, November 13, 2026. Content may change.

failure shows the finding depends on a choice, which weakens it without refuting it. A replication failure is direct evidence against the effect. If you argue instead that reproducibility is most damaging, on the grounds that it undermines the ability to evaluate the other two at all, you have made a reasonable case and should say so explicitly.

Exercise 3

Yes, the choice of sampler still changes the result, and for the same reason as in the worked example. Both `sample(1:10)` and a bootstrap resample, `sample(x, replace = TRUE)`, obtain their randomness by drawing from the same underlying generator, so a change of `sample.kind` changes the index sequence and therefore the resample. Substituting `sample(x, length(x), replace = TRUE)` for the shuffle in `draw_sample()` and calling it under `Rounding` and `Rejection` at seed 123 produces two different resamples.

The implication for a bootstrap confidence interval is the substantive part of the exercise. The interval is a function of the resamples, so two R versions straddling 3.6.0 produce different intervals from the same seed, the same data, and the same code. The difference is small when the number of replicates is large, since both intervals estimate the same quantity and the Monte Carlo error shrinks. It is not zero, however, and it is enough to move a reported bound in its last printed digits. If a published interval is quoted as excluding the null by a narrow margin, a rerun under the other sampler can move it across that boundary.

A seed is therefore not a sufficient record of a stochastic analysis. The seed pins the starting point, but the RNG kind determines what the generator does with it. A lockfile does not fix the kind at all. A container that pins the R release fixes the *default* kind, but an override set in code or in a project `.Rprofile` is session configuration, which only an explicit `RNGkind()` call in the analysis fixes. An analysis that depends on a bootstrap should record `RNGkind()` alongside the seed, or set it explicitly.

Draft, November 13, 2026. Content may change.

Exercise 4

Answers will vary in their justification, and more than one choice is defensible for each archetype. What a good answer establishes is that the archetypes stress different layers, so a single reproducibility recipe serves them unevenly.

A defensible set. For a data-analysis project, the data itself, since the analysis is worthless without the extract it ran on and that extract is often the artifact least likely to travel. For a manuscript, the rendering toolchain, since the deliverable is a document and the numbers in it must be regenerated rather than retyped. For an R package, the declared dependency set, since the package is consumed by other code and its correctness is a claim about behavior under a stated environment. For a simulation study, the random-number discipline, since the entire output is a function of the seed and the generator, as this chapter's worked example shows. For a blog, the pinned package versions, since posts are written once and read for years while the packages beneath them move.

If you answered 'the data' for every archetype, you have not made an error so much as missed the exercise, which is to notice that the binding constraint moves.

Exercise 5

Answers will vary. What a good answer establishes is that reproducibility is a property of re-execution and carries no claim about correctness.

A clean construction: a cohort analysis computes thirty-day readmission but the cohort was defined with an off-by-one error in the date window, so patients readmitted on day thirty are excluded. The analysis is containerized, its packages pinned, its outputs checked by a baseline test against recorded values, and it regenerates the same number, within tolerance, on every machine and every rerun. It is L3. It is also wrong, and it will remain wrong, identically, for as long as anyone runs it. Other serviceable examples are a sign error in a contrast, a covariate adjusted for that sits on the causal pathway, and a survival analysis that treats censoring as an event.

What a reader needs beyond reproducibility to detect the error is subject-matter review of the code and the design. That might be someone who knows what thirty-day readmission means reading the cohort definition, or a unit test asserting the boundary case against a hand-computed answer. This is the point at which Chapter 11 takes over from environment capture, and it is why the book treats validation as a separate register from capture. Note the uncomfortable corollary: reproducibility makes an error easier to find once someone suspects it, and does nothing to make anyone suspect it.

Exercise 6

The two papers measure different things because they address two of the three words this chapter separates.

The Open Science Collaboration ran new studies collecting new data, and asked whether the original findings held (Open Science Collaboration, 2015). That is *replication*: a statement about the durability of the science. Trisovic and colleagues took deposited code and data and asked whether it would execute at all (Trisovic et al., 2022). That is *reproducibility* in the narrow sense: a statement about re-execution, in which the science is not in question because it never got as far as being evaluated.

For the argument of this book, Trisovic is the more relevant evidence, even though the replication literature is the more famous. This book is about the computational floor. It has nothing to say about whether an effect is real. Its whole claim is that a large fraction of deposited analyses cannot be re-executed, and that is a problem soluble by engineering rather than by more studies. The Open Science Collaboration measures a problem this book does not address and cannot fix.

The stronger version of the answer notes that the two are not independent in practice. A field that cannot re-execute its own analyses cannot easily diagnose why a replication failed, because it cannot establish what the original analysis actually did.

Draft, November 13, 2026. Content may change.

Exercise 7

The solution for this exercise is in the instructor guide, which is distributed separately to course instructors. Self-study readers who want to check their answer can compare the two autogradable values against those reported in the chapter's own 'The cohort, continued' section.

Exercise 8

Chapter 1 mentions, in passing, 'a default that shifted between releases of R' and later names the specific instance, the `stringsAsFactors` flip. It also names the stack informally, the language runtime among its layers, but it does not say which layer the flip occupies. This chapter's own stack figure (Figure 2.1) and its 'Why the failures are computational, not statistical' section supply the name: `stringsAsFactors` is a default of the **R runtime**. The runtime *default* is secured only at L2, the same rung that secures the operating system and the system libraries, and not by a lockfile at L1. The exposure itself can be closed earlier, in code, as `load_cohort()` does in 'The cohort, continued'. Naming the layer converts a story into a diagnosis with a remedy attached. 'A default changed' tells a reader only that something went wrong somewhere. 'The R-runtime layer was unpinned' tells the reader exactly which artifact, a container image pinning the R version, would have prevented the failure, and at which rung of the ladder it would have been prevented. This chapter's vocabulary makes the informal account of Chapter 1 precise without contradicting it.

Answers to the checks

Check 1. *from earlier chapters* (page 36)

'Replicate' is the wrong word for the failed re-run of the 2019 cohort script. In the National Academies' terms that Chapter 1 adopts, re-running the same code on the same data tests reproducibility; replicability asks whether a finding holds in a new study with its own data. The character column is the R 4.0.0 change of the `stringsAsFactors` default from `TRUE` to `FALSE`.

Draft, November 13, 2026. Content may change.

The fault therefore lies in the language runtime, R itself, and not in the statistics. A lockfile from `renv` would not have closed it, because `renv` pins package versions and this change is in R. A container built from a Rocker image would have, since it pins the operating system and the runtime, the rung the book calls L2, pinned environment. Continuous integration on a clean machine would then confirm that the pinned layers suffice.

Check 2. *naming the rung* (page 38)

At most L0, locatable. The script could be found and read, which is all L0 requires; nothing about it suggests that the exact package versions, the operating system, or the runtime had been pinned. That the script ran at all, once a missing package was located, is evidence of nothing beyond locatability. A script can be found and still fail to regenerate its own number, as this one did.

Check 3. *whose judgment* (page 41)

The third, deciding what must be pinned. Reaching L2 pins the operating system, the runtime, and the system libraries, but it does not by itself decide whether the random-number configuration is a live exposure for this particular analysis. A bootstrap or a simulation makes it one; a deterministic calculation does not. The reviewer is asking the authors to have made, and stated, exactly that judgment, which no tool makes for them.

Check 4. *two kinds of failure* (page 42)

The colleague has demonstrated, at most, a failure of computational reproducibility. That is, the same code and data did not regenerate the same number, most likely because the software environment shifted between the original run and this one. That is a statement about re-execution, not about whether the scientific finding holds in a new sample, which is what replication means. The situation is precisely the one that Trisovic and colleagues (Trisovic et al., 2022) measured at scale. Until the environment is reconstructed and the code is shown either to regenerate the original number or to contain a genuine error, no claim about replication is warranted.

Check 5. *three words, one study* (page 46)

The second registry's independent confirmation, with new data aimed at the same question, is replicability. That the confirmation held under a different

Draft, November 13, 2026. Content may change.

covariate adjustment is also evidence of robustness, since the conclusion survived a defensible change in analytical choice. The third team's failure to regenerate the deposited table, same data and same code required, is a reproducibility failure. It is independent of the other two, since the first registry's finding could be both replicable and robust while its own code still fails to re-execute.

Check 6. *locating the failure* (page 48)

The most likely culprit is the numerical-library layer, specifically the BLAS and LAPACK implementation that performs the singular value decomposition underlying `prcomp()`. Different implementations (reference BLAS, OpenBLAS, Intel MKL, Apple's Accelerate) return results that are numerically equivalent but not bit-identical, and a sign flip or reordering of components can make the difference visible in a plot. A lockfile pins the R packages but says nothing about the numerical libraries the packages link against, which live below the package layer. A container image (Chapter 7) determines which BLAS is installed, but no rung of the ladder requires that library to be fixed, and the thread count and the processor still vary between machines. No rung therefore guarantees this layer, as Chapter 3 explains. Setting a seed does not help because the difference is not stochastic.

Check 7. *floor, not ceiling* (page 49)

L3 shows only that the pinned definition regenerates its own recorded output within a stated tolerance; it says nothing about whether that output was the right thing to compute. Leek and Peng's argument, that reproducibility guarantees regeneration and not correctness (Leek & Peng, 2015), refutes the inference directly. A script that faithfully regenerates a mistaken p-value at L3 is perfectly reproducible and perfectly wrong.

Check 8. *the seed was never enough* (page 51)

The RNG kind, set by `RNGkind()`. Its default comes with the R release, and an override set in code is part of the session configuration; neither is part of the seed. The seed fixes the starting point; the kind fixes what the generator does with it. Recording a seed without the kind is recording half of a determinant.

Check 9. *which layer is closed* (page 52)

Draft, November 13, 2026. Content may change.

No. The R-runtime layer is pinned only by fixing which R release runs the code, which is an L2 concern for Chapter 7 to close. What `load_cohort()` pins is the analysis-code layer: the function states its own choice of `stringsAsFactors` explicitly, so the column's class no longer depends on which runtime default happens to be in force. It is a narrow, code-level fix for one instance of exactly the hazard this chapter describes, not a substitute for pinning the runtime itself.

2.13. Cheat sheet

Term or rule	What it means
Reproducibility	Same code, same data, re-run; a statement about re-execution (National Academies of Sciences, Engineering, and Medicine, 2019)
Replicability	A new study, new data, same question; a statement about a finding's durability
Robustness	Same data, a defensible analytical change, same conclusion
Computational failure	Code is sound; its environment (a package, an R default) has shifted
Statistical failure	The method or inference was wrong; the environment is not the issue
`RNGkind()`	Sets the generator algorithm; two runs with the same seed but a different kind draw differently
`stringsAsFactors`	R default, `TRUE` before R 4.0.0, `FALSE` from R 4.0.0 on (R Core Team, 2020)
Determinant stack	What a result depends on beneath the code and data; developed fully in Chapter 3
Floor, not ceiling	A reproducible result can still be wrong; never claim correctness from re-execution alone

Draft, November 13, 2026. Content may change.

Term or rule	What it means
Honest level	State the rung actually reached, never one asserted above it
Record model use (Section 2.10.2)	One `LLM-USE.md` row per use: date, tool and identifier, purpose, files, prompt path, verification

2.14. Further reading

- The National Academies consensus report (National Academies of Sciences, Engineering, and Medicine, 2019) is the authoritative source for the distinction between reproducibility and replicability, and repays reading in full for the care with which it separates them.
- Goodman and colleagues (Goodman et al., 2016) document the terminological confusion directly and propose a cleaner lexicon; read it to understand why the same words mean opposite things across fields.
- Peng's short article (Peng, 2011), together with his earlier treatment (Peng et al., 2006), is the origin of the spectrum idea that becomes the levels ladder of Chapter 3, and states the case for reproducibility as a minimum standard.
- For the empirical scale of the problem, the Open Science Collaboration report (Open Science Collaboration, 2015) and Begley and Ellis (Begley & Ellis, 2012) establish the replication failures, while Trisovic and colleagues (Trisovic et al., 2022) and Saju and colleagues (Saju et al., 2025) isolate the computational ones. Baker's Nature survey (Baker, 2016) captures how researchers themselves perceive the crisis, and Ioannidis (Ioannidis, 2005) supplies the theoretical argument that anticipated it.
- On the load-bearing caveat that reproducibility is a floor and not a ceiling, Leek and Peng (Leek & Peng, 2015) are brief, pointed, and worth reading before one is tempted to equate a reproducible result with a correct one.

Draft, November 13, 2026. Content may change.

- For the historical origin of the ideal, Claerbout's work in reproducible geophysics (Claerbout & Karrenbach, 1992) is the standard reference for where the modern practice begins.

Draft, November 13, 2026. Content may change.

3. Levels of Reproducibility

> *Reproducibility has the potential to serve as a minimum standard for judging scientific claims when full independent replication of a study is not possible.*
>
> Roger D. Peng, *Reproducible Research in Computational Science* (2011)

By the end of this chapter you will be able to look at any repository, your own or someone else's, and say exactly which of four rungs it has reached. You will also be able to say what still stands between it and the next one, rather than reach for the unfalsifiable claim 'fully reproducible.'

Before you start

Reading time. About 40 minutes.
Prerequisites. Chapter 1 (the introduction), for the first, informal pass at the determinant stack this chapter makes precise, and Chapter 2 (Chapter 2), for the two silent defaults of R itself; no other chapter is assumed. Statistically, nothing beyond reading a regression output, a hazard ratio or an odds ratio with its confidence interval, is required. No R fluency beyond reading a short script is assumed; the chapter's own code chunks are simple enough to follow without having written R before.
By the end of this chapter you will be able to answer:

- Can you enumerate the layers of the determinant stack on which a computational result stands, and say which layers common practice pins and which it leaves to chance?
- Can you place an analysis on the four-rung ladder of this book (L0 locatable, L1 pinned packages, L2 pinned environment, L3 verified) and name the artifact that marks each transition?
- Can you distinguish capturing reproducibility from validating it, and explain why continuous integration adds no new pinned

layer?
- Can you distinguish a result that is reproduced from one that is replicated, and navigate the shifting use of those two words across the literature?
- Can you state plainly the level an analysis has reached, and no higher, including the candid admission that no present framework pins the whole stack?

Outline.

- Orientation: a result that silently changed on a second machine.
- The analyst's contribution: the three judgments no tool automates.
- The stack a result stands on: the nine layers, and which rungs secure which.
- The ladder: capture sets the level, validation confirms it.
- Reproduced, replicated, and a warning about the words.
- Two numbering schemes, and which this book uses.
- The ladder grades every archetype.
- No framework pins the whole stack.
- Worked example: one analysis walked up all four rungs.
- The cohort, continued: placing this book's own compendium on the ladder, today.

Check 1: from earlier chapters

A colleague re-runs a 2018 heart-failure analysis, written under R 3.5, on a laptop with a current release of R. The script calls `set.seed(42)` before a bootstrap built on `sample()`, yet the bootstrap interval has moved. In the same run, `levels(d$arm)` on a column read by `read.csv()` returns `NULL`. Drawing on Chapter 1 and Chapter 2, is this a replication failure? For each discrepancy, name the R release and the determinant layer responsible, and say what would close it.
The answer is on page 100.

Draft, November 13, 2026. Content may change.

3.1. Orientation

Consider a registry study of thirty-day readmission after heart failure. The analyst fits a mixed-effects model, draws a forest plot, and reports an adjusted odds ratio of 1.24. The script runs, the manuscript renders, the paper is accepted. Two years later a reader requests the code, downloads it, and runs it on a fresh laptop. The model fits, but differently. A package that the analysis relied upon has changed its default optimizer, and the estimate now returned is 1.19. Which number is the finding? The discrepancy is computational in origin. The model specified was the same on both machines; what differed was everything underneath it, and the analyst never recorded it. Its size, however, is a statistical finding. An estimate that moves from 1.24 to 1.19 when only the optimizer changes signals a flat or ill-conditioned likelihood, or a fit that has not converged. It should be checked, for example by refitting under several optimizers with `lme4::allFit()`, and reported as optimizer-sensitive.

This chapter is about naming that 'everything underneath', and about grading how much of it a given analysis has secured. We introduced in Chapter 1 the observation that a computational result depends on a stack of determinants beneath the code and the data, and that a published script pins almost none of them. Here we make that observation precise. After setting out the judgments that remain with the analyst, we shall lay out the stack layer by layer. We then define the ladder that this book uses to say how far down the stack a particular effort has reached: L0 locatable, L1 pinned packages, L2 pinned environment, and L3 verified. With the ladder in hand, we consider how the words 'reproduced' and 'replicated' have shifted across the literature, and how the framework's own numbering differs from ours. Lastly, we examine how the ladder grades each archetype, admit that no framework pins the whole stack, walk one analysis up all four rungs, and then place this book's own compendium on the ladder.

The ladder is the conceptual backbone of everything that follows. Every later chapter is, in effect, an account of how to climb one rung of it. Pinning packages with a lockfile (Chapter 6) is the L0-to-L1 step. Pinning the environment with a container (Chapter 7) is the L1-to-L2 step. And verifying the outputs (Chapter 12) is the L2-to-L3 step. Before we can teach the climbing we need the ladder itself, together with the discipline of naming, without flinching, the rung one has actually reached.

Draft, November 13, 2026. Content may change.

Check 2: what changed, and what did not

In the registry-study story above, the code and the data were unchanged between the two runs. What changed, why does calling the discrepancy purely 'statistical' mischaracterize its origin, and what statistical question does it nonetheless raise?
The answer is on page 101.

3.2. The analyst's contribution

We begin with three judgments at the center of this chapter that belong to the analyst and are automated by no tool. The machinery of later chapters exists to support them.

1. **Choosing, and declaring, the honest level.** A framework can compute which rung a repository has reached and print it; it cannot decide what the author will claim. The judgment that a paper's reproducibility statement will match the level actually secured, and never assert above it, is the analyst's alone. It is the single habit this book most wishes to instill.

2. **Matching the investment to the stakes.** Climbing the ladder costs effort, and not every analysis warrants the climb. A two-day exploratory calculation may honestly rest at L0; a registry study destined for a guideline warrants L2 or L3. Deciding where on the ladder a given project should sit, given its lifespan, its audience, and the cost of being wrong, is a design decision.

3. **Naming what remains exposed.** No rung of the ladder pins the whole stack. The analyst who reaches L2 must still ask whether the layers left unpinned (the numerical libraries, the random-number discipline, the integrity of an external data source) matter for this particular analysis. Those that do must be closed by hand. Telling a live exposure from an inert one is a matter of subject-matter judgment.

Draft, November 13, 2026. Content may change.

Check 3: whose judgment is it?

A vendor's reproducibility dashboard reports that a project has reached 'L2'. Which of the three judgments above, if any, has the dashboard made for the analyst, and which remain the analyst's alone?
The answer is on page 101.

The vocabulary of this chapter

New terms are defined again on first use; they are collected here for reference.

- **Determinant stack.** The layered set of things a computational result depends upon, from the hardware at the base through the environment and the packages to the analysis code and the data at the top. A layer is 'pinned' when it has been fixed to a recorded value.
- **Capture.** Fixing an input so that it cannot drift: writing a lockfile, building a container image. Capture sets levels L0 to L2; L3 is reached by validation.
- **Validation.** Confirming that the captured inputs actually suffice, by rebuilding and re-running. Validation does not pin anything new; it checks.
- **The level ladder (L0 to L3).** The four-rung scale, L0 locatable, L1 pinned packages, L2 pinned environment, L3 verified, used throughout this book to grade how far down the stack an analysis has been secured.
- **Reproduced.** Obtaining the same computational result by re-executing the same code on the same data. Contrast **replicated**, obtaining a consistent finding from a new study with new data.
- **Lockfile.** A file, such as `renv.lock`, recording the exact version of every package an analysis used.
- **Container image.** A packaged environment, built from a recipe, that carries the operating system, the system libraries, and the language runtime, and provides the same environment wherever a container runtime for its platform runs.
- **RNG kind.** The random-number algorithm in force, set by `RNGkind()`. Two runs with the same seed but a different kind

Draft, November 13, 2026. Content may change.

draw different numbers.

3.3. The stack a result stands on

We turn now to the determinants, because the ladder is only intelligible once we see what it grades. A computational result is the output of running some code against some data. However, the code and the data are only the two topmost of a tall stack of dependencies. The result is a function of the whole stack, not of its top two layers alone. Arranging the determinants from the hardware upward to the analysis logic yields the nine layers this book uses; a finer split, such as one that separates the document toolchain, yields ten or more.

At the base sits the **hardware**: the processor architecture, its instruction-set extensions, and the behavior of its floating-point unit. Above it lie the **numerical libraries**, the BLAS and LAPACK implementations that perform the linear algebra beneath every regression and every matrix decomposition. These matter more than their obscurity suggests. A different BLAS, or the same BLAS run with a different thread count, reorders the floating-point summations and can change a result in its low-order digits. Above these sit the **operating system and its system libraries**: the kernel, the C library, the locale data, the fonts, and the document toolchain of `pandoc` and LaTeX. Then comes the **R runtime** itself, whose default behaviors shift between releases. The random-number change of R 3.6.0 and the `stringsAsFactors` change of R 4.0.0 both illustrate the point.

Above the runtime sit the layers most analysts think of first. The **package versions**, the exact release of every R package the analysis loads, are the layer that churns fastest and breaks analyses most often. The **session configuration** is a cluster of settings that shape the session before any user code runs. It comprises the random-number kind set by `RNGkind()`, the contrasts used for factor variables, and the number of printed `digits`. It also comprises the locale that governs sorting, and the time zone that governs date arithmetic. Then come the two layers the analyst actually writes. These are the **analysis code**, including every `set.seed()` call and the order of execution, and the **input data**, its content and its integrity. Finally, threaded through the whole stack, sits **intrinsic non-determinism**: the

Draft, November 13, 2026. Content may change.

use of the wall-clock, race conditions in parallel code, and iteration over unordered structures are each capable of producing a different answer on two runs of identical inputs.

Common practice pins very few of these layers. A bare published script pins exactly one layer with any reliability, the analysis code, since the code is the file itself. If the reader is fortunate, it pins a second, the data, when it has been deposited alongside. Every other layer is left to whatever the next machine happens to provide. Figure 3.1 maps the stack against the rung of the ladder that first secures each layer. Three layers (the hardware, the numerical libraries, and intrinsic non-determinism) are secured by no rung of the capture ladder at all. The numerical libraries are the case most often misread. A container image determines which BLAS is installed, but the ladder does not require that library to be fixed, and the thread count and the processor still vary between machines. No rung therefore guarantees this layer.

```
plot_determinant_stack(
  captures = determinant_layers()$layer,
  title = 'Which rung first secures each layer')
```

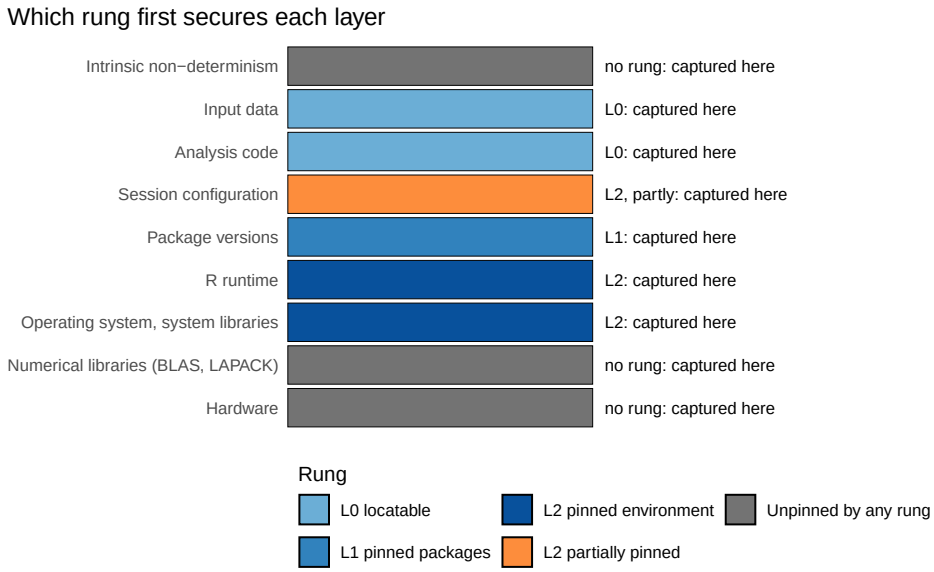


Figure 3.1.: The determinant stack, from the hardware at the base to the data at the top, colored by the rung of the ladder that first secures each layer. Package versions are secured at L1, the operating system and R runtime at L2, and the analysis code and data are merely locatable at L0. The session-configuration layer is only partially pinned at L2: the locale and the time zone are fixed, but the RNG kind, the contrasts, and the number of printed digits are not. Three further layers, the hardware, the numerical libraries, and intrinsic non-determinism, are pinned by no rung of the capture ladder. Every chapter that draws this stack draws it with the same function, as the Conventions page explains, so that no two chapters can disagree about what a layer is called or which rung secures it.

Two settings of the session-configuration layer deserve a word, because they are the settings most often forgotten by an analyst who has already adopted a lockfile and a container. The random-number kind is the more dangerous. An analysis that seeds its generator and draws a sample is reproducible only if the algorithm behind the seed is also fixed. The change of the default `sample.kind` in R 3.6.0 broke exactly this, silently, for every analysis that

Draft, November 13, 2026. Content may change.

had set a seed under the old default: the same seed now drew different samples, so no seeded result regenerated digit for digit. Whether a result also changed as a statistical matter is a separate question. Chapter 11 answers it on the book's cohort (Table 11.2): the shift is a fraction of one Monte Carlo standard error, so the bound reproduces statistically although not seed-exactly. Given that simulation studies, multiple imputation, and bootstrap confidence intervals are staples of health research, the RNG kind is a first-order concern. The locale is the second. `sort()` and `order()` on a character vector depend on `LC_COLLATE`, so an analysis developed under one locale and re-run under another can reorder a table without any error. The decimal mark is a different matter. R keeps `LC_NUMERIC` at 'C' and warns if code changes it, so base parsing reads '.' as the decimal mark whatever the system locale. A file written with decimal commas is read as text unless the reading function is told otherwise, for example with `dec = ','` in `read.csv()`.

The RNG kind also shows why a setting can belong to two layers at once. The R release sets the *default* kind, so the default is part of the R runtime and is fixed at L2. Code or a project `.Rprofile` can *override* the default, and an override is session configuration, fixed only by an explicit `RNGkind()` call in the analysis code. The same distinction between a default and an override recurs across the stack, and the table below states it for the settings this book returns to most often.

Setting	What fixes the default	What can override it	What closes the override
RNG kind (`sample.kind`)	The R release (R runtime, L2)	Code, `.Rprofile`, a package	An explicit `RNGkind()` call in the analysis
`stringsAsFactors`	The R release (R runtime, L2)	The argument in each call	Writing the argument in the code
Locale and time zone	The image or the host (L2 when the image sets them)	`Sys.setlocale()`, `Sys.setenv(TZ = )`	Setting them in the image or in the analysis

Draft, November 13, 2026. Content may change.

Setting	What fixes the default	What can override it	What closes the override
BLAS and LAPACK	The image or the host (no rung)	The library path, the thread count	No rung; fixing the thread count narrows it, and a stated tolerance absorbs what remains

Check 4: locating a failure on the stack

An epidemiologist reports that a colleague, re-running her cohort analysis, obtained a least-squares regression coefficient identical to five decimal places but a different value in the sixth. The fit was computed as `solve(crossprod(X), crossprod(X, y))` on a large design matrix. The code, the data, the package versions, and the operating system were all identical. Which layer of the determinant stack is the most likely culprit, and is the discrepancy a bug?
The answer is on page 101.

3.4. The ladder: capture sets the level, validation confirms it

With the stack in view we can define the ladder. The central design idea, which we take from the framework this book uses and develop as a general principle, is that the features acting on reproducibility divide into two kinds that are constantly confused. Separating them is what makes an honest grade possible.

The **capture axes** fix inputs, and therefore set the level. There are two of them: the package backend, which pins the package layer, and the computational environment, which pins the operating system, the runtime, and the system libraries. Adding a capture feature raises the level; removing one lowers it. The **validation layer** confirms the captured level without capturing anything new. Its members are continuous integration,

Draft, November 13, 2026. Content may change.

the dependency-consistency check, and the output-verification command. Removing a validation feature leaves a repository exactly as reconstructable as before; it only stops the automated checking. This is why continuous integration, which we develop in Chapter 12, is best understood as the harness that runs the verification rather than as a layer of reproducibility in its own right. Pinning is capture; re-running to confirm is validation.

The four rungs follow from this distinction. Levels L0 through L2 are reached by the capture axes, and L3 is reached by the validation layer. Table 3.1 states each rung, what it pins, the artifact that marks it, and what the rung means for a reader of health research.

```
library(tibble)
library(knitr)

ladder <- tibble::tibble(
  Level = c('L0 locatable',
            'L1 pinned packages',
            'L2 pinned environment',
            'L3 verified'),
  `What is pinned` = c(
    'Nothing computational; the source is under version
     ↪ control',
    'The exact version of every R package',
    'The operating system, the R runtime, and the system
     ↪ libraries as well',
    'Nothing new; the recorded outputs are shown to
     ↪ regenerate within a stated tolerance from a clean
     ↪ rebuild'),
  `Primary artifact` = c(
    'A version-controlled repository',
    'A lockfile such as renv.lock',
    'A container image or a Nix environment',
    'A passing baseline test, run after a clean rebuild,
     ↪ that compares regenerated outputs with values
     ↪ recorded in a committed baseline file within a
     ↪ stated tolerance'),
  `Health-research reading` = c(
    'A reviewer can find the code but cannot yet rebuild
     ↪ it',
```

```
    'A colleague installs the same packages the analysis
     ↪  used',
    'The analysis yields the same numbers on a different
     ↪  machine',
    'The reported table has been shown to regenerate, not
     ↪  merely asserted to'))

knitr::kable(ladder)
```

Table 3.1.: The reproducibility ladder used throughout this book. Each rung names what it pins, the primary artifact that marks it, and what reaching it means for a reader of health research.

Level	What is pinned	Primary artifact	Health-research reading
L0 locatable	Nothing computational; the source is under version control	A version-controlled repository	A reviewer can find the code but cannot yet rebuild it
L1 pinned packages	The exact version of every R package	A lockfile such as renv.lock	A colleague installs the same packages the analysis used
L2 pinned environment	The operating system, the R runtime, and the system libraries as well	A container image or a Nix environment	The analysis yields the same numbers on a different machine

Draft, November 13, 2026. Content may change.

Table 3.1.: The reproducibility ladder used throughout this book. Each rung names what it pins, the primary artifact that marks it, and what reaching it means for a reader of health research.

Level	What is pinned	Primary artifact	Health-research reading
L3 verified	Nothing new; the recorded outputs are shown to regenerate within a stated tolerance from a clean rebuild	A passing baseline test, run after a clean rebuild, that compares regenerated outputs with values recorded in a committed baseline file within a stated tolerance	The reported table has been shown to regenerate, not merely asserted to

L3 is the one rung that cannot be read off a file listing. An analysis is at **L3, verified**, when its recorded outputs have been shown to regenerate within a stated tolerance from a clean rebuild. The showing is done by a *baseline test*, that is, a test that recomputes the reported quantities and compares them with values recorded in a committed baseline file. The comparison is made within a tolerance written into the test. A clean rebuild whose tests pass, but which compares nothing against a recorded baseline, demonstrates that the environment builds and the code runs. That is L2 confirmed, not L3. But why a tolerance, and not exact equality? Estimates are floating-point numbers, and exact equality would fail on harmless differences in the last digits while saying nothing about whether the substantive result moved. In the framework this book uses, `zzc verify --full` applies this definition. It records L3 only when the image rebuilds, the test suite passes, and at least one baseline test (a file named `inst/tinytest/test-baseline*.R`) exists and appears in the test log. Otherwise it reports the project as rebuilt and tested, L2, with no baseline comparison. Continuous integration that re-runs the check on every change keeps an L3 claim current. However, it is the baseline comparison, not the automation, that earns the rung.

The rungs are cumulative. An L2 analysis is also locatable and package-pinned; the label names the highest rung reached. The transition that matters most for a first-year student to internalize is the one between L1 and L2. A lockfile alone, the L1 artifact, pins the packages but not the R

Draft, November 13, 2026. Content may change.

version, the operating system, or the system libraries a package links against. A lockfile restored under a different R version may fail to install or, more insidiously, may install and behave differently. The container image, the L2 artifact, fixes what the lockfile cannot. The two are complements, not substitutes, and a genuinely pinned-environment analysis carries both.

The ladder consolidates several graded views of reproducibility that the literature has offered in different vocabularies. Peng (2011) arranged computational work along a spectrum, from the merely published, through code available, code and data available, and linked and executable, to full replication. Our L0, L1, and L2 track the lower and middle reaches of that spectrum. The artifact-review badges used by the ACM and several computing venues express a related ordering, from artifacts that are available, through artifacts evaluated as functional and then reusable, to results independently reproduced (Association for Computing Machinery, 2020). The badges and the rungs grade different things and should not be equated. The 'functional' and 'reusable' badges grade how well an artifact is documented and exercised. They do not grade the package-versus-environment pinning depth that separates L1 from L2. Likewise, reaching L3 is not the same act as earning a venue's reproduction badge, which is awarded by reviewers under that venue's own policy. The reader who needs a badge should read the venue's current definition directly. The L-scheme adds no new definition of reproducibility. It maps these graded views onto artifacts a tool can detect, so that the level of a repository can be determined mechanically and reported honestly, rather than asserted by its author.

Check 5: capture or validation?

A laboratory adds a continuous-integration workflow that rebuilds its compendium and runs the tests on every push. A colleague objects that this cannot improve reproducibility, because it pins no new layer of the determinant stack. Is the colleague correct, and has the laboratory wasted its effort?

The answer is on page 102.

Draft, November 13, 2026. Content may change.

3.5. Reproduced, replicated, and a warning about the words

The ladder grades re-execution, and re-execution is only one of the things the word 'reproducibility' is made to carry. We need to fix the terms before reading the literature, because the same two words are attached to opposite meanings across the sources a reader will meet.

The National Academies, in their 2019 consensus report, adopt the distinction this book follows (National Academies of Sciences, Engineering, and Medicine, 2019). *Reproducibility* is obtaining consistent computational results using the same input data, the same code, and the same conditions of analysis. That is, it is a statement about re-execution, and it is what the ladder grades. *Replicability* is obtaining consistent results across studies aimed at the same question, each with its own data. It is a statement about whether a scientific finding holds on new data. The two are independent. A result can be perfectly reproducible and entirely wrong, if the code faithfully regenerates a mistaken number every time. Conversely, a finding can replicate across laboratories while no single analysis in the set is computationally reproducible. Leek and Peng press exactly this point, that a reproducible analysis can still be wrong, and that reproducibility is therefore a floor and not a ceiling (Leek & Peng, 2015). Patil, Peng, and Leek go further and offer a statistical framing of the distinction. They cast replication as a question about the agreement of estimates across studies rather than about the re-running of code (Patil et al., 2016).

How do other vocabularies relate to this one? The current ACM artifact-badge vocabulary, in its 2020 revision, aligns its direction of use with the National Academies convention rather than inverting it (Association for Computing Machinery, 2020). However, the two draw the boundary differently. Under the National Academies convention, 'Reproduced' means obtaining a prior result using the same code and data; the same team re-running their own analysis qualifies. Under the ACM 'Results Reproduced' badge, the re-running must be done by a team other than the original authors, using the author's provided artifacts. Both conventions therefore point 'Reproduced' toward re-execution with the original artifacts and 'Replicated' toward independence. The ACM badge, however, specifically

Draft, November 13, 2026. Content may change.

requires a different team, while the National Academies definition does not.

The genuine word-swap that trips students belongs to two other places. One is the pre-2020 ACM badging, which attached the two words in the opposite direction along the re-execution axis. The other is a set of biomedical and other field-specific usages, which a reader may still encounter, where 'reproduce' names an independent study rather than a re-run. A reader who carries the sense of the words from one of those older documents into a current one will be misled. The misreading happens silently and without any error to warn them, in exactly the manner of a shifted package default. This book therefore fixes the meaning and states it once. Throughout, 'reproduced' carries the National Academies sense. L3 means that the captured analysis, rebuilt from its pinned definition and re-run, regenerates its own recorded outputs within a stated tolerance, as checked by a baseline test. It is deliberately not replication in the sense of an independent study on new data, which no software framework can supply. When the reader meets 'reproduced' or 'replicated' in a journal's author instructions or a venue's badging policy, the first task is to determine which convention that document uses.

⚠ Another warning about the words: 'validation'

This chapter uses 'validation' and the L3 label 'verified' for a narrow act: confirming that a previously captured value still holds, as a baseline test does. Regulated industry work, pharmaceutical and other FDA-adjacent settings among them, attaches 'validation' to something larger: a formal, documented process confirming that a system is fit for its intended use, as under 21 CFR 11.10(a) (*21 CFR Part 11*, n.d.). European regulators expect something comparable of a sponsor's analysis tools (European Medicines Agency, GCP Inspectors Working Group, 2023). A graduate moving into that kind of work will meet the regulated sense on arrival; this book's narrower one does not prepare them for the paperwork that comes with it.

Draft, November 13, 2026. Content may change.

Check 6: which word, which sense?

A manuscript's cover letter states that the authors 'reproduced the primary finding in an independent cohort.' Under the National Academies convention this chapter follows, is that reproduction or replication, and why does the sentence as written risk being misread?
The answer is on page 102.

! The honest level

This chapter defines the ladder rather than climbing it. Its one executed placement, the cohort compendium in 'The cohort, continued' below, is computed from the repository's artifacts each time the chapter renders, not asserted here. Every rung above the one that section prints is left to the chapters this ladder points toward.

3.6. Two numbering schemes, and which this book uses

Before applying the ladder to particular archetypes, we need to dispose of a source of confusion. A reader who consults the documentation of the framework this book uses will encounter a second numbering scheme, and the two are easily confused.

The framework's teaching materials include a vignette that presents a *five-level* ramp. Level 1 is a basic R project with manual package management, and Level 2 is the addition of `renv`. Level 3 is the addition of Docker, Level 4 the addition of unit testing, and Level 5 the addition of continuous integration. That scale counts *which tools a project has adopted*, and it is a pedagogical on-ramp, meant to be walked from the bottom as a project matures. It is a useful thing, but it is not the scale of this book.

The scale of this book is the *architectural* ladder of the preceding sections, L0 through L3. It grades something different: *how far down the determinant stack a result has been locked, and whether the lock has been verified.* The two axes are related but not identical. Adopting `renv` (the tool scheme's Level 2) is what lifts a project to L1. Adopting a container (the tool

Draft, November 13, 2026. Content may change.

scheme's Level 3) is what lifts it to L2. The tool scheme's testing and continuous-integration levels belong to the validation layer that reaches L3. However, one can imagine a project that has adopted every tool and still verified nothing. The architectural ladder, not the tool count, is what tells the reader whether the result has been shown to reproduce.

To avoid ambiguity we adopt a single convention. This book uses the L0 to L3 architectural ladder throughout, in every chapter and every worked example. Where the framework's own documentation speaks of 'Level 1' through 'Level 5', it is using the tool-adoption ramp, and the reader should read it as such and not conflate it with our L0 to L3. When we write a bare 'L2', we always mean the architectural rung.

Check 7: two scales, one word

A student writes in a lab notebook, 'the project is at Level 2, so it is container-pinned.' Using this chapter's two scales, is the inference safe?
The answer is on page 102.

3.7. The ladder grades every archetype

Does the ladder apply equally to every kind of compendium? Table 5.2 in Chapter 5 shows how each archetype reads it. A data-analysis project and a manuscript read it the same way. An R package reaches L2 by `R CMD check` and L3 by a baseline on its function outputs, and a blog applies the ladder post by post. The case most easily missed is the simulation study. Its exposure on the session-configuration layer is acute, because its outputs are functions of the random-number stream. An unseeded or wrongly seeded simulation can therefore sit at L2, with a perfectly pinned environment, and still fail to reproduce. For this archetype the discipline of fixing the RNG kind and the seed is the difference between a nominal L2 and a real one. The ladder is a property of the determinant stack, which every archetype shares, and not of any particular directory layout.

Draft, November 13, 2026. Content may change.

Check 8: grading an R package

An R package has `renv.lock`, a Dockerfile, and a full `testthat`-style test suite that passes in a clean rebuild, but none of its tests compare a function's output with a value recorded earlier. What rung has it reached, and what is missing for the next one?
The answer is on page 102.

3.8. No framework pins the whole stack

Reaching L2 or L3 does not pin the entire determinant stack, and no present framework pins all of it.

By its own published accounting, which splits the stack more finely than this chapter into about ten determinants, the framework this book uses robustly pins about four. These are the operating system and system libraries (as strongly as its base image is pinned), the R package versions (nominally, through the lockfile), the locale and time zone, and the source code. It leaves others to convention or to the base image. The random-number kind is not pinned by default. The BLAS and LAPACK implementation is inherited from the base image (the generated Dockerfile sets `OMP_NUM_THREADS=1`, which fixes the thread count but not the library). The integrity of external data is recorded by provenance but not by checksum, and the fonts and the graphics device are pinned only for some profiles. We know of no integrative framework that currently pins these layers, and an honest L2 or L3 claim names them. The reader will therefore find, at each rung of the ladder in the chapters ahead, an explicit account of what is secured and what remains exposed. That account includes the framework this book teaches.

Check 9: naming the gap honestly

A compendium reaches L3 by this chapter's definition: it rebuilds from clean, its tests pass, and a baseline test compares a reported hazard ratio with a recorded value within a stated tolerance. Is it now safe to call the analysis fully reproducible, with nothing left to disclose?
The answer is on page 103.

Draft, November 13, 2026. Content may change.

3.9. Worked example

To make the ladder concrete, consider a single analysis walked up all four rungs. A biostatistician has a folder containing `analysis.R`, which reads `cohort.csv`, fits a Cox model with the `survival` package, and writes a hazard ratio to `results.csv`. We shall place it on the ladder and name each transition.

At the start the folder is on the analyst's laptop and nowhere else. It is at no rung of the ladder at all, because it cannot even be found by a second person. The first act, placing the folder under version control and pushing it to a repository, lifts it to **L0, locatable**. A reviewer can now find `analysis.R` and read it. Nothing computational is pinned; a reviewer who runs it will install whatever version of `survival` happens to be current, which may not be the version the analyst used.

The analyst now runs `renv::init()` and `renv::snapshot()`, which we develop in Chapter 6. This writes a `renv.lock` recording that the analysis used `survival` version 3.5-7, together with the exact version of every other package in the closure. The folder is now at **L1, pinned packages**. A colleague who restores from the lockfile obtains the same `survival` the analyst used. But the colleague's laptop runs a newer R and a different operating system, and one of the packages links against a system library that is absent, so the restore fails to build.

To close that gap the analyst adds a container, developed in Chapter 7, built from a dated `rocker` base image pinned by its content digest. The image carries a fixed operating system, a fixed R, and the system libraries the packages need, and the lockfile restores inside it. The folder is now at **L2, pinned environment**. It will build the same environment on the colleague's laptop, on a reviewer's, and on a server. Because the analyst computes bootstrap confidence intervals for the hazard ratio, the analyst also sets the RNG kind with an explicit `RNGkind()` call in `analysis.R`, and passes a seed at the call that draws the resamples. Neither goes in the project `.Rprofile`. A seed set at session start makes every draw depend on whatever consumed random numbers earlier in the session, the rule Chapter 11 develops. This closes a session-configuration exposure that L2 does not close by default.

Draft, November 13, 2026. Content may change.

Finally the analyst adds a verification step, developed in Chapter 12. The step rebuilds the image from clean and re-runs `analysis.R`. It then runs a baseline test comparing the regenerated hazard ratio and its confidence limits with the values recorded in a committed baseline file, within a stated relative tolerance. It passes, and a continuous-integration service re-runs it on every change. The folder is now at **L3, verified**. That is, the reported hazard ratio has been shown to regenerate, not merely asserted to. The honest reproducibility statement for the paper can now name the rung, the artifacts, and the layers that remain outside the guarantee. No rung pins the hardware, the numerical libraries, or intrinsic non-determinism. For this analysis they amount chiefly to the cross-architecture numerical determinism of the bootstrap.

> Check 10: which rung, which artifact
>
> The biostatistician's colleague restores the lockfile but skips the container, then reports the analysis as 'reproducible.' Which rung has the colleague actually reached, and what single artifact would the colleague need to add to reach the next one?
> *The answer is on page 103.*

3.10. The cohort, continued

The worked example above follows a composite biostatistician. We now perform the same placement exercise on this book's own running cohort compendium, as it stands in this repository when the chapter is rendered.

```
raw <- load_cohort('../data/raw_data/cohort_raw.csv')
cohort <- clean_cohort(raw)
nrow(raw)
#> [1] 338
nrow(cohort)
#> [1] 337
```

The extract loads and cleans without error, and `clean_cohort()` drops the one duplicated `mrn` the Conventions page describes, leaving the row

Draft, November 13, 2026. Content may change.

count shown above. Whether the compendium is locatable, and whether anything beneath the code is pinned, is a question about which artifacts the repository holds, so we check rather than assert it.

```
repo_state <- c(
  source_code = file.exists('../../R/cohort.R'),
  data_generator =
   ↪  file.exists('../scripts/simulate_cohort.R'),
  lockfile = file.exists('../../renv.lock'),
  dockerfile = file.exists('../../Dockerfile')
)
repo_state
#>    source_code data_generator       lockfile
 ↪  dockerfile
#>           TRUE           TRUE           TRUE
 ↪  FALSE
rung <- if (repo_state[['lockfile']] &&
 ↪  repo_state[['dockerfile']]) {
  'L2'
} else if (repo_state[['lockfile']]) {
  'L1'
} else {
  'L0'
}
rung_name <- c(
  L0 = 'locatable', L1 = 'pinned packages',
  L2 = 'pinned environment'
)[[rung]]
next_artifact <- c(
  L0 = 'a lockfile, `renv.lock`',
  L1 = 'a container recipe, a `Dockerfile`',
  L2 = 'a baseline test that passes after a clean rebuild'
)[[rung]]
```

The code is under version control, so a reader of this book can find `R/cohort.R` and read exactly what ran. The extract itself is not committed; the repository holds the generator, `analysis/scripts/simulate_cohort.R`, which writes the synthetic extract from a fixed seed. That much

Draft, November 13, 2026. Content may change.

is L0, locatable. The lockfile entry above is TRUE and the Dockerfile entry is FALSE, so on its capture artifacts this compendium is at **L1, pinned packages**. What it still lacks for the next rung is a container recipe, a `Dockerfile`. Any layer that no present artifact pins is left to whatever R, system libraries, and packages happen to be installed on the machine that runs this chunk. The chapters on pinning packages (Chapter 6) and pinning the environment (Chapter 7) teach the artifacts that mark L1 and L2. The verification chapter (Chapter 12) is the one that can lift the compendium to L3, once a baseline test compares a quantity computed from `cohort` against a value recorded today; the chunk above does not look for one, so it never reports L3. The reader who returns to this cell after reading those later chapters is looking at the same lines of code. What changes, as artifacts are added, is the rung those lines print.

Check 11: the compendium's own rung

Reading only the `repo_state` output above, what rung does this book's own cohort compendium sit at, and what is the single artifact whose absence keeps it from the next rung?
The answer is on page 103.

3.11. Collaborating with an LLM

We turn finally to the use of a large language model, which is a capable assistant for reasoning about levels. However, it is confident in a register that this chapter has just taught the reader to distrust, and its confidence is not calibrated to the honest level.

! Restricted data

One rule comes first (habit 3). Never paste or upload protected health information, identifiable records, or data held under a data use agreement, nor outputs derived from them, into a model or a coding agent unless three conditions all hold. The institution has approved that endpoint for that class of data. The IRB protocol and the consent permit the processing. The data use agreement, or the provider's

Draft, November 13, 2026. Content may change.

terms, permits it. If any of the three is unclear, the answer is no.
The third condition is the one most often missed. Institutional approval of a tool does not override the terms under which a data provider released the data. NIH, for example, treats sharing controlled-access data through prompts with public generative AI tools, which it glosses as third-party tools, as a breach of the data use certification. It treats a model trained on such data as a data derivative, to be shared only with other approved users and not kept after the project closes (National Institutes of Health, 2025a).
Describe the structure instead: the column names and types, or a small synthetic example generated from the data dictionary, never edited from real rows. A coding agent sees whatever files are in its working tree, so run it in a checkout or container where the real data is not mounted.
An agent also acts, so confine it to a branch and keep merges, pushes, and deposits for yourself (Section 2.10.1).

A directory listing can itself be identifying when file names carry site or participant codes. List only the code, configuration, and lock files, and leave data directories out. The following triples illustrate productive prompts and the verification each demands.

Prompt. 'Here is the listing of my project's code and configuration files, with the data directories omitted. What reproducibility level has it reached on the L0 to L3 ladder, and what would it take to reach the next rung?'

Watch for. Reading a `renv.lock` as L1 and a `Dockerfile` as L2 is the right heuristic as far as it goes. However, no one can tell from a listing whether the lockfile is current, whether the image is pinned by a mutable tag or an immutable digest, or whether a verification run has ever passed. Check whether the answer reports L3 on the strength of a continuous-integration file that has never been run green.

Verification. Confirm each rung against the artifact itself, not its filename. For L1, check that the lockfile is in step with the code. For L2, check that the base image carries a digest and that the library is not shadowed by a bind mount. For L3, find an actual passing run of a baseline test after a clean rebuild, and read the test to confirm it compares outputs with

Draft, November 13, 2026. Content may change.

recorded values under a stated tolerance. The presence of a file is capture asserted; a green baseline run is validation observed.

Prompt. 'Below are the four rungs of a reproducibility ladder with their definitions, followed by a description of my project. Place the project on the ladder. For each rung, name the artifact in the description that meets its definition, or say what is missing. The rungs are cumulative. Say what you could not determine from the description.'

Supply the rows of Table 3.1 and the paragraph on cumulative rungs as the definitions, since a model cannot apply a scale it has not been given. Describe the project in prose: what is under version control, whether a lockfile exists, what the `Dockerfile`'s `FROM` line says, and what the tests compare.

Watch for. Check whether the answer applies the cumulative rule, or credits L2 to a `Dockerfile` in a project with no lockfile. Check whether it treats a base image named by tag as pinned in the same way as one named by digest. Check also whether it awards L3 to a test suite that compares nothing with a recorded baseline, which this chapter calls L2 confirmed, not L3.

Verification. Place the project yourself, from the artifacts, before reading the answer, and compare the two placements rung by rung. Then give the model a case whose answer you know, such as a project with a `Dockerfile` and no lockfile, and confirm that it does not award L2. A model that misplaces the known case should not be trusted on the unknown one.

Prompt. 'My simulation gives slightly different numbers each time even inside the container. Why, and is it a bug?'

Watch for. Check whether the answer blames a layer that the container has already pinned, such as a package version or the operating system. Check also that it distinguishes an unseeded RNG (a real defect) from low-order-digit variation due to BLAS threading (an inherent limit at L2).

Verification. Trace the discrepancy to a layer of the determinant stack. If the numbers differ in all digits, look to the RNG kind and the seed, which the analyst controls. If they differ only in the low-order digits, suspect the numerical-library layer, which no rung of the capture ladder pins, and decide whether that residual matters for the inference at hand.

Draft, November 13, 2026. Content may change.

A better prompt. 'My simulation, run twice in the same container, gives the two outputs pasted below. Also pasted are the `sessionInfo()` output, which names the BLAS and LAPACK libraries, the `RNGkind()` output, the line of my code that sets the seed, and the thread count. Say which layer of the determinant stack each difference most plausibly comes from, and for each give one command I can run to confirm or exclude it.'

3.12. Exercises

1. For an analysis of your own, list the nine layers of the determinant stack and, for each, state whether your current setup pins it, and by what artifact. Identify the lowest layer you have left unpinned.

2. Take a published paper in your field that provides code. Without running it, place it on the L0 to L3 ladder from the materials the authors supply, and write the one-sentence reproducibility statement the paper should have carried.

3. Explain, to a colleague who believes that a Docker image makes an analysis 'fully reproducible', two determinants that the image does not pin and one circumstance in which each would cause two runs to disagree.

4. The framework's documentation grades a project 'Level 4' on its five-level tool-adoption scale. State what this does and does not tell you about the project's rung on the L0 to L3 architectural ladder, and name one artifact you would inspect to settle the question.

5. A colleague reports that a finding 'replicated' in a second cohort. A reviewer reads this as an ACM artifact badge and asks for the container. Explain the miscommunication in terms of the two conventions for the word, and state which sense the colleague almost certainly intended.

6. Construct a small simulation in R that draws a sample under a fixed seed, and show that changing the RNG kind with `RNGkind()` changes the draw while leaving the seed unchanged. Explain which layer of the determinant stack this demonstrates and why a container alone would not protect against it.

Draft, November 13, 2026. Content may change.

7. Run `cohort <- clean_cohort(load_cohort('analysis/data/raw_data/cohort_raw.csv'))` and report `nrow(cohort)`. Then check `file.exists()` for `renv.lock` and `Dockerfile` at the project root, as the `repo_state` chunk in 'The cohort, continued' does. From that output alone, state the single rung of the L0 to L3 ladder this book's own cohort compendium occupies, and name the one committed artifact, absent when you run the check, that would be needed to lift it to the next rung.

8. This exercise returns to the computational-versus-statistical distinction of Chapter 2 (Chapter 2). That chapter shows that the R 3.6.0 sampler correction changed the draws a seeded `sample()` call produces, with identical code and an identical seed. Name the exact layer of this chapter's determinant stack responsible, and state which rung, if any, of the L0 to L3 ladder would have protected a 2018 analysis against this specific change.

3.13. Solutions

Attempt an exercise before expanding its solution. Two of these ask you to work with an analysis or a paper of your own, so they have no single correct answer; for those the box says what a good answer establishes.

Exercise 1

No single correct answer. What a good answer establishes is that you can name the layers in order and attach an artifact, or an admission, to each.
Work up from the base. The **hardware** is pinned by nothing short of running on the same machine. The **numerical libraries** are pinned by no rung. The container image determines which BLAS is installed, but the ladder does not require it to be fixed, and the thread count and the processor still vary. The **operating system and system libraries** are pinned by the container image. The **R runtime** is likewise pinned by the container image, and merely *recorded* by the lockfile's `R` block. The **package versions** are pinned by `renv.lock`. The **session configuration** is partly pinned by the image through

Draft, November 13, 2026. Content may change.

the locale and the time zone, and not pinned at all for the RNG kind, the contrasts, and the printed digits. The **analysis code** is pinned by version control, while the **input data** are pinned by a checksum or an archived deposit, and are often only locatable. Lastly, **intrinsic non-determinism**, such as parallel reduction order, is pinned by nothing.
Your lowest unpinned layer is most likely either the numerical libraries or the session configuration, and both answers are common and honest. If your lowest unpinned layer is the input data, you have discovered something more urgent than anything this chapter teaches, because the analysis is not reconstructable at all.

Exercise 2

No single correct answer. What a good answer establishes is that the rung is read off the *deposited artifacts*, not off the quality of the paper or the reputation of the authors.
The procedure is mechanical. Is there code and data that can be located? That is L0. Is there a lockfile, or an equivalent exact version record? That is L1. Is there a container recipe, or another artifact pinning the runtime and the system libraries? That is L2. Is there evidence that the outputs were regenerated from the pinned definition and compared with recorded values within a stated tolerance, shown by a baseline test and a CI log or verification report of it passing? That is L3. A CI log that only shows the code ran is not enough. Stop at the highest rung for which an artifact is actually present.
The reproducibility statement should then claim that rung and no more. For the common L0 case: 'The analysis code and the derived dataset are deposited at [location]. The software environment was not recorded, so re-execution may require package versions that differ from those used here.' The second clause is the part most papers omit, and it is the part that makes the statement honest rather than merely modest.

Draft, November 13, 2026. Content may change.

Exercise 3

Two determinants an image does not pin, with a circumstance for each.

The numerical libraries. An image built on a base that links against a BLAS chosen at run time, or one run on hosts that expose different thread counts or processors, will reorder floating-point summations. Two runs of the same image on two machines can then return a least-squares coefficient, computed through `crossprod()` and `solve()`, agreeing to five decimal places and differing in the sixth. The image pinned the software and not the arithmetic beneath it.

The session configuration, specifically the RNG kind. An image fixes the R version, and the R version determines the *default* sampler, so this exposure is narrower than it first appears. It bites when the analysis sets `RNGkind()` itself, or inherits it from a project `.Rprofile` that is mounted rather than baked in, or when a package changes the generator as a side effect. Two runs then draw different samples from the same seed, and a bootstrap interval moves.

A third acceptable answer is the input data: an image mounting a host directory reproduces the environment and not the extract, so a changed file underneath produces a different result with the image unchanged. A fourth is intrinsic non-determinism from parallelism, where the reduction order of a parallel sum varies run to run on the same machine.

In general, a container pins the environment, and the environment is not the whole stack.

Exercise 4

'Level 4' on the framework's scale means the project has adopted a particular set of *tools*: an R project, `renv`, Docker, and unit testing. It is a statement about tool adoption, and it is read from the bottom up, so Level 4 implies the first three as well.

What it tells you about the architectural ladder is therefore quite a lot, but indirectly and with one gap. The presence of `renv` implies the machinery for L1; the presence of Docker implies the machinery for L2. What it does not tell you is whether any of it has been *verified*, because the tool scale counts adoption and the architectural ladder

Draft, November 13, 2026. Content may change.

grades locking and verification. A project can hold every tool in the list and have never rebuilt from its pinned definition. In that case it sits at L2 and is asserted rather than shown to reproduce. L3 requires a baseline comparison, which neither Level 4 nor Level 5 implies by itself. Unit tests and continuous integration can both be present while nothing is compared with a recorded result.

The artifacts to inspect are a baseline test and the continuous-integration record. The record should be a log showing that the container was rebuilt from the recipe, the analysis re-run, and its outputs compared with recorded values within a stated tolerance. Absent that, the honest reading of 'Level 4' is L2. Inspecting `renv.lock` and the Dockerfile tells you the capture artifacts exist; only the CI record tells you the capture works.

Exercise 5

The colleague almost certainly meant the National Academies sense: a second cohort, collecting its own data, found a consistent result. That is *replication*, a statement about the durability of the finding, and it is the sense in which the word is ordinarily used in health research.

The reviewer read it in an artifact-badging frame, where the pair of words attaches to re-execution rather than to new data. Under the current 2020 ACM vocabulary the two conventions actually agree, so the current standard cannot explain the request. The miscommunication traces to one of two places. One is the pre-2020 ACM badging, which attached the two words in the opposite direction. The other is a field usage in which 'reproduce' names an independent study rather than a re-run. A reviewer carrying the sense of the words from an older document into a current one is misled silently, with no error to warn them.

The practical resolution is that the container request is a category error given what the colleague did. A container would let the reviewer re-execute the *original* analysis, which is not what the colleague claimed and not what a second cohort demonstrates. The colleague should answer by naming the convention, stating that the second cohort collected its own data, and offering the artifact that actually matches the claim, which is the second cohort's own analysis code.

Draft, November 13, 2026. Content may change.

Exercise 6

The demonstration is the worked example of Chapter 2 in miniature.

```
draw <- function(kind) {
  suppressWarnings({
    RNGkind(sample.kind = kind)
    set.seed(42)
    sample(1:10, 5)
  })
}
original <- RNGkind()
draw('Rounding')
draw('Rejection')
suppressWarnings(RNGkind(
  kind = original[1], normal.kind = original[2],
  sample.kind = original[3]
))
```

The two calls share a seed and return different draws. Restoring the original generator afterward matters, since `RNGkind()` sets session state that would otherwise persist and silently affect everything downstream.

The layer demonstrated is the **session configuration** layer, which sits above the package versions and below the analysis code. It is the layer this chapter singles out as only partially pinned at L2. An image fixes the locale and the time zone, and the R version it carries fixes the *default* sampler. However, the RNG kind, the contrasts, and the number of printed digits are all settable from inside the session and are not properties of the image.

A container alone would not protect against this because the change is made by the analysis, inside the container, after the environment is fixed. Two runs of the identical image produce different draws if the code sets the sampler differently, or if a mounted `.Rprofile` does. The protection lies in an explicit declaration, which a heavier container cannot supply. Concretely, set `RNGkind()` in the analysis, record it alongside the seed, and treat it as part of the analysis code rather than part of the environment.

Draft, November 13, 2026. Content may change.

Exercise 7

The solution for this exercise is in the instructor guide, which is distributed separately to course instructors. Self-study readers who want to check their answer can compare the autogradable value against those reported in the chapter's own 'The cohort, continued' section.

Exercise 8

The layer is the **R runtime** layer of this chapter's determinant stack, not the package-versions layer above it, and not the analysis code further up. The sampler that turns a seed into a draw is, by default, a behavior of R itself. The change arrived with a new R release rather than with a new package version or an edited script. Exercise 6 places the RNG kind in the session configuration instead because there the analysis *overrides* the default; here the default itself moved.
By this chapter's own stack figure, the R runtime is secured only at **L2, pinned environment**, the same rung that secures the operating system and system libraries. A lockfile at L1 fixes the packages but not the R version underneath them. So an L0 or L1 analysis in 2018 would have had no protection at all. Whichever R release the person re-running it happened to install would supply whichever sampler that release shipped. Only an analysis built from a container image pinned to an R release predating 3.6.0 would have run the old sampler regardless of when it was restored. That protection is exactly what the L1-to-L2 step of this chapter's worked example buys, and what a lockfile alone does not.

Answers to the checks

Check 1. *from earlier chapters* (page 70)

Neither change in the re-run 2018 analysis is a replication failure. The same code ran on the same data, so by the definitions of Chapter 1 and Chapter 2 this is a failure of computational reproducibility, with a computational cause. The moved interval traces to R 3.6.0, which changed the default `sample.kind`: the seed was fixed, but the sampler that turns it into draws

Draft, November 13, 2026. Content may change.

was not (Table 2.1). The `NULL` levels trace to R 4.0.0, which flipped the `stringsAsFactors` default to `FALSE`, so `arm` now arrives as a character vector. Both are defaults of the R runtime, and pinning the R release, which a container does at L2, fixes both. Each can also be closed in the analysis code: an explicit `RNGkind()` call, the only fix for an override, and an explicit `stringsAsFactors` argument, as `load_cohort()` writes it.

Check 2. *what changed, and what did not* (page 72)

A package the analysis depended on changed its default optimizer between the two runs. That change sits on the package-versions layer of the determinant stack, not on the analysis code or the data. The model specified was identical on both machines; what differed was an unpinned determinant beneath it. Calling the discrepancy purely 'statistical' hides a computational failure behind a word that belongs to a different kind of disagreement, one about the data or the model. Its size is still a statistical question. An estimate that moves when only the optimizer changes comes from a poorly conditioned or unconverged fit, so it should be refit under other optimizers and reported as optimizer-sensitive. Pinning the package makes that fragile estimate reproducible; it does not make it trustworthy.

Check 3. *whose judgment is it?* (page 73)

The dashboard can detect the capture artifacts (a lockfile and a container recipe) and so can compute the rung mechanically; that much is automatable. It cannot decide whether L2 was the right level to target for this analysis given its stakes. Nor can it know which unpinned layers, among the numerical libraries, the RNG kind, or an external data source, are live exposures for this particular question. Judging the match of effort to stakes, and naming what remains exposed, stay with the analyst no matter how good the dashboard is.

Check 4. *locating a failure on the stack* (page 78)

Neither the code nor the data nor the pinned layers can be at fault, since all were identical. A discrepancy confined to the low-order digits is the signature of the numerical-library layer. `crossprod()` hands the matrix product to the BLAS, and `solve()` hands the linear system to LAPACK. The two machines used a different BLAS implementation, or the same one with a different thread count, and so the floating-point summations were reordered. It is not a bug in the analysis. It is the reason that

Draft, November 13, 2026. Content may change.

byte-for-byte identity across different hardware is a stronger claim than number-for-number agreement on a fixed architecture. And it is a layer that no rung of the capture ladder pins.

Check 5. *capture or validation?* (page 82)

The colleague is correct on the narrow point and wrong on the conclusion. Continuous integration is a validation feature, not a capture feature: it pins no new layer, and removing it would leave the compendium exactly as reconstructable as before. What it adds is confirmation. It re-runs the captured analysis in a clean environment that carries none of the author's undeclared state. It therefore catches the silent decay that a captured-but-never-retested analysis suffers, and the 'works on my machine' failure at its source. The laboratory has moved from L2 asserted to L2 confirmed: the environment is shown to build and the code to run. A baseline test, comparing regenerated outputs with recorded values, is what would make it L3.

Check 6. *which word, which sense?* (page 85)

It is replication: an independent cohort means new data, and the National Academies convention attaches 'replicated' to a new study and 'reproduced' to re-running the original artifacts on the original data. The sentence risks being misread because of the current ACM badge sense. A reader applying it would expect the word 'reproduced' to mean an independent team re-ran the authors' own artifacts, not that a second cohort was collected. The fix is to write 'replicated', not 'reproduced', or to spell out which convention is meant.

Check 7. *two scales, one word* (page 86)

No. On the tool-adoption ramp, Level 2 means `renv` has been added, which is the architectural chapter's L1, not its L2; the container arrives at the ramp's Level 3. The student's sentence silently mixed the two scales' numbers. The safe habit is to never write a bare number without saying which scale it belongs to. This book's own convention, stated above, is to let an unqualified 'L2' always mean the architectural rung.

Check 8. *grading an R package* (page 87)

It is at L2, pinned environment, confirmed: the lockfile and the container pin the packages and the environment, and the clean rebuild and passing

Draft, November 13, 2026. Content may change.

test suite show the package builds and runs. It is not at L3, because none of its tests is a baseline test. Passing means the code ran without error, not that a key function's output still matches a recorded reference value within a tolerance. Reaching L3 needs at least one test that compares a fixed input's output against a value committed earlier.

Check 9. *naming the gap honestly* (page 87)

No. L3 shows that the pinned definition regenerates the reported quantity within a stated tolerance. It does not pin the numerical libraries, the hardware, or intrinsic non-determinism, none of which any rung of the capture ladder secures. An honest statement at L3 still names these as exposed, for example that the bootstrap interval has not been checked for agreement across a different BLAS or CPU architecture. 'Fully reproducible' claims a closed stack that no present framework, including the one this book teaches, actually closes.

Check 10. *which rung, which artifact* (page 89)

Restoring the lockfile reaches L1, pinned packages, and no further. The operating system, the R runtime, and the system libraries the colleague's machine happens to carry are still whatever that machine provides, not what the original analyst used. Calling this 'reproducible' without qualification overstates by at least one rung. The single artifact that closes the gap to L2 is the container image, built from the same recipe the original analyst used, which pins the layers the lockfile cannot reach.

Check 11. *the compendium's own rung* (page 91)

L1, pinned packages, as the `repo_state` chunk above prints. The code is under version control, which is L0. L1 requires a lockfile, `renv.lock`, recording the exact version of `tibble` and every other non-base package the two functions call; base packages such as `stats` and `utils` have no record of their own because the R version fixes them. L2 requires a container recipe as well. The artifact the compendium still lacks for the next rung is a container recipe, a `Dockerfile`. Chapter 6 teaches the lockfile and Chapter 7 the container recipe.

3.14. Cheat sheet

Draft, November 13, 2026. Content may change.

Rung or rule	What it means or checks
L0 locatable	Source under version control; nothing computational pinned
L1 pinned packages	A lockfile (`renv.lock`) records every package's exact version
L2 pinned environment	A container or Nix recipe also fixes the OS, R runtime, and system libraries
L3 verified	A baseline test compares regenerated outputs with recorded values, within a stated tolerance, after a clean rebuild
Capture vs. validation	Capture (a lockfile, a container) sets L0 to L2; validation (CI, a baseline test) adds nothing pinned, confirms the captured level, and is what reaches L3 (habit 5)
Reproduced (this book)	Same code, same data, re-run; contrast replicated: a new study, new data
`zzc verify --full`	Reports L3 only if the image rebuilds, tests pass, and a `test-baseline*.R` file appears in the test log
Honest statement	Name the rung, the artifact, and what remains exposed; never claim 'fully reproducible' (motto)

3.15. Further reading

- National Academies of Sciences, Engineering, and Medicine (2019) is the National Academies consensus report that fixes the reproducible-versus-replicable terminology this book follows; read its opening chapter for the definitions and the reasoning behind them.
- Peng (2011) introduces the reproducibility spectrum that our ladder consolidates, and argues for reproducibility as a minimum standard

Draft, November 13, 2026. Content may change.

when full replication is impractical; the earlier Peng (2009) applies the idea to epidemiology directly.

- Leek & Peng (2015) makes the case that reproducibility is a floor and not a ceiling, and Patil et al. (2016) offers a statistical framing of the distinction between reproducing a computation and replicating a finding.
- Association for Computing Machinery (2020) documents the artifact-review badges whose 'available', 'functional', 'reusable', and 'reproduced' ordering grades related but different things from our ladder; read it alongside National Academies of Sciences, Engineering, and Medicine (2019) to see how the 2020 revision aligns the two vocabularies.
- The Turing Way Community (2022) crosses the reproducibility axis with a what-varies axis, same or different data against same or different analysis, and is the most accessible community handbook for a first-year student.
- For the determinant stack and the capture-versus-validation distinction as a framework applies them, Sandve et al. (2013) and Wilson et al. (2017) give the practices, the former stressing the recording of random-number seeds and exact versions that the ladder's session-configuration layer names.

Draft, November 13, 2026. Content may change.

4. Version Control with Git and GitHub

Maintaining a high degree of transparency in scientific reporting is essential not just for gaining trust and credibility within the scientific community but also for facilitating the development of new ideas.

Karthik Ram, *Git Can Facilitate Greater Reproducibility and Increased Transparency in Science* (2013)

This chapter has three practical aims. By its end you will be able to write a commit history that answers 'what changed and why' for any analysis you run. You will be able to build a `.gitignore` that keeps protected health information out of a repository before the first commit rather than after a breach. Finally, you will be able to state exactly which single rung of the reproducibility ladder a GitHub URL has actually earned, and which it has not.

We begin by placing version control on the determinant stack and naming the failure it cures. We then set out the analyst's three judgments, the git model, and the reading of a history, before turning to what must never be committed. Branches, the undoing of mistakes and merge conflicts, and remotes follow, and we close the exposition with what git does not do. A worked example, a note on collaborating with an LLM, and the exercises complete the chapter.

Before you start

Reading time. About 65 minutes.
Prerequisites. Chapter 2 and Chapter 3, for the determinant stack and the L0 rung that this chapter's practice secures; no other chapter is assumed. Statistically, nothing is required. R knowledge assumed: none beyond reading a short script. The git and shell commands

Draft, November 13, 2026. Content may change.

that make up most of this chapter are independent of R, and the two R chunks that appear are read for their output, not written by the reader. Git itself must be installed and configured; the preflight in the setup appendix (Git preflight) checks both in a few minutes.

By the end of this chapter you will be able to answer:

- Can you describe the failure mode version control cures, the drawer of `analysis_final_v2_FINAL.R` files, and explain why the cure is a different kind of object, a history, rather than a tidier naming scheme?
- Can you explain the git model, the repository, the commit as an atomic, message-bearing unit, and the history as a tamper-evident chain, together with the limits of that record as evidence?
- Can you perform the everyday git operations at the shell, staging and committing, reading the log and the diff, and writing an archetype-appropriate `.gitignore`?
- Can you state what must never be committed in health research, and explain why deleting such a file in a later commit does not remove it from the history?
- Can you place version control on the reproducibility ladder, name the one rung it secures, and list the layers of the determinant stack it leaves exposed?

Outline.

- Where this sits on the stack.
- Orientation: the drawer of `FINAL` files, and the object that replaces it.
- The analyst's contribution: three judgments no tool automates.
- What version control records, and what it cures.
- The git model: repository, commit, history.
- Reading the history: the log and the diff.
- What must never be committed: `.gitignore` and PHI.
- Branches, lightly.
- Undoing things, and merge conflicts.
- The remote: GitHub, GitLab, and collaboration.
- What git does not do.
- Worked example: an untracked folder to a tagged repository.

Draft, November 13, 2026. Content may change.

- Collaborating with an LLM.
- Exercises, a cheat sheet, and further reading.

Check 1: from earlier chapters

A collaborator's repository holds `analysis.R` and a `Dockerfile` whose build installs packages from the current CRAN release; it has no `renv.lock`. Its README claims L2. A year later the image is rebuilt. One call now fails because a package changed its interface. Inside the same rebuilt image, a regression coefficient computed on the cluster differs from the laptop's in the sixth decimal place. Using Chapter 2 and Chapter 3, which rung has the repository earned? For each discrepancy, which layer is responsible, and which tool or rung addresses it?
The answer is on page 154.

4.1. Where this sits on the stack

Version control is this book's clearest case of pure capture at the very top of the determinant stack. Figure 4.1 marks the layer it secures. Before a repository exists, the analysis code is not merely unpinned; it is not reliably even *locatable* by anyone but the analyst who wrote it. Placing it under git, and pushing it to a remote, is the single act that lifts that top layer from nothing to L0. The input-data layer directly beneath it is deliberately left unhighlighted here, because this chapter's central judgment runs the other way. Health data, unlike code, often must be kept *out* of the very history that pins everything else. That is the subject of the section on `.gitignore` below.

```
plot_determinant_stack(
  captures = c('Analysis code'),
  title = 'Version control captures the analysis code;
   ↪  later chapters capture the rest')
```


Draft, November 13, 2026. Content may change.

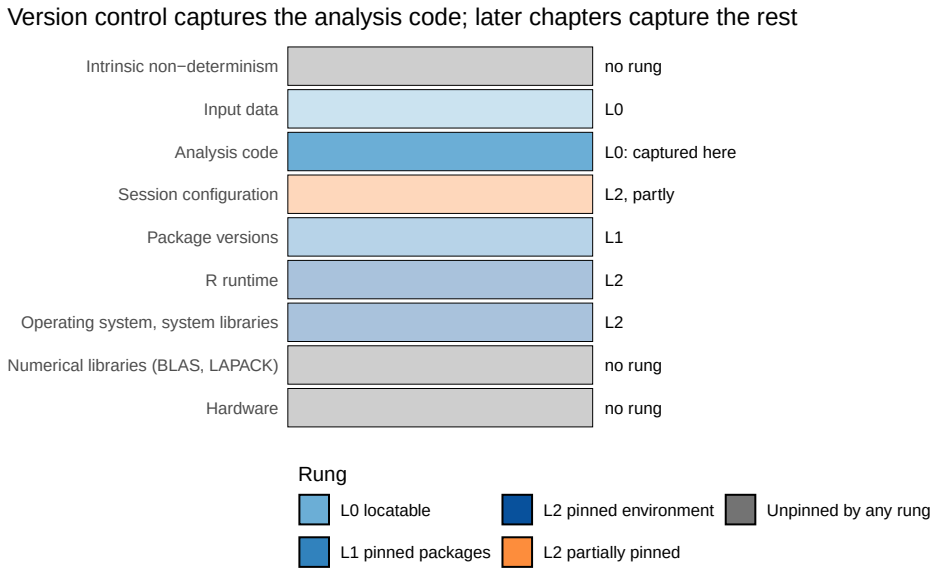


Figure 4.1.: The determinant stack of Chapter 3, with the analysis-code layer marked as captured here. Version control is what first pins the code, lifting it from unlocatable to L0; the input-data layer immediately below is left unhighlighted on purpose, because this chapter spends as much effort keeping some data out of the history as it spends getting the code into it.

4.2. Orientation

If we open the folder of almost any analysis that has run for more than a month without version control, the same artifact greets us. There is `analysis.R`, and beside it `analysis_v2.R`, `analysis_final.R`, `analysis_final_v2.R`, and, inevitably, `analysis_final_v2_FINAL.R`. A dated copy or two, `analysis_2025-03-14.R`, sits among them, and a `analysis_reviewer.R` from the week the referee reports came back. Every file is a manual snapshot, taken because the analyst feared to overwrite a version that worked. The scheme fails in three ways at once. It does not say which file produced the figure in the submitted manuscript. It does not say what differs between any two of them, so recovering a change

Draft, November 13, 2026. Content may change.

means reading both side by side. And it does not say *why* any change was made, because the reason lived in the analyst's head and is now gone.

This is the failure that version control exists to cure. What, then, is the cure? A tidier naming scheme will not do; the cure is a different object. A version-control system keeps a single working copy of each file. It records, as a separate and complete history, every state that copy has passed through, together with a note on each change saying what it was and why it was made. The drawer of copies collapses back into one `analysis.R`, and each copy becomes an entry in a history that can be read, searched, and, if need be, restored. The tool that has won this role, across the sciences and across software of every kind, is git. The service that most often hosts a git history on the network is GitHub, with GitLab its principal institutional sibling. This chapter teaches both from first principles. Version control is the single most universally taught element of a modern reproducibility course, and it is the foundation on which the rest of this book stands.

Check 2: what the drawer loses

Of the three things the `FINAL`-file drawer loses, the identity of the working version, the content of each change, and the reason for each change, which one is recoverable, laboriously, by reading the files side by side, and which is not recoverable at all once the analyst forgets or leaves?
The answer is on page 154.

It is also the foundation in a precise, technical sense. The next chapter, Chapter 5, makes the compendium the unit of reproducible work, and Chapter 3 placed the bare compendium at the bottom rung of the ladder, L0 locatable. The artifact that marks L0 is exactly a version-controlled repository. That is, an analysis is locatable when, and only when, it is under version control and can therefore be found, read, and referred to by a second person. Everything the later chapters add is installed on top of a version-controlled history: the lockfile of Chapter 6, the container of Chapter 7, the continuous integration of Chapter 12.

Draft, November 13, 2026. Content may change.

4.3. The analyst's contribution

Version control automates the mechanics of history perfectly and automates none of the judgment that makes a history useful and safe. Three decisions belong to the analyst and are the subject of this chapter as much as the commands are.

1. **Deciding what must never be committed.** A commit is, by design, permanent and copied to every clone of the repository. The judgment of what may safely be made permanent and public, and what must be kept out of the history entirely, is the analyst's. In health research it is a judgment with legal and ethical force. Protected health information, identifiable data, and credentials must never enter the history. As we shall see, deleting them in a later commit does not remove them from it. This is the first and gravest of the three judgments.
2. **Choosing the granularity and the message of a commit.** A commit is a unit of change. How much change it should contain, and what its message should say, are matters of craft that no tool decides. A history of one enormous commit reading 'did the analysis' is almost as useless as no history at all. A history of small, coherent commits whose messages say why each change was made is a readable record of decisions.
3. **Deciding when to branch and when to tag.** Whether a piece of work warrants a branch of its own, and whether a particular state of the analysis warrants a permanent tag, are design decisions about the shape of the history. They are cheap to make and expensive to omit. The state of the code at the moment a manuscript was submitted is worth a tag, and no tool will place it for the analyst.

Check 3: which judgment is health-specific

Of the three judgments listed above, which one carries legal and ethical force specific to health research rather than being craft judgment that applies to any software project, and why?
The answer is on page 154.

Draft, November 13, 2026. Content may change.

The vocabulary of this chapter

New terms are defined again on first use; they are collected here for reference.

- **Repository (repo).** A directory whose contents and full history git tracks, held in a hidden `.git` subdirectory. A *clone* is a complete copy of a repository, history and all.
- **Working tree.** The ordinary files in the repository directory as they currently exist on disk, as opposed to the versions recorded in the history.
- **Commit.** A recorded snapshot of the tracked files at one moment, carrying an author, a timestamp, a message, and an immutable identifier (a *hash*). The atomic unit of history.
- **Staging area (index).** The set of changes marked, with `git add`, to go into the next commit. Staging lets a commit hold a chosen subset of the current changes.
- **`.gitignore`.** A file listing path patterns that git should not track, used to keep derived outputs, large data, and, above all, sensitive files out of the history.
- **Branch.** A movable name for a line of development. The default branch is conventionally called `main`, although git itself still names it `master` unless told otherwise, as with `git init -b main`.
- **Remote.** A copy of the repository hosted elsewhere, typically on GitHub or GitLab, used for backup, distribution, and collaboration. `origin` is the conventional name of the primary remote.
- **Tag.** A permanent, human-readable name for one specific commit, used to mark a milestone such as a manuscript submission.
- **Pull request (merge request).** A proposal, made on the remote, to merge one branch or fork into another, reviewed before it is accepted.

4.4. What version control records, and what it cures

The value of a version-control system is easiest to state as the negation of the `FINAL`-file failure. That failure loses three things, and version control restores each.

It loses the *identity of the working version*. With a drawer of dated copies, no file is authoritative. With git, there is one `analysis.R`, and its history holds every prior state, so the question 'which version is current' cannot arise. It loses the *content of each change*. Two snapshot files differ in ways that can be recovered only by comparing them line by line by hand. A git history presents each change as a *diff*, a precise statement of the lines added and removed. Git computes the diff on demand from the snapshots it stores, since git keeps full snapshots, not diffs. Lastly, it loses the *reason for each change*, the single most valuable and most perishable piece of information about an analysis. Why was the outlier-exclusion threshold changed from three standard deviations to two? A dated file cannot say. A commit whose message reads 'Tighten outlier threshold to 2 SD per reviewer 2's comment' says exactly why, and the record stays with the code.

The recorded reason is what makes version control an instrument of scientific accountability. A computational result is the end of a chain of decisions, each of which could have gone otherwise. In a regulated analysis, a clinical trial subject to audit, a study destined for a guideline, the chain of decisions is itself an object of scrutiny. 'Why did the estimate change between the interim and the final analysis' is a question an auditor is entitled to ask and the analyst is obliged to answer. A commit history in which each change carries its reason is most of the answer, already written, dated, and attributed. It is, however, a record of what the committers *asserted*, not a regulated audit trail. The next section explains the difference.

> Check 4: what one message restores
>
> A commit message reads 'Tighten outlier threshold to 2 SD per reviewer 2's comment.' Which of the three things the `FINAL`-file drawer lost does this message, by itself, restore, and which two does it leave to the rest of git's model?
> *The answer is on page 154.*

Draft, November 13, 2026. Content may change.

4.5. The git model: repository, commit, history

Git is often taught as a list of commands to memorize, which is why it is so often found bewildering. It is far better understood as a small and coherent data model, after which the commands follow naturally. There are three ideas.

The first is the **repository**. A git repository is an ordinary directory, the analyst's project, that git has been told to track. Git then maintains, in a hidden `.git` subdirectory, the complete history of everything under that directory. The ordinary files the analyst edits are the *working tree*; the `.git` directory is the history. Creating a repository is a single command run once in the project directory.

```
cd diabetes-cohort
git init -b main
```

The `-b main` names the first branch `main`; without it, git as of version 2.56 names it `master`, and the `git push -u origin main` commands later in this chapter would then fail. The git preflight of the setup appendix (Git preflight) sets `main` as the default once per machine, together with the name and e-mail address every commit records. It is worth running before this chapter's first command.

The second idea, and the central one, is the **commit**. A commit is a snapshot of the tracked files at one moment, recorded together with four things: who made it, when, a message describing it, and a hash that names it. That hash is a forty-character SHA-1 string, usually shown in an abbreviated form such as `a1b2c3d`. It is computed over the whole commit object, and the snapshot alone does not determine it. The commit object references the snapshot, folds in the metadata, and records the hash of the commit before it. (Git also supports repositories that use the longer SHA-256 hash of sixty-four characters, though SHA-1 remains the default.) Because each commit's hash incorporates its parent's, the history is a chain. No past entry can be altered without changing its hash and the hash of every commit after it. The hashes are, for practical purposes, unique, and git guards against the known SHA-1 collision attacks with built-in collision detection. The chain makes a history *tamper-evident* in a narrow

Draft, November 13, 2026. Content may change.

sense. Anyone who recorded a commit's hash, in a tag, a manuscript, or a colleague's clone, can detect that the history leading to it was later changed.

But how much evidence is tamper-evidence? It is weaker than it sounds, and anyone headed for regulated work should know exactly how much weaker. The author's name and e-mail address, and both dates a commit carries, are whatever the committer's own git client supplies. Setting `GIT_AUTHOR_DATE` and `GIT_COMMITTER_DATE` and passing `--author` produces a commit under another person's name dated years in the past, which git accepts without complaint.

```
GIT_AUTHOR_DATE='2019-03-01T09:00:00' \
GIT_COMMITTER_DATE='2019-03-01T09:00:00' \
  git commit --author='Someone Else <someone@example.org>'
   ↪  \
    -m 'Backdated commit'
```

History is also rewritable by design. `git commit --amend`, `git rebase`, and `git filter-repo` all replace commits with new ones, and in a repository the analyst controls nothing prevents it. The hash chain detects a rewrite only for someone holding an earlier hash to compare against. Assurance is added from outside the commit. One source is commits signed with a key bound to a verified identity. Another is a forge configured to protect the main branch against force-pushes and to log pushes, reviews, and merges on its own servers with its own clock. Even then, a git history is not, by itself, the audit trail that FDA's regulations require of electronic records in the studies they cover. That trail must be secure, computer-generated, and time-stamped, and must record operator actions independently (*21 CFR Part 11*, n.d.). Within those limits, a well-kept history is an honest and very useful record of how an analysis was built. The rest of this chapter treats it as that.

The third idea is the **history** itself, the chain of commits, each pointing back to its parent, running from the first commit to the most recent. Reading that chain is reading the story of how the analysis came to its current state. A branch, which we treat below, is nothing more than a movable name pointing at one commit in this structure.

Draft, November 13, 2026. Content may change.

The everyday cycle of work has just two steps against this model. They exist because git deliberately separates *choosing* what goes into a commit from *making* the commit. First one *stages* the changes to be recorded, with `git add`; then one *commits* the staged changes, with `git commit`, attaching the message. The staging step is not ceremony. It lets a single commit hold a coherent subset of the work in progress, so that an afternoon's editing across several concerns can be recorded as several focused commits.

```
# edit analysis.R, then see what has changed
git status

# stage the file for the next commit
git add analysis.R

# record the staged change with a message
git commit -m "Add Cox model for 30-day readmission"
```

The message is the part of a commit that carries the scientific value. The convention, which merits adoption from the first day, is a short imperative summary line of about fifty characters. It is optionally followed by a blank line and a paragraph giving the *why*. 'Fix bug' is a wasted message. 'Correct person-time denominator to exclude pre-enrollment follow-up' is a message that will still be intelligible, and still exculpatory, at an audit three years later.

Check 5: why stage at all?

A colleague argues that the staging area is needless friction: since one almost always wants to commit everything one has changed, git should simply commit the whole working tree and spare the analyst a `git add`. What does the staging step buy that a commit-everything model would lose?
The answer is on page 155.

Draft, November 13, 2026. Content may change.

4.6. Reading the history: the log and the diff

Once a history exists, how do we read it? A history that cannot be read is of little use, and git provides two complementary views of it. The **log** is the list of commits, most recent first, each with its hash, author, date, and message. It is the table of contents of the analysis's development.

```
# the full log
git log

# a compact one-line-per-commit view
git log --oneline

# which commits touched analysis/scripts/
git log --oneline -- analysis/scripts/
```

In a scratch copy of the worked example below, three commits in, the compact view reads as follows. Each line is the abbreviated hash and the summary line of the message, most recent first.

```
# output abridged, from git 2.56.0
git log --oneline
#> c7d4711 Tighten outlier threshold to 2 SD per reviewer 2
#> a9aa55f Add cleaning script for registry extract
#> 936e050 Initialize compendium and .gitignore
```

The **diff** is the complementary view. Where the log lists the changes, the diff gives the content of one. `git diff` shows the changes in the working tree that have not yet been staged. `git show` displays the full diff of any single commit named by its hash, and `git diff` between two commit hashes shows everything that changed between them.

```
# what have I changed but not yet staged?
git diff

# show one commit's changes in full
```


Draft, November 13, 2026. Content may change.

```
git show a1b2c3d

# what changed between two points in the history?
git diff a1b2c3d f4e5d6c
```

Here is the diff from the same scratch repository, taken just before the third commit, when one line of the cleaning script had been edited. A line beginning `-` is the committed version and a line beginning `+` is its replacement. The unmarked lines are context, and the `@@` header gives their position.

```
# output abridged, from git 2.56.0
git diff
#> diff --git a/analysis/scripts/01-clean.R
 ↪  b/analysis/scripts/01-clean.R
#> --- a/analysis/scripts/01-clean.R
#> +++ b/analysis/scripts/01-clean.R
#> @@ -1,3 +1,3 @@
#>  cohort <-
 ↪  read.csv('analysis/data/raw_data/extract.csv')
#>  cohort <- subset(cohort, age >= 18)
#> -sd_cut <- 3
#> +sd_cut <- 2
```

Together the log and the diff answer the two questions the `FINAL`-file drawer could not. The log tells us *when and why* a thing changed, and the diff tells us *what exactly* the change consisted of. To illustrate, consider the commit history of the first weeks of an incident-diabetes cohort study. It is rendered below as it might appear to an auditor reviewing how the reported hazard ratio came to be. The table is generated from synthetic data, so that we may display a realistic history without any real project's contents.

```
library(tibble)
library(dplyr)
library(knitr)
```

```
commit_history <- tibble::tibble(
  hash = c('a1b2c3d', 'e4f5a6b', 'c7d8e9f',
           'b0a1c2d', 'd3e4f5a', 'f6a7b8c',
           'a9b0c1d'),
  date = as.Date(c('2026-02-03', '2026-02-05',
                   '2026-02-09', '2026-02-12',
                   '2026-02-18', '2026-02-24',
                   '2026-03-02')),
  author = c('analyst.a', 'analyst.a', 'analyst.a',
             'analyst.b', 'analyst.a', 'analyst.b',
             'analyst.a'),
  message = c(
    'Initialize compendium and .gitignore',
    'Add cleaning script for registry extract',
    'Build analysis-ready cohort table',
    'Fit Cox model for 30-day readmission',
    'Tighten outlier threshold to 2 SD per reviewer 2',
    'Add Kaplan-Meier figure to report',
    'Correct person-time denominator'),
  files_changed = c(2L, 1L, 3L, 1L, 1L, 2L, 1L)
)

commit_history |>
  dplyr::mutate(date = format(date, '%Y-%m-%d')) |>
  knitr::kable(
    col.names = c('Hash', 'Date', 'Author',
                  'Message', 'Files'),
    align = c('l', 'l', 'l', 'l', 'r'))
```


Draft, November 13, 2026. Content may change.

Table 4.1.: A synthetic commit history for the first weeks of a cohort analysis, read as a record of decisions. Each row is one atomic, message-bearing commit; together they record not only what changed but, in the message, why. Note the reviewer-driven change to the outlier threshold, the kind of decision an auditor is entitled to trace.

Hash	Date	Author	Message	Files
a1b2c3d	2026-02-03	analyst.a	Initialize compendium and .gitignore	2
e4f5a6b	2026-02-05	analyst.a	Add cleaning script for registry extract	1
c7d8e9f	2026-02-09	analyst.a	Build analysis-ready cohort table	3
b0a1c2d	2026-02-12	analyst.b	Fit Cox model for 30-day readmission	1
d3e4f5a	2026-02-18	analyst.a	Tighten outlier threshold to 2 SD per reviewer 2	1
f6a7b8c	2026-02-24	analyst.b	Add Kaplan-Meier figure to report	2
a9b0c1d	2026-03-02	analyst.a	Correct person-time denominator	1

Two features of this history could not come from a bare list of files. The fifth commit records that the outlier threshold changed *in response to a reviewer*. The final commit records a correction to the person-time denominator, the sort of methodological fix whose provenance an auditor will want to establish.

Check 6: log or diff

An auditor asks two questions about Table 4.1: 'why did the outlier threshold change' and 'exactly which lines changed when it did.' Which git view, the log or the diff, answers each question?
The answer is on page 155.

Draft, November 13, 2026. Content may change.

4.7. What must never be committed: `.gitignore` and PHI

We come now to the gravest of the three judgments. A commit records the files git is tracking, and by default git offers to track every file in the working tree. Two categories of file must be kept out of that net. The mechanism for keeping them out is the `.gitignore` file, a plain-text list of path patterns that git will refuse to track.

The first category is the merely unwanted. These are files that *can* be committed but *should not* be, because they are regenerable or bulky and would only clutter the history. Rendered outputs (a compiled PDF, an `.html` report, the figures a script regenerates on every run) are the clearest case. Committing them couples the history to products that the code already determines. Large raw data files bloat every clone. Editor and operating-system cruft (`.Rhistory`, `.DS_Store`, `.Rproj.user/`) is pure noise. None of these is dangerous; all of them are better ignored.

The second category makes this section a matter of ethics and law. It comprises files that must *never* enter the history, because making them permanent and shareable would be a breach. In health research three kinds dominate.

- **Protected health information and identifiable data.** A file containing patient identifiers, dates of birth, medical-record numbers, free-text clinical notes, or any data from which an individual could be re-identified must never be committed. This is the first rule of version control in a health setting, and it admits no exception.
- **Credentials.** Database passwords, API tokens, the contents of a `.Renviron` holding a secret, private keys: a credential committed to a repository, and worse pushed to a remote, is a credential to be treated as compromised.
- **Restricted or embargoed data** more generally, under a data-use agreement that forbids redistribution, even when not directly identifying.

Draft, November 13, 2026. Content may change.

! Restricted data

Protected health information, identifiable data, and anything held under a data-use agreement that forbids redistribution must never go into a git history. The rule admits no exception: not even briefly on a branch that is never merged, and not even in a local repository that will ever be pushed. A commit is permanent by design and, once pushed, copied to every clone. The gravest and least intuitive fact about git is that its ordinary commands only ever *add* to the history. Suppose a file containing protected health information is committed, and the mistake is noticed and the file deleted in a later commit. The file is gone from the working tree but *remains in full in the history*. It is recoverable by anyone with a clone from the commit that introduced it. The deletion is itself just another commit layered on top; it conceals nothing. Removing the file requires rewriting the history. That rewrite changes every subsequent commit hash, cannot reach copies already made, and is error-prone precisely when it matters most. If the offending commit was ever pushed to a remote, one must assume it has been cloned and cached beyond recall. The only sound response is then to treat the exposed data as disclosed and follow the applicable incident procedure. The remedy is therefore prevention. That means a correct `.gitignore` written *before* the first commit, and the habit of reading `git status` before every commit to confirm that nothing sensitive has been staged.

When prevention has failed, the order of the response matters more than the commands, and the commands are the last step.

1. **Stop and report.** Push nothing further. If the commit reached any remote, a disclosure has occurred. Notify the data steward, privacy office, or IRB under the institution's incident procedure, because that obligation is not a technical question and does not wait for the cleanup.

2. **Rotate credentials first.** If the file held a password, token, or key, revoke or rotate it at once. A rotated secret is harmless wherever copies survive; a rewritten history is not a substitute (GitHub, n.d.-c).

3. **Rewrite the history on a fresh clone.** `git filter-repo` removes

the file from every commit on every branch and tag. Its `--sensitive-data-removal` option also fetches all refs first and keeps the remote configured so that the result can be pushed.

```
git clone https://github.com/lab/diabetes-cohort.git
cd diabetes-cohort
git filter-repo --sensitive-data-removal --invert-paths
 ↪  \
  --path analysis/data/raw_data/extract.csv
# expect no output: the file is in no remaining commit
git log --all --oneline --
 ↪  analysis/data/raw_data/extract.csv
git push --force --mirror origin
```

The force-push may require branch protection to be lifted for the purpose and restored afterwards.

> **Regulated records**
>
> In a repository that holds records under GxP or another regulated quality system, a history rewrite is a controlled change, not a cleanup. Before running it, open a deviation or incident, obtain QA approval under the organization's SOPs, document what is being removed and why, and keep whatever record of the pre-rewrite state the quality system requires. The audit-trail provision of 21 CFR Part 11 states that 'Record changes shall not obscure previously recorded information' (*21 CFR Part 11*, n.d., Section 11.10(e)). A rewrite obscures that information by design, which is why the deviation record must carry what the history no longer does. Better still, keep regulated records out of a rewritable forge in the first place, so that the question never arises.

4. **Clean what the forge holds.** On GitHub the purged commits can remain reachable through cached views and pull-request references, which the owner cannot push over. Removing them requires a request to GitHub Support. Forks are not touched by the rewrite at all, and their owners must be asked to delete them (GitHub, n.d.-c).

Draft, November 13, 2026. Content may change.

5. **Replace every clone.** Every collaborator's clone still holds the file, and a collaborator who pulls and pushes from an old clone puts it back. Each one must delete the clone and clone afresh, and any copy outside the team's reach, a backup, a CI cache, a stranger's clone, is beyond recall (GitHub, n.d.-c).

In a scratch repository, a clone made before the rewrite still lists the offending commit after the remote has been cleaned; the rewrite reaches only the copies it is run on and pushed to.

A `.gitignore` is a small file, and its content depends on the archetype of the compendium, a classification the next chapter, Chapter 5, develops in full. For now two archetypes suffice: a data-analysis compendium, whose deliverable is a set of results from data, and an R-package compendium, whose deliverable is reusable code. The reproducibility machinery is identical across archetypes; the list of files to ignore is not. A **data-analysis compendium** ignores its *derived outputs and its raw data*. These are the regenerable tables and figures, and the large or sensitive registry extract that must stay out of the history entirely. An **R-package compendium** has no `analysis/` outputs to ignore but must ignore its *build artifacts*. These are the `.Rcheck` directory, the built `.tar.gz` and binary archives, and the `src/` object files a compiled package leaves behind. The two `.gitignore` files barely overlap, and writing the right one is part of naming the archetype.

```
# .gitignore for a data-analysis compendium
# --- sensitive: never commit ---
analysis/data/raw_data/*
.Renviron
*.pem

# --- regenerable outputs ---
analysis/figures/*
analysis/tables/*
analysis/report/*.pdf
analysis/report/*.html

# --- editor and OS cruft ---
.Rhistory
```

```
.RData
.Rproj.user/
.DS_Store
```

```
# .gitignore for an R-package compendium
# --- build artifacts ---
*.tar.gz
*.Rcheck/
src/*.o
src/*.so
# generated pkgdown site; a comment must be on its own
# line, since text after a pattern is part of the pattern
docs/

# --- editor and OS cruft ---
.Rhistory
.RData
.DS_Store
```

Verification

A `.gitignore` that has never been watched blocking a real file is untested. To see it hold, create a file the pattern should catch and confirm git refuses it: `touch analysis/data/raw_data/scratch.csv && git status` should not offer to add the new file, because `analysis/data/raw_data/*` already matches it. To make it fail on purpose, comment out that one line in `.gitignore`, repeat the `touch` and `git status`, and confirm the file now appears as untracked and ready to be staged. Then restore the line before doing anything else (habit 4).

Two refinements warrant mention. A leading `!` re-includes a pattern otherwise ignored. This is how one ignores a whole data directory but keeps a small, non-sensitive example file or a data dictionary under version control (`analysis/data/raw_data/*` followed by `!analysis/data/raw_data/data_dictionary.csv`). Second, a committed-by-accident file, once added to `.gitignore`, is still tracked until explicitly untracked with `git`

Draft, November 13, 2026. Content may change.

`rm --cached`. Ignoring a file does not retroactively remove an already-committed copy, which returns us to the warning above.

This book's own repository applies the rule to its running cohort. The extract `analysis/data/raw_data/cohort_raw.csv` is synthetic and describes no real person, yet it is not committed. What is committed is the script that writes it, `analysis/scripts/simulate_cohort.R`, which regenerates the extract from stated parameters; a reader with a fresh clone runs it once before the examples. The `.gitignore` excludes the whole `raw_data` directory rather than the one filename, so that a second extract saved there under any name is excluded too. The check below confirms that the generated extract carries identifying-looking fields, that the generator is present, and that the directory pattern is in force.

```
raw <- load_cohort('../data/raw_data/cohort_raw.csv')
nrow(raw)
#> [1] 338
intersect(c('mrn', 'patient_name', 'dob'), names(raw))
#> [1] "mrn"          "patient_name" "dob"

gitignore <- readLines('../../.gitignore')
repo_state <- c(
  generator_present =
   ↪  file.exists('../scripts/simulate_cohort.R'),
  raw_dir_ignored = 'analysis/data/raw_data/*' %in%
   ↪  gitignore
)
repo_state
#> generator_present   raw_dir_ignored
#>              TRUE              TRUE
```

The generated extract carries 3 of the three columns this section warns against, by name. The synthetic cohort invents an identifier, a name, and a date of birth so that the risk appears in a real file instead of only in prose. The generator is present, and the pattern `analysis/data/raw_data/*` is in the `.gitignore`, which is the arrangement a real project should copy: the code that produces or fetches the extract under version control, the extract itself never.

A `.gitignore` guards paths, however, and nothing else. The tests in `inst/tinytest/` need a copy of the cohort, and an earlier version of this repository met that need by copying the extract to `inst/testdata/cohort_raw.csv` and committing it. No ignore pattern covered that path, so the identifier columns entered the history by a second route while the first stayed closed. The fixture now at that path is derived from the synthetic extract. It keeps the rows and the duplicate structure the tests need, drops the name and date-of-birth columns, and replaces each record number with a pseudonymous `ID-0001`-style id.

```
fixture <-
 ↪  load_cohort('../../inst/testdata/cohort_raw.csv')
intersect(c('mrn', 'patient_name', 'dob'), names(fixture))
#> [1] "mrn"
all(grepl('^ID-[0-9]{4}$', fixture$mrn))
#> [1] TRUE
```

Replacing the file only added a commit, and the earlier copy is still in the history. Two commands find it. The first lists every commit, on any branch, that touched the path; the second prints the header of the file as the oldest of those commits recorded it.

```
git log --all --oneline -- inst/testdata/cohort_raw.csv
git show <oldest-hash>:inst/testdata/cohort_raw.csv | head
 ↪  -1
```

```
#>    commits_touching_path old_copy_has_identifiers
#>                        2                         1
#>      old_copy_same_bytes
#>                        1
```

Run on this repository when the page was rendered, the log listed 2 commits touching the path. The oldest, 0bec92e, still holds a copy whose header names patient_name and dob, and its bytes are identical to the generated extract. Anyone holding a clone, made at any time since, can read it. The data are synthetic, so nothing real has leaked, but the pattern is the one by

Draft, November 13, 2026. Content may change.

which real data leak: an extract protected at one path and copied, for a convenience such as a test, to another. Removing the copy from the history would take the rewrite described above, with every cost listed there; we keep it as this worked example. Deriving the new fixture from the extract is acceptable here only because the extract is synthetic. Rows of real data with the name columns dropped still carry dates, ages, and measurements that can re-identify, so a fixture for a real project is generated or built from the data dictionary, and a commit that adds a data file under any path warrants the question of where its rows came from.

> Check 7: the same file, two archetypes
>
> A `report.pdf` sits at the top of one analyst's ignore list and is deliberately committed by another. A `data_dictionary.csv` describing a restricted registry is committed, while the registry extract it describes is ignored. Explain both choices.
> *The answer is on page 155.*

4.8. Branches, lightly

With the history and its exclusions in place, we turn to its shape. For a solo analyst working linearly, a single line of commits on the `main` branch is often the whole story, and it is honest to say so rather than to impose ceremony. But two situations make branches worth the small investment of learning them. The first is *speculative work*: trying an alternative model specification, or a reviewer's proposed reanalysis, without disturbing the `main` line that produced the current results. The second is *collaboration*, treated in the next section. There each contributor works on a branch of their own, and the branches are merged only when ready.

A branch is created and switched to in one command, and merged back with another.

```
# create and switch to a branch for a reanalysis
git switch -c sensitivity-analysis

# ... make commits on the branch ...
```

```
# return to main and merge the branch in
git switch main
git merge sensitivity-analysis
```

The mental model is the one given above: a branch is a movable name for a line of development, and `main` is merely the name given to the first branch, here by `git init -b main`. Nothing about branching changes the commit model; it only lets two lines of commits coexist and later rejoin. We keep the treatment light here. We shall return to the collaborative use of branches, which is where they earn their keep, when we reach pull requests.

Figure 4.2 draws the shape. The branch does not copy the history. It names a new commit whose parent is a commit on `main`, and the merge names a commit with two parents. Everything before the branch point is shared, and no hash anywhere in the diagram changes when the branch is created or merged.

```
library(ggplot2)
library(tibble)

nodes <- tibble(
  id    = c('A', 'B', 'C', 'M', 'D', 'S1', 'S2'),
  x     = c(1, 2, 3, 4.6, 5.7, 3.1, 4.0),
  y     = c(0, 0, 0, 0, 0, 0.8, 0.8),
  shade = c(rep('main', 5), 'branch', 'branch')
)

edges <- tibble(
  x    = c(1, 2, 3, 4.6, 2, 3.1, 4.0),
  xend = c(2, 3, 4.6, 5.7, 3.1, 4.0, 4.6),
  y    = c(0, 0, 0, 0, 0, 0.8, 0.8),
  yend = c(0, 0, 0, 0, 0.8, 0.8, 0)
)

ggplot() +
  geom_segment(data = edges,
```


Draft, November 13, 2026. Content may change.

```
            aes(x = x, y = y, xend = xend, yend = yend),
            color = 'grey45', linewidth = 0.45) +
geom_point(data = nodes, aes(x = x, y = y, fill = shade),
           shape = 21, size = 9, stroke = 0.5,
           color = 'grey30', show.legend = FALSE) +
geom_text(data = nodes, aes(x = x, y = y, label = id),
          size = 3, fontface = 'bold') +
scale_fill_manual(values = c(main = '#c3ddda',
                             branch = '#f0e2c4')) +
annotate('text', x = 0.55, y = 0, label = 'main',
         hjust = 1, fontface = 'bold', size = 3.2) +
annotate('text', x = 2.55, y = 0.8, label = 'branch',
         hjust = 1, fontface = 'bold', size = 3.2) +
annotate('text', x = 4.6, y = -0.52,
         label = 'M: two parents', size = 2.9,
         color = 'grey25') +
annotate('text', x = 2.0, y = -0.52,
         label = 'branch point', size = 2.9,
         color = 'grey25') +
coord_cartesian(xlim = c(-0.1, 6.1), ylim = c(-0.8, 1.1))
 ↪  +
theme_void()
```


Draft, November 13, 2026. Content may change.

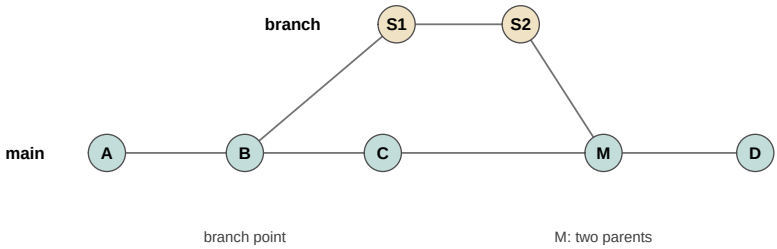


Figure 4.2.: A branch and its merge, drawn as the commit graph git actually stores. The lower line is `main`; the upper line is a sensitivity-analysis branch created at commit B and merged back at M. Commits A through C exist once and are shared by both lines: the branch adds commits rather than copying them. The merge commit M is the only commit in the diagram with two parents, which is what records that the two lines rejoined and when.

Check 8: what the merge changes

After creating and merging the `sensitivity-analysis` branch of the example above, do any of the commit hashes on `main` from before the branch was created change?
The answer is on page 156.

4.9. Undoing things, and merge conflicts

Two kinds of trouble are common enough to deserve their own section: a change that needs to be undone, and a merge that git cannot finish by itself.

Undoing. Git has three distinct commands for undoing work, and the right one depends on where the unwanted change lives.

`git restore <file>` discards *unstaged* edits in the working tree, putting the file back to its last committed (or staged) content. It never touches history; there is no commit to undo, only a draft to throw away.

Draft, November 13, 2026. Content may change.

```
# an edit no one has staged or committed
echo 'threshold <- 99' > analysis.R
git restore analysis.R
# analysis.R is back to its last committed content
```

`git reset` moves what `HEAD` points to. That is, it moves the branch itself to an earlier (or different) commit. The three flags differ only in how much else they touch along the way:

- `--soft` moves `HEAD` and leaves everything else alone: the undone commit's changes reappear staged, ready to be recommitted differently.
- `--mixed`, the default, additionally unstages those changes, so they sit in the working tree, edited but not staged.
- `--hard` additionally overwrites the working tree, discarding the changes outright.

```
# move HEAD back one commit; --mixed is the default
git reset HEAD~1
# the undone commit's edits are now unstaged, not gone

git reset --hard HEAD~1
# the undone commit's edits are discarded entirely
```

A reset, like the `amend` and `rebase` discussed earlier in this chapter, is a rewrite of what the branch name points to, not an addition to history. The same caution applies. A reset is harmless on commits that exist only in the analyst's own clone. It is a serious act, needing the collaborators' agreement and a force-push, once those commits have been pushed or pulled elsewhere. A local reset that moves past *commits* can usually be undone. The discarded commits are not immediately deleted, and `git reflog` lists where `HEAD` pointed before the reset, so `git reset --hard <that-hash>` restores them. The reflog records only commits, however. Edits that were never committed are not in it, and `--hard` overwrites them in the working tree. Unstaged edits are then gone permanently; staged ones survive only as unnamed objects that `git fsck --lost-found` can sometimes recover, file by file and without their names. Before any `--hard`, therefore, read `git`

Draft, November 13, 2026. Content may change.

`status`, and commit or `git stash` whatever uncommitted work should survive.

`git revert <commit>` takes the opposite approach. Instead of moving `HEAD` backward, it makes a *new* commit whose changes are the exact opposite of the one named, applied on top of the current history. The reverted commit is left untouched, in place, for anyone to read later. Because revert only ever adds a commit, it is the one of the three that is safe on a branch other people have already pulled; reset and amend are not.

```
git revert HEAD
# creates a new commit, e.g. "Revert 'Add note line'",
# that undoes the previous commit's changes; the
# reverted commit itself still exists in the history
```

The choice can be stated briefly. For an edit that is not yet staged, use `restore`. For a commit that exists only on a private branch, `reset` is simplest. For a commit that others may already have, use `revert` and let history show both the mistake and its correction.

Merge conflicts. A merge like the one in Figure 4.2 goes through cleanly whenever the two branches changed different lines. When both branches change the *same* lines of the *same* file, git has no way to choose between them, and the merge stops partway, leaving the decision to the analyst.

```
git switch main
git merge sensitivity-analysis
# Auto-merging analysis.R
# CONFLICT (content): Merge conflict in analysis.R
# Automatic merge failed; fix conflicts and then
# commit the result.
```

`git status` reports the file as having 'unmerged paths', and the file itself now contains both versions, separated by conflict markers that git inserts directly into the text:

```
<<<<<<< HEAD
```


Draft, November 13, 2026. Content may change.

```
threshold <- 4
=======
threshold <- 5
>>>>>>> sensitivity-analysis
```

Everything between `<<<<<<< HEAD` and the `=======` divider is the current branch's version; everything between the divider and the final marker is the version being merged in. Resolving the conflict means editing the file by hand down to the single version that should survive. That may be one side, the other, or a combination of both. The analyst then deletes the markers themselves; nothing removes them automatically. Once the file reads the way it should, the usual two-step cycle finishes the merge. One runs `git add` on the resolved file to tell git the conflict is settled, then `git commit` to record the merge. (`git merge --abort` backs out of the whole attempt instead, if the conflict turns out to need more thought than a mid-merge edit allows.)

```
# edit analysis.R to resolve the conflict, then:
git add analysis.R
git commit -m "Merge sensitivity-analysis: keep threshold
 ↪  4"
```

A conflict is not git malfunctioning; it is git correctly refusing to guess at a scientific decision, such as which of two threshold values the analysis should use, that only the analyst can make.

Check 9: which command for which undo

A collaborator has three problems: an edit in `analysis.R` that was never staged and should be thrown away; a commit made on a private branch, pushed nowhere, that turns out to be wrong; and a commit on `main` that was already pulled by two other collaborators and turns out to be wrong. Which command fits each case?
The answer is on page 156.

Draft, November 13, 2026. Content may change.

4.10. The remote: GitHub, GitLab, and collaboration

Everything so far is *local*. The repository, its history, and its branches all live in the `.git` directory on one machine, and if that machine is lost, so is the work. A **remote** is a copy of the repository hosted elsewhere, on GitHub, on GitLab, or on an institution's own server. Connecting a local repository to a remote serves three distinct purposes that are worth separating.

The first is **backup**. A remote holds a complete copy of the history, so that a lost laptop costs at most the commits made since the last push. The second is **distribution**. A public remote is the mechanism by which a compendium becomes locatable to the world. It is a URL a reviewer can be given, a repository a reader can clone, the concrete form of the L0 rung of the ladder. The third is **collaboration**, the purpose that shapes the rest of this section.

The commands that connect to and synchronize with a remote are few. One names the remote once, then pushes local commits to it and pulls others' commits from it.

```
# connect the local repo to a remote named origin
git remote add origin
 ↪  https://github.com/lab/diabetes-cohort.git

# push the main branch and remember the link
git push -u origin main

# later: send new local commits up
git push

# and bring collaborators' commits down
git pull
```

Consider a team working across sites, a clinical trial with several participating centers, a multi-institution consortium. Such a team works in one of two models on GitHub and GitLab. In the **shared-repository** model, every contributor has write access to one repository, works on a branch of it,

Draft, November 13, 2026. Content may change.

and the `main` branch is protected so that nothing reaches it without review. This is the usual arrangement for a private trial repository, where forking is often disabled. In the **fork-and-pull-request** model, a contributor without write access *forks* the canonical repository, making a personal copy on the remote, and works on a branch of that fork. In either model the contributor opens a **pull request** (GitLab calls it a *merge request*, the same thing under another name) proposing that the branch be merged into `main`. The pull request is a unit of review. The maintainers see exactly the diff being proposed, discuss it inline, request changes, and merge it only when satisfied. In a multi-site analysis, the pull request is a documented, attributable, reviewable gate through which every change to the shared analysis passes. Its record, who proposed what, who reviewed it, what was said, and when it was merged, is held on the forge's servers rather than asserted by a client. It therefore complements the commit history with a record of the *review* of each change that is harder to backdate. That record is review evidence, not by itself a regulated signature. An approval on a forge does not, by default, carry the printed name, the date and time, and the meaning of the signature that 21 CFR 11.50(a) requires of a signed electronic record (*21 CFR Part 11*, n.d.). Whether it counts as a controlled approval depends on the organization's validated configuration of the forge and on its SOPs.

The same gate serves when the contributor is a coding agent. The agent works on a branch of its own and opens a pull request; the analyst reads the diff, reviews it, and alone decides whether to merge. The agent's account of its change is a description to check against the diff, not a substitute for reading it (Section 2.10.1).

The last piece of the remote's vocabulary is the pair **tags and releases**, and they matter to reproducibility specifically. A tag is a permanent, human-readable name for one commit. `git tag -a v1.0-submission -m "State at journal submission"` marks, forever and unambiguously, the exact state of the analysis at the moment a manuscript was submitted. A *release* on GitHub or GitLab is a tag dressed with notes and downloadable archives. The reproducibility payoff is that a reviewer, or the analyst a year later, can recover the precise state that produced the submitted numbers by naming the tag. There is no need to guess which commit was current on the submission date. In Chapter 13 we shall see the same tagged state deposited to a public archive with a citable identifier. The tag is the local

Draft, November 13, 2026. Content may change.

anchor to which that archival identity attaches.

> Check 10: what does the pull request add?
>
> A single analyst, working alone, keeps a clean commit history with careful messages. A reviewer asks why, if the history already records each decision, a multi-site trial should bother with the additional machinery of forks and pull requests. What does the pull request record that the commit history does not?
> *The answer is on page 156.*

4.11. What git does not do

Having made version control the foundation, we must be exact about the boundary of what it provides. The whole discipline of this book is matching a claim to what has actually been secured. Git versions *text and tracks its history*. It records every state of the files under it and the reason each committer gave for each change. That is the entirety of what it does, and it is the L0 rung of the ladder. An analysis under version control is locatable, findable and readable by a second person, and no more.

In particular, git does not pin the *package versions* an analysis uses. A repository cloned a year later contains the same `analysis.R`, but running it installs whatever package versions are current. These may not be the versions the analyst used, so the code that ran may no longer run, or may run and return a different number. Git also does not pin the *computational environment*. The R version, the operating system, the system libraries a package links against are nowhere in the history, and a clone is at the mercy of whatever the next machine provides. These are exactly the layers of the determinant stack that Chapter 3 enumerated as unpinned at L0. Closing them is the work of the two chapters that follow. The lockfile of Chapter 6 pins the package layer to reach L1, and the container of Chapter 7 pins the environment layer to reach L2. Version control is necessary for all of this, since the lockfile and the Dockerfile are themselves files whose history git tracks. It is sufficient for none of it.

Draft, November 13, 2026. Content may change.

Check 11: history without a lockfile

A repository has a full, well-written commit history, a tag marking the submitted manuscript, and is hosted on GitHub. Has it reached L1? *The answer is on page 156.*

! The honest level

This chapter's own practice reaches L0, locatable, by placing the repository under version control and pushing it to a remote. The cohort extract is not part of that history; its generator is, and the extract is regenerated from it. The practice leaves every layer beneath the analysis code, the packages, the environment, and the rest of the determinant stack, exposed and unpinned.

4.12. Worked example

To make the chapter concrete, we take the incident-diabetes cohort of the running example from an untracked folder to a tagged, remotely hosted repository. We stay at the level of the commands and the judgment behind them. The shell sessions below are not executed when the book renders. The outputs shown with `#>` were captured once, on 2026-11-08, from a scratch copy of the example with a synthetic identity, and are abridged.

The analyst begins with a folder of scripts and a registry extract. The very first act, before a single commit, is to write the `.gitignore`, because the extract contains protected health information that must never enter the history.

```
cd diabetes-cohort
git init -b main
# write .gitignore FIRST, ignoring raw_data and credentials
git status --untracked-files=all   # is the extract listed?
```

The flag matters. Plain `git status` collapses a directory that holds no tracked files into one line, here `analysis/`, so it cannot show whether the

Draft, November 13, 2026. Content may change.

extract inside is ignored. Run on a scratch copy of this example, the full listing names the extract before the `.gitignore` is written and omits it after, and `git check-ignore -v` names the rule responsible.

```
# output abridged, from git 2.56.0
# before .gitignore
git status --untracked-files=all
#> Untracked files:
#>   DESCRIPTION
#>   analysis/data/raw_data/extract.csv
#>   analysis/scripts/01-clean.R

# after .gitignore
git status --untracked-files=all
#> Untracked files:
#>   .gitignore
#>   DESCRIPTION
#>   analysis/scripts/01-clean.R

git check-ignore -v analysis/data/raw_data/extract.csv
#> .gitignore:1:analysis/data/raw_data/
 ↪  analysis/data/raw_data/extract.csv
```

Only once `git status` confirms that the sensitive extract is ignored does the analyst make the first commit, the one that lifts the folder to L0, locatable. Its output names the branch, marks the commit as the root of the history, and lists the files it records.

```
git add .gitignore DESCRIPTION
git commit -m "Initialize compendium and .gitignore"
# output abridged, from git 2.56.0
#> [main (root-commit) 936e050] Initialize compendium and
 ↪  .gitignore
#>  2 files changed, 4 insertions(+)
#>  create mode 100644 .gitignore
#>  create mode 100644 DESCRIPTION
```


Draft, November 13, 2026. Content may change.

A commit that stops instead with 'Please tell me who you are' means git has no identity on this machine. The Git preflight sets it, and When git refuses lists this and the other refusals a first push can meet.

Work proceeds as a sequence of small, message-bearing commits: the cleaning script, the derived cohort table, the Cox model. Each is its own commit with its own reason, exactly the history tabulated in Table 4.1. When a reviewer later asks why the outlier threshold changed, the answer is one `git log` away.

To try a reviewer's proposed reanalysis without disturbing the results already reported, the analyst branches.

```
git switch -c reviewer-sensitivity
# ... reanalysis commits ...
git switch main            # main is untouched and still
 ↪  runs
```

The work is backed up and made locatable by connecting a remote and pushing.

```
git remote add origin
 ↪  https://github.com/lab/diabetes-cohort.git
git push -u origin main
# output abridged, from git 2.56.0 (local stand-in remote)
#>  * [new branch]      main -> main
#> branch 'main' set up to track 'origin/main'.
git status
#> On branch main
#> Your branch is up to date with 'origin/main'.
```

The last line of the push is the one to look for: the local `main` now tracks the remote's, so a later bare `git push` or `git pull` knows where to go. The scratch run pushed to a local bare repository rather than to a forge, so its output omits the forge's own progress lines. A push that is refused instead is diagnosed from When git refuses.

A collaborator at a second site forks the repository, adds a site-specific adjustment on a branch of their fork, and opens a pull request. The

Draft, November 13, 2026. Content may change.

analyst reviews the diff inline, requests one change, and merges it, leaving a reviewable record of the exchange. Finally, at journal submission, the analyst marks the exact state.

```
git tag -a v1.0-submission -m "State at journal submission"
git push origin v1.0-submission
```

The folder is now a version-controlled, remotely hosted, tagged repository. It is firmly at L0, with a clean history, a reviewed collaboration record, and a permanent anchor to the submitted state. It is *not* yet L1 or L2, because nothing has pinned the packages or the environment, and the honest reproducibility statement says exactly that. The lockfile of Chapter 6 is the next commit.

> Check 12: what the worked example still lacks
>
> At the point the worked example ends, tagged, pushed, and with a reviewed pull request merged, what single artifact would need to be added for the compendium to reach L1, and is adding it itself a version-control operation?
> *The answer is on page 156.*

4.13. Collaborating with an LLM

A language model is a fluent assistant for git, which is a domain rich in memorized command recipes. Precisely that fluency is the hazard. Check every command it proposes for operations beyond what was asked, irreversible ones above all; the model does not know which of the analyst's files are sensitive. A coding agent that works in the repository can read those files, run git itself, and push. It is therefore run only in projects whose data it is allowed to see, and never with credentials that let it push to a remote unreviewed.

Draft, November 13, 2026. Content may change.

! Restricted data

One rule comes first (habit 3). Never paste or upload protected health information, identifiable records, or data held under a data use agreement, nor outputs derived from them, into a model or a coding agent unless three conditions all hold. The institution has approved that endpoint for that class of data. The IRB protocol and the consent permit the processing. The data use agreement, or the provider's terms, permits it. If any of the three is unclear, the answer is no.

The third condition is the one most often missed. Institutional approval of a tool does not override the terms under which a data provider released the data. NIH, for example, treats sharing controlled-access data through prompts with public generative AI tools, which it glosses as third-party tools, as a breach of the data use certification. It treats a model trained on such data as a data derivative, to be shared only with other approved users and not kept after the project closes (National Institutes of Health, 2025a).

Describe the structure instead: the column names and types, or a small synthetic example generated from the data dictionary, never edited from real rows. A coding agent sees whatever files are in its working tree, so run it in a checkout or container where the real data is not mounted.

An agent also acts, so confine it to a branch and keep merges, pushes, and deposits for yourself (Section 2.10.1).

For git, the structure a prompt describes is the repository's: its paths and file types, not the contents of the files. The three triples below mark the division of labor.

Prompt. 'I accidentally committed a file of patient data three commits ago. Give me the git commands to remove it.'

Watch for. A history-rewriting recipe (`git filter-repo`, or the older `filter-branch`, or a `reset`) presented as a clean fix. The model cannot know whether the offending commit was ever pushed to a remote. That is the fact that determines whether removal is even possible, so check whether the answer raises the breach question at all.

Verification. Treat the exposure, not the git command, as the primary

Draft, November 13, 2026. Content may change.

problem. Establish first whether the commit was pushed. If it was, assume the data has been cloned and cached and follow your institution's incident procedure, because no git command can recall it. Only then apply a rewriting tool, on a fresh clone and in the order the five-step procedure earlier in this chapter gives, and in a repository of regulated records only as the approved, documented change its third step requires. Check that the recipe covers the forge's caches, forks, and every collaborator's clone. A coding agent asked the same question does not stop at a recipe: it can run the rewrite and the force-push itself. The breach assessment, the decision to rewrite, and the force-push stay with the analyst. At most the agent prepares the commands on a fresh clone, for the analyst to read and run (Section 2.10.1).

Prompt. 'Write a `.gitignore` for my R health-data analysis project.'

Watch for. A generic R `.gitignore` (the standard `.Rhistory`, `.RData`, `.Rproj.user/` list) that omits the one category that matters most, the raw data and credentials specific to *your own* project. A chat model has not seen your directory. A coding agent may have listed it, but a file's name does not say whether it holds PHI. An agent that opened the files to find out has already read the data, which is why it belongs only in projects it is cleared to see.

Verification. Take the generic list as a starting point, then add, by hand, the explicit paths of every raw-data directory, every `.Renviron` or credential file, and every restricted dataset. Confirm with `git status` that each is untracked before the first commit. The generic patterns are safe to accept; the project-specific sensitive paths are yours to supply.

A better prompt. 'Below is the directory tree of my R analysis compendium, file names only, with no file opened. Write a `.gitignore` that excludes `analysis/data/raw_data/`, `.Renviron` and any other credential file, and regenerable outputs, but keeps the code, `renv.lock`, and the synthetic cohort tracked. Label each line generic or project-specific, and list the commands that would show me each sensitive path is ignored.'

Prompt. 'Write good commit messages for these changes.'

Watch for. Messages that fluently describe *what* changed, which the model can read from the diff, but invent or omit the *why*. The why is the

Draft, November 13, 2026. Content may change.

scientifically load-bearing part, and the model cannot know it. A plausible-sounding ‘Update outlier threshold’ hides the reviewer request that the record must carry.

Verification. Supply the reason yourself. Let the model draft the imperative summary line from the diff, then edit in the motivation, the reviewer comment, the corrected denominator, the protocol amendment, that turns a description into an accountable record.

4.14. Exercises

1. Take a folder of your own currently versioned by the `FINAL`-file method, count the manual snapshot files it contains, and for two of them state what changed between them and why. Note which of those two questions you can still answer, and use the difficulty as a measure of what a commit history would have preserved.

2. Initialize a git repository in a scratch directory, make three commits that each change one file, and then use `git log --oneline` and `git show` to recover, for the middle commit, both what changed and the message you wrote. Confirm that the commit hashes are unchanged by a later commit.

3. Write a `.gitignore` for a data-analysis compendium and a second for an R-package compendium, and list the patterns that appear in one but not the other. Explain each difference in terms of what the archetype produces.

4. Explain, to a colleague who has just deleted a PHI-containing file in a new commit and believes the problem solved, why the file is still in the history, what determines whether it can be removed, and what to do if the offending commit was already pushed to GitHub.

5. Construct the sequence of commands by which a collaborator at a second trial site would contribute a change through a fork and a pull request, and state, for each step, what part of the project’s record it adds that a direct push to `main` would not.

Draft, November 13, 2026. Content may change.

6. An analyst pushes a version-controlled compendium to GitHub and writes in the methods section that the analysis is 'fully reproducible'. State the rung the analysis has actually reached, name the two layers of the determinant stack that remain unpinned, and rewrite the sentence so that it claims exactly what version control secures and no more.

7. In a fresh clone of this book's repository, run `Rscript analysis/scripts/simulate_cohort.R` from the top directory, then `raw <- load_cohort('analysis/data/raw_data/cohort_raw.csv')`. Report `nrow(raw)` and the three columns among `names(raw)` that would, in a real project, require the extract to sit behind the `.gitignore` pattern `analysis/data/raw_data/*`, and use `git check-ignore -v analysis/data/raw_data/cohort_raw.csv` to name the line that excludes it. Then run `git log --all --oneline -- inst/testdata/cohort_raw.csv` and state what the oldest commit it lists shows about whether that pattern kept the identifier columns out of the history.

8. This exercise returns to the determinant stack and the L0-to-L3 ladder of Chapter 3 (Chapter 3). A colleague deletes the `.git` directory of a well-tested, fully containerized analysis and runs `git init` again to 'start clean' before pushing, so the repository now hosted on GitHub shows a single commit from yesterday. Using what this chapter and Chapter 3 together establish, state the rung the repository occupies today, and explain what the colleague's own prior discipline, the lockfile and the container, can and cannot restore of what the deleted history held.

4.15. Solutions

Attempt an exercise before expanding its solution. The first two ask you to work in your own directories, so what you produce will vary; the boxes say what a good answer establishes.

Draft, November 13, 2026. Content may change.

Exercise 1

No single correct answer. What a good answer establishes is the asymmetry between the two questions, because the asymmetry is the argument for version control.

Almost every reader can answer *what changed* between two snapshot files, because the files are still there and a diff tool will show it. Almost no reader can answer *why*, because the reason was never written down anywhere. It lived in the analyst's head, or in an email, or in a conversation after a meeting, and it is gone. Readers commonly find they can reconstruct the what for files a year old and cannot reconstruct the why for files a month old.

That gap is precisely what a commit history preserves. The diff is recoverable from the files themselves, so it is the cheaper half; the message is not recoverable from anything, so it is the half that decays. This is why the chapter insists that a commit is an atomic unit of one idea with a message attached. The message is the only part of the record that cannot be reconstructed later.

If you find that you *can* answer why, because you named the files carefully, you have demonstrated the `FINAL` method working at its best. Notice, then, how much of the reasoning fits in a filename.

Exercise 2

The mechanics are straightforward, so the substance is the last clause.

Draft, November 13, 2026. Content may change.

```
git init
# edit file_a.txt
git add file_a.txt && git commit -m 'Add the cohort
 ↪  filter'
# edit file_b.txt
git add file_b.txt && git commit -m 'Raise the outlier
 ↪  threshold'
# edit file_c.txt
git add file_c.txt && git commit -m 'Add the summary
 ↪  table'
git log --oneline
git show <middle-hash>
```

`git log --oneline` lists the three commits newest first, each with an abbreviated hash and its subject line. `git show` on the middle hash prints both halves of the record: the message you wrote, and the diff of what changed.

On the hashes: recording the middle commit's hash before making the third commit and comparing afterward shows it unchanged. A commit hash is computed over the commit's content *and* its parent's hash, so it fixes the entire history leading to it. Adding a commit on top cannot alter it, because the new commit points backward and nothing points forward. That is the property that makes the history tamper-evident to anyone who recorded a hash.

The honest qualification, which a careful reader may raise: hashes are stable under *adding* history, not under *rewriting* it. An `amend` or a `rebase` produces new commits with new hashes, and the old ones become unreferenced. A rewrite is detectable only by someone who kept an old hash, and the author and dates inside each commit were set by the committer's own client in the first place. The record is trustworthy for a history that is appended to and whose hashes are held elsewhere, in tags, in collaborators' clones, on a protected forge. That is why rewriting shared history is treated as a serious act.

Draft, November 13, 2026. Content may change.

Exercise 3

Your own files will differ in detail. Taking the two samples in this chapter as the model answer, they barely overlap, and the difference follows from what each archetype produces.

Common to both: the editor and operating-system cruft, `.Rhistory`, `.RData`, and `.DS_Store`.

In the **data-analysis** file but not the package one: the raw-data extract, `analysis/data/raw_data/*`, which carries the protected information; the regenerable outputs, `analysis/figures/*`, `analysis/tables/*`, and the rendered `*.pdf` and `*.html` reports, since the report is a product the code regenerates; and `.Renviron`, `*.pem`, and `.Rproj.user/`.

In the **R-package** file but not the analysis one: the build artifacts, `*.tar.gz` and `*.Rcheck/`; the compiled object files `src/*.o` and `src/*.so`, which exist only if the package carries compiled code and are rebuilt on every install; and `docs/`, the generated pkgdown site.

The last three analysis-only lines are differences of the abridged samples, not of the archetypes. Credentials and editor state can appear in any project, so a package repository should ignore `.Renviron`, `*.pem`, and `.Rproj.user/` as well. A good answer says so.

The explanation to give for each remaining difference has the same shape. An archetype ignores what it *generates* and commits what it *is*. A data-analysis compendium generates a report and so ignores it. A package generates a tarball, check output, compiled objects, and a website, and so ignores those. Neither ignores its own source.

The one line that is not a matter of archetype is the data carrying protected health information. It is ignored in every archetype, always, and is the subject of the next exercise.

Exercise 4

The colleague's mental model is that a repository holds the current state of the files. It holds their whole history, and that is the point of it.

Why the file is still there. Deleting a file in a new commit records the deletion as a change. The commit that added the file is untouched,

and the file's contents are retrievable from it by anyone who clones the repository. The deletion is a new fact appended to the record, not an erasure of the old one.

What determines whether it can be removed. Whether the history has been shared. Removing content means rewriting every commit from the offending one forward, with `git filter-repo` or an equivalent, which produces new hashes for all of them. If the offending commit exists only in the local repository, this is a local operation and costs nothing but care. If it has been pushed, every clone and fork holds the old objects, and rewriting the local copy does not touch them.

What to do if it was already pushed to GitHub. Treat the data as disclosed and act in that order. Notify whoever is responsible for the data, an IRB or a privacy office, because a disclosure has occurred and the notification obligation is not a technical question. Rotate anything rotatable. Then rewrite the history and force-push, and ask GitHub Support to purge cached views and any dangling objects. Require every collaborator to re-clone rather than pull, since a pull onto a rewritten history reintroduces the old objects. Note the ordering: the cleanup is the last step and the least important one, because it does not undo the exposure.

This is why the `.gitignore` for protected data is written before the first commit rather than after the first mistake.

Exercise 5

A serviceable sequence, with what each step adds to the record.

Draft, November 13, 2026. Content may change.

```
# 1. Fork the canonical repository on the forge.
git clone https://github.com/<site2>/<study>.git
cd <study>
git remote add upstream
 ↪  https://github.com/<coord>/<study>.git

# 2. Branch from an up-to-date main.
git fetch upstream
git switch -c fix/outlier-threshold upstream/main

# 3. Commit the change.
git add R/clean_cohort.R
git commit -m 'Raise outlier threshold to 4 SD per SAP
 ↪  amendment 2'

# 4. Push to the fork and open a pull request.
git push origin fix/outlier-threshold
```

What each step records that a direct push would not. The **fork** lets a contributor without write access to the canonical analysis propose a change, which then cannot enter without someone else's action. (A fork does not show that the contributor lacked write access; anyone may fork. On a private trial repository with forking disabled, a branch of the shared repository with a protected `main` plays the same role.) The **branch** separates the proposal from the accepted state, so the history of `main` contains only what was agreed. The **commit** is the same in either workflow and records what and why. The **pull request** is the artifact that carries everything a direct push destroys. It holds the proposed diff before it was accepted, the discussion, any requested changes and the revisions answering them, the identity of the approver, and the timestamp of the approval. The **merge** records that the approval preceded the entry into `main`.

The compressed version of the answer: the commit history answers 'what changed and why'; the pull request answers 'who checked it, and did they agree'. A direct push to `main` answers only the first, and in a multi-site trial the second question has auditors too. In a regulated trial, the approval is review evidence rather than, by itself,

Draft, November 13, 2026. Content may change.

a signed record carrying the printed name, the date and time, and the meaning that 21 CFR 11.50(a) requires; whether it serves as one depends on the sponsor's validated configuration and SOPs. The analysis change behind the commit is itself approved through the sponsor's own process, here an amendment to the statistical analysis plan, not through the pull request.

Exercise 6

The rung reached is L0. Version control makes the analysis locatable and gives it a history. It pins the analysis code, which is a real achievement and the precondition for everything above it. However, the ladder grades how far *down* the determinant stack a result is locked, and the code sits at the top of that stack.

Two layers that remain unpinned. The first is the package versions, since nothing in a git repository records which version of `dplyr` produced the numbers. The second is the environment beneath them, the R runtime, the operating system, and the system libraries. You may equally name the numerical libraries or the session configuration and be right, since the repository touches neither. The common wrong answer is to name the data, which version control *can* pin for a small committed dataset, though usually does not for a large or protected one.

A rewritten sentence. 'The analysis code is version controlled and deposited at [location], so the exact code that produced these results can be retrieved. The software environment was not recorded; re-execution may therefore require package versions differing from those used here, and we have not verified that the results regenerate.' The three clauses map onto the three claims: what is secured, what is not, and what has not been checked. The last clause is the one authors most often omit, and it is the difference between claiming L0 honestly and claiming L3 by implication.

Exercise 7

The solution for this exercise is in the instructor guide, which is distributed separately to course instructors. Self-study readers who

Draft, November 13, 2026. Content may change.

want to check their answer can compare it with the executed output of the cohort checks in the `.gitignore` section of this chapter: the row count and identifier columns of the generated extract, the directory pattern, and the history of `inst/testdata/cohort_raw.csv`. The point of the last step is that `git check-ignore` succeeds on the extract while the oldest copy of the fixture, on the render of this page, still carries the identifier columns: the pattern held at its own path, and the copy entered the history by another.

Exercise 8

The rung is L2, unchanged by the deletion. The repository that is hosted on GitHub today is still locatable, so L0 holds and a reviewer can find and read it. This is so even though its visible history begins with yesterday's single commit. Nothing about `git init` run fresh touches the lockfile or the container. Those files, if they still sit in the working tree, are tracked again by the new history and hold the analysis at L1 and L2 exactly as before. Package pinning and environment pinning are properties of *files*, not of the commit chain that records them. If the analysis had reached L3, its baseline test is a file too and survives, but the CI runs that showed it passing were attached to commits that no longer exist. L3 is reached only by verification, so it must be shown again by a fresh run of that test after a clean rebuild.

What is lost, and what is not. The lockfile and the container restore the analysis's position on the capture ladder, L1 and L2. `renv.lock` and the Dockerfile are ordinary files that a fresh `git init` tracks exactly as well as the old history did. What they cannot restore is the history itself: the sequence of decisions, the reviewer-driven changes, the dated and attributed record Table 4.1 exemplifies, and any tag that marked a past submission. That record is not a determinant of the computational result at all, so no rung of the L0 to L3 ladder grades its loss. It is a loss of scientific provenance and accountability, which is this chapter's subject rather than the ladder's. What the colleague erased is the one part of the record a baseline test or a container rebuild was never designed to protect.

Draft, November 13, 2026. Content may change.

Answers to the checks

Check 1. *from earlier chapters* (page 109)

The collaborator's repository, with a `Dockerfile` and no lockfile, has earned L0, locatable, and no more. The rungs are cumulative (Chapter 3): L1 requires a lockfile, and L2 requires a container recipe as well, so a Dockerfile without a lockfile is L0 whatever its README says. The failing call is package churn in the package-versions layer, the code rot of Chapter 2. A lockfile, `renv.lock`, closes it and marks L1; restoring that lockfile inside the image then reaches L2. The sixth-decimal difference is the signature of the numerical-library layer. No rung guarantees it: the image determines which BLAS is installed, but the thread count and the processor still vary between machines (Chapter 2, Chapter 3). Fixing the thread count narrows the gap, and a stated tolerance in an L3 baseline test absorbs what remains.

Check 2. *what the drawer loses* (page 111)

The content of a change, what differs between two snapshot files, is recoverable by diffing them line by line, though at real cost in effort. The reason for a change is not recoverable from the files at all. It lived only in the person who made the change, and once they forget it or leave the project, it is gone for good. Version control is useful precisely because the reason is the one loss that no amount of careful reading of the files themselves can undo.

Check 3. *which judgment is health-specific* (page 112)

The first, deciding what must never be committed. Choosing a commit's granularity and message, and deciding when to branch or tag, are matters of craft that apply to any software project, health-related or not. The judgment that protected health information and identifiable data must never enter the history carries legal and ethical force specific to working with human-subjects data. It is also the one judgment in this chapter whose consequence, once realized, no git command can undo.

Check 4. *what one message restores* (page 114)

It restores the reason for the change, naming the reviewer's request as the cause. It does not, by itself, restore the identity of the working version, which comes from git keeping one tracked `analysis.R` rather than a drawer

Draft, November 13, 2026. Content may change.

of copies. Nor does it restore the precise content of the change, which comes from the diff the commit carries alongside its message.

Check 5. *why stage at all?* (page 117)

It buys the ability to make a commit an *atomic unit of one idea* rather than a snapshot of whatever happened to be on disk. In an afternoon an analyst may fix a bug in the cleaning script, add an unrelated figure, and jot a half-finished note in a third file. Committing all three together produces a history entry that cannot be described in one honest message and cannot be reverted without disturbing unrelated work. Staging lets the analyst record the bug fix as one commit, with its own message and reason, and the figure as another, leaving the unfinished note uncommitted. The history is only as useful a record as its individual commits are coherent, and staging is the mechanism that coherence requires. (For the genuine everyday case where one *does* want everything, `git commit -a` stages all tracked changes in one step, so the friction is opt-out, not mandatory.)

Check 6. *log or diff* (page 121)

The log answers the first question: `git log` lists each commit's message, and the fifth commit's message in Table 4.1 names the reviewer's request directly. The diff answers the second: `git show` on that commit's hash displays exactly which lines were added and removed. Neither replaces the other; the log without the diff says why but not what, and the diff without the log says what but not why.

Check 7. *the same file, two archetypes* (page 129)

The rendered `report.pdf` is a regenerable output. In a data-analysis or manuscript compendium it is a product the code reproduces on render, so committing it couples the history to a derivative and is usually avoided. An analyst who nonetheless commits it, perhaps to preserve the exact submitted PDF as a fixed reference, is making a defensible different judgment about what the artifact is *for*. It is a judgment, not a rule. The `data_dictionary.csv` and the registry extract sit on opposite sides of the one line that is not a matter of taste. The dictionary carries variable names and codings, which are metadata and safe to share, and belongs in the history so a reader can interpret the analysis. The extract carries the protected health information itself and must never be committed, whatever

Draft, November 13, 2026. Content may change.

its convenience. The `!` re-inclusion pattern exists precisely to keep the safe metadata while ignoring the unsafe data beside it.

Check 8. *what the merge changes* (page 132)

No. A branch adds new commits whose parent chain points backward to an existing commit on `main`; it never rewrites or recopies what came before the branch point. The merge adds one further commit with two parents rather than replacing anything. Every hash that existed before the branch was created, on `main` or anywhere else, is exactly as it was. That is what Figure 4.2's shared commits A through C illustrate.

Check 9. *which command for which undo* (page 135)

The unstaged edit: `git restore analysis.R`, since nothing has been committed yet. The private, unpushed commit: `git reset`, most simply `--hard` if the change should be discarded outright, since no one else has a copy of that commit to lose synchronization with. The already-shared commit: `git revert`. Reset or amend would rewrite a commit other clones already hold. Reverting instead adds a new commit that undoes it without touching what collaborators already pulled.

Check 10. *what does the pull request add?* (page 138)

The commit history records what changed and, in its messages, why. It does not record who *reviewed* the change before it entered the shared analysis, or what they said. In a solo project there is no reviewer, so nothing is lost. In a multi-site trial, a change to the shared analysis by one center affects every center's results. There the review is itself a controlled step, and the pull request is the artifact that records it: the proposed diff, the discussion, the requested changes, the approval, and the merge, each attributable and timestamped. The commit history answers 'what was changed and why'; the pull request answers 'who checked it, and did they agree'. For a regulated analysis both questions have auditors.

Check 11. *history without a lockfile* (page 139)

No. Everything described pins the analysis code and makes it locatable, which is exactly L0. Reaching L1 requires a lockfile recording the exact version of every package the analysis used, and no amount of git history, however complete, records that on its own. A repository can be a model of version-control discipline and still sit at L0 if no lockfile exists.

Draft, November 13, 2026. Content may change.

Check 12. *what the worked example still lacks* (page 142)

A lockfile, such as `renv.lock`, recording the exact version of every package the analysis uses. Writing it is not a version-control operation in the sense this chapter teaches; it belongs to Chapter 6. Version control will track the lockfile once it exists, exactly as it tracks any other file, but producing it is a different act from committing, branching, or tagging.

4.16. Cheat sheet

Command or rule	What it does
`git init -b main`	Starts tracking the current directory as a repository, with the first branch named `main`.
`git add <file>`	Stages a change for the next commit.
`git commit -m '...'`	Records the staged changes with a message.
`git status`	Shows staged, unstaged, and untracked files; read it before every commit (habit 3).
`git log --oneline`	Lists commits, most recent first, by hash and message.
`git show <hash>`	Displays one commit's full diff.
`git switch -c <name>`	Creates and switches to a new branch.
`git remote add origin <url>`; `git push -u origin main`	Connects to, and first pushes to, a remote.
`git tag -a v1.0 -m '...'`	Marks one commit permanently, e.g. at submission (Chapter 13 anchors to this).
`!pattern` in `.gitignore`	Re-includes a path otherwise ignored, e.g. a data dictionary beside an ignored extract.

Draft, November 13, 2026. Content may change.

Command or rule	What it does
`git check-ignore -v <path>`	Names the `.gitignore` rule that ignores a path, or prints nothing if none does.
`git rm --cached <file>`	Untracks a file already committed; does not erase it from the history.
Git refuses	Look up the message in When git refuses; run the Git preflight once per machine.
`git filter-repo --sensitive-data-removal`	Rewrites every commit to remove a file; only on a fresh clone, only before re-pushing. Where records are regulated, only as an approved, documented controlled change.
Rule of thumb	PHI, credentials, and restricted data are never committed; deleting them later does not remove them from the history.
Rule of thumb	Version control reaches L0 only; it pins the analysis code, not the packages or the environment.
Habit 1	Commit before you compute.

4.17. Further reading

- Ram (2013) is the direct argument for git as an instrument of reproducibility and transparency in science. It is the natural first reading for a researcher who has met git only as a software tool, and it makes the case for the commit history as a record of scientific provenance.
- Bryan (2018) is the standard orientation to version control written for data scientists rather than software engineers. It motivates GitHub as the collaboration substrate with the working analyst's concerns foremost. The longer online book by Bryan and colleagues, *Happy Git and GitHub for the useR* (Bryan et al., n.d.), is the companion practical manual. The setup appendix points to its chapters on installing git, identity, and credentials.

Draft, November 13, 2026. Content may change.

- Wilson et al. (2017) and Sandve et al. (2013) place version control among the good practices of reproducible computational research. The former treats it as one of a set of habits for scientific computing, and the latter places it among its rules for reproducible analysis. Read them to see version control situated alongside the other pillars this book develops.
- The layers that version control does not pin are the subject of Chapter 6 and Chapter 7. The continuous integration and collaboration built upon a git history are developed in Chapter 12 and in Chapter 13; this chapter is best read as the foundation those four presuppose.

Draft, November 13, 2026. Content may change.

5. The Research Compendium

The purpose of this article is to show how the R package can be a suitable template for organising files into a research compendium to enhance the reproducibility of research.

Marwick, Boettiger, and Mullen, *Packaging Data Analytical Work Reproducibly Using R (and Friends)* (2018)

By the end of this chapter you will be able to look at a folder of scripts and spreadsheets and name what kind of research object it is trying to be. You will also be able to draw the line between what counts as raw input and what counts as a derived product. Finally, you will be able to say exactly which single build gate, `R CMD check`, a tidy directory earns for free, and what that gate still leaves unexamined.

Before you start

Reading time. About 60 minutes.
Prerequisites. Chapter 3, for the L0-L3 ladder vocabulary this chapter uses when it places a compendium on it; no other chapter is assumed. Statistically, nothing beyond reading a hazard ratio and a coefficient table, as in the chapter's cohort and comorbidity examples, is required. No R fluency beyond reading a short script and a directory listing is assumed; this chapter writes almost no code itself.
By the end of this chapter you will be able to answer:

- Can you explain what a research compendium is, and trace its lineage from Gentleman and Temple Lang through `rrtools`?
- Can you justify why the R package is the compendium container, say precisely what `R CMD check` thereby checks, and say what it does not?
- Can you enumerate the Five Pillars of a compendium and name the single artifact that captures each?

Draft, November 13, 2026. Content may change.

- Can you distinguish the five compendium archetypes and state how their directory layout, their report, and their gates differ while the reproducibility machinery stays the same?
- Can you apply progressive disclosure to walk a project from one archetype to another without reorganizing it?

Outline.

- Where this sits on the stack: locatable, and nothing pinned yet.
- Orientation: a cohort analysis that is a scatter, not an object.
- The analyst's contribution: naming the archetype, fixing the raw-and-derived boundary, deciding what becomes a function.
- The compendium as the unit of reproducible work: where the idea comes from.
- The R package as the compendium container: `R CMD check` as a universal gate, and its limit.
- Building and maintaining the package skeleton: the `usethis` and `devtools` commands.
- The Five Pillars.
- The directory layout, concretely.
- The README and the data dictionary, including the README's computational requirements.
- The five archetypes, each with a health-sciences example.
- Progressive disclosure: one layout, several stages.
- Worked example: one compendium through two archetypes.
- The cohort, continued: this book's own compendium against the Five Pillars.

Check 1: from earlier chapters

A cohort analysis was tagged `v1.0-submission` in 2018 and pushed to a remote. In 2024 you check out that tag and re-run it under the current R. The code matches the tag, and it sets a seed before drawing bootstrap resamples with `sample()`. Yet the bootstrap interval no longer matches the paper. There is no lockfile and no container. Drawing on Chapter 2, Chapter 3, and Chapter 4, which determinant layer most likely moved, which rung did the tag secure, and what

Draft, November 13, 2026. Content may change.

would have held that layer fixed?
The answer is on page 205.

5.1. Where this sits on the stack

We begin by placing the chapter on the determinant stack of Chapter 3. In the sense of pinning a package version or a system library, this chapter captures nothing on that stack. What it captures are the two layers that sit above the packages and the system, the analysis code and the input data, by making them locatable and organized in the first place. Figure 5.1 marks these two layers. A compendium is what first turns a scatter of files into something a reader can find and name, and within which the reader can locate the raw-versus-derived boundary. This is the L0 rung and no further.

```
plot_determinant_stack(
  captures = c('Analysis code', 'Input data'),
  title = 'The compendium captures locatability, not yet
   ↪  packages or environment')
```


Draft, November 13, 2026. Content may change.

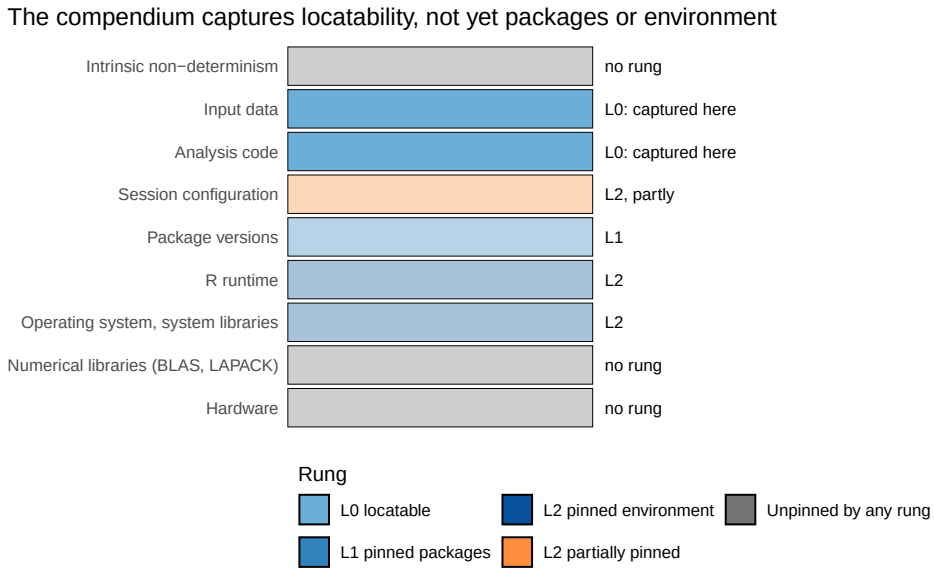


Figure 5.1.: The determinant stack of Chapter 3, with analysis code and input data marked as captured by this chapter. A research compendium does not pin a package version or an operating system; what it captures, for the first time, is that the code and the data are locatable, organized by convention, and separated into raw and derived, which is the L0 rung this chapter secures and the container the later chapters install L1 through L3 into.

5.2. Orientation

Consider a first-year epidemiology student who has spent three months on a cohort analysis. The analysis is an incident-diabetes study drawn from a registry, with a cleaning script, a handful of models, four figures, and a draft of a results section. The work exists, but it exists as a scatter. The cleaning script sits in a downloads folder, and the models sit in a second file that reads a spreadsheet whose path is hard-coded to one laptop. The figures are loose `png` files pasted by hand into a manuscript, and the record of which package versions produced any of it is nowhere at all. Nothing here is wrong as statistics. The difficulty is that the analysis is not an object. It

Draft, November 13, 2026. Content may change.

cannot be handed to a colleague, cited, archived, or re-run, because there is no single thing to hand over.

The remedy is to make the analysis one object. A *research compendium* is that object. It is a single, self-describing directory that bundles the code, the data, the prose, and the record of the environment in which the analysis was run. Its contents are organized by convention, so that another person, or the same person a year later, can find every part and regenerate the result. The compendium is the third of the recurring ideas of this book, after the levels ladder and the capture-versus-validate distinction. It is also the one that gives the other two something to act on. A reproducibility level is a property *of* a compendium; a validation gate is a check run *against* a compendium.

The compendium also sets the starting rung on the ladder. Merely by being a version-controlled directory whose parts can be located, a compendium reaches L0, locatable. It is not yet L1, because nothing has pinned the package versions. It is not yet L2, because nothing has pinned the environment. And it is certainly not yet L3, because nothing has re-run it and shown the numbers to match a recorded baseline. The compendium provides the container into which the later chapters install those higher guarantees.

The remainder of the chapter proceeds as follows. The three judgments that remain the analyst's alone come first, and we then trace where the idea of the compendium comes from and why the R package became its container. We next set out the Five Pillars and the directory layout, then the README, the data dictionary, and project-oriented paths with `here`. The five archetypes follow, each with a health-sciences example and a note of how it weighs the chapters ahead. Lastly, we consider progressive disclosure, work one compendium through two archetypes, and measure this book's own compendium against the Five Pillars.

Check 2: what the scatter is missing

The epidemiology student's three months of work already contains correct statistics. What, precisely, does the compendium add that the statistics alone do not supply?
The answer is on page 205.

Draft, November 13, 2026. Content may change.

5.3. The analyst's contribution

Three judgments at the center of this chapter are the analyst's alone. No tool scaffolds them. Getting them wrong produces a tidy directory that is nonetheless the wrong shape for the work.

1. **Naming the archetype.** The analyst must decide what kind of object the project is: a data analysis, a manuscript, a reusable package, a simulation study, or a blog. This choice governs the layout, whether a report is rendered, and which gates apply. It is a statement of intent, not a technical setting, and it is the organizing decision of the chapter.

2. **Fixing the raw-and-derived boundary.** The analyst must decide what is an immutable input and what is a product of the analysis. A file placed in `raw_data/` is a promise that it will never be modified; a file in `derived_data/` is a promise that it can always be regenerated from the raw inputs and the code. No program can draw this line, because it depends on where the analyst judges the science to begin.

3. **Deciding what becomes a function.** The analyst must decide which code is a throwaway pipeline step, belonging in `analysis/scripts/`, and which is a reusable procedure worth documenting and testing, belonging in `R/`. This judgment determines what `R CMD check` and the test suite will actually protect, and it is revisited continuously as a project matures.

Check 3: three judgments, one tool-free

A newcomer asks which of `zzcollab`'s commands tells the analyst where the raw-and-derived boundary falls. What is the honest answer? *The answer is on page 206.*

The vocabulary of this chapter

- **Research compendium.** A single self-describing directory bundling the code, data, prose, and environment record of an analysis, organized by convention so that it can be located and

Draft, November 13, 2026. Content may change.

re-run.

- **R-package skin.** The minimal set of files (`DESCRIPTION`, `NAMESPACE`, an `R/` directory) that makes a directory a valid R package, and therefore checkable by the R build tools.
- **`R CMD check`.** The standard R command that builds a package, runs its tests, and applies dozens of standardized checks; here it is a universal integrity gate on the compendium's package skin (`R/`, the tests, `DESCRIPTION`). It does not check `analysis/`, which the build excludes.
- **The Five Pillars.** The five artifacts a compendium manages, one per capture point: the `Dockerfile`, `renv.lock`, `.Rprofile`, the source code, and the data.
- **Archetype.** The kind of research object a compendium is (data analysis, manuscript, package, simulation study, or blog), which varies the layout and the gates but not the underlying machinery.
- **Render gate.** A validation check that a report or document rebuilds cleanly from source; it applies to archetypes that carry a report and not to those that do not.
- **Raw versus derived data.** Immutable original inputs versus the processed datasets regenerated from them by code.
- **Progressive disclosure.** The property that one layout supports a project's evolution from one archetype to another with no reorganization.

5.4. The compendium as the unit of reproducible work

We turn now to the history of the idea, which is older than its tooling and which explains the shape of the modern object. Gentleman and Temple Lang, writing in 2007, proposed the compendium in its current sense: a document that combines text, code, and data in a form a reader can actually execute, rather than a paper that merely describes an analysis performed elsewhere (Gentleman & Temple Lang, 2007). The proposal drew on Knuth's earlier notion of literate programming, in which prose and code are woven into one source (Knuth, 1984). For some years the compendium remained an aspiration without a conventional implementation. Peng, arguing that reproducibility should be treated as a minimum standard for computational

Draft, November 13, 2026. Content may change.

science, sharpened the motivation without settling the container (Peng, 2011).

The decisive move came from Marwick, Boettiger, and Mullen. They observed that a container of exactly the required shape already existed and was in daily use by thousands of R developers: the R package (Marwick et al., 2018). An R package, as standardized by Wickham (Wickham, 2015), already provides a `DESCRIPTION` file recording metadata and dependencies and an `R/` directory for reusable functions. It also provides a `tests/` directory for unit tests, a `vignettes/` directory for long-form prose, and a rigorously specified build system. Marwick and colleagues extended this layout with a single top-level `analysis/` directory holding subdirectories for raw data, derived data, scripts, figures, and the manuscript itself. They packaged the whole as the `rrtools` tool. The resulting compendium can be installed, tested, and checked with the very same commands used for any R package. Nüst, Boettiger, and Marwick later wrote a companion guide on how to *read* such a compendium. That such a guide exists is a useful sign that the object had matured enough to have a reading protocol at all (Nüst et al., 2018).

> Check 4: why the R package, not a new format
>
> Marwick, Boettiger, and Mullen could have proposed a wholly new compendium format instead of reusing the R package. What did reusing an existing format buy that a bespoke one would not have? *The answer is on page 206.*

5.5. The R package as the compendium container

The choice of the R package as the container is the most consequential design decision in this lineage. At first it looks like a convenience, the reuse of a familiar layout. However, it also makes an existing, rigorous, and universal gate apply to the compendium's package skin for free. In this section we state both what that gate reaches and what it does not.

The gate is `R CMD check`. When R was designed as a system for distributing statistical software, its authors built a command that takes a package and subjects it to dozens of standardized checks. The command checks that

Draft, November 13, 2026. Content may change.

the code parses and loads, that every documented function matches its documentation, and that declared dependencies are actually available. It also checks that the examples run and that the tests pass. The command was built to keep the tens of thousands of packages on CRAN mutually consistent. By dressing a compendium in the R-package skin, we inherit this entire apparatus at no cost. A compendium that passes `R CMD check` is one whose functions in `R/` load, whose stated dependencies resolve, and whose tests succeed. These properties hold in a clean process that carries none of the author's accumulated and undeclared state.

The qualification 'in `R/`' is exact, not cautious. But what, precisely, does the check examine? As `rcmdcheck::rcmdcheck()` and `devtools::check()` run it, `R CMD check` does not check the directory it is pointed at. It checks the package tarball that `R CMD build` makes from that directory, and the build omits every path listed in `.Rbuildignore`. (Plain `R CMD check` on a directory skips the build, but it parses R code only in the package's own directories, such as `R/` and `tests/`, so the scripts in `analysis/` go unparsed either way.) A compendium lists `^analysis$` there, as it must, because raw data, derived files, and a manuscript do not belong in an installable package. The consequence is that the check never sees `analysis/`. An analysis script with a syntax error, or one that calls `library()` on a package declared nowhere, passes `R CMD check` with no error, warning, or note. The same undeclared package used inside a function in `R/` draws a WARNING ('`::` or `:::` import not declared from'). The code in `analysis/` therefore needs gates of its own. These are a render gate, which executes it, and a dependency scan, which reads it for packages that the lockfile does not pin. Both are developed in Chapter 12.

Verification

To see the claim hold: add a deliberate syntax error to any script under `analysis/scripts/`, then run `rcmdcheck::rcmdcheck('.')` (we ran this on a toy compendium with R 4.6.1 and `rcmdcheck` 1.4.0) from the project root. It reports zero errors, warnings, or notes, because the broken file never entered the build. To make the same claim fail on purpose, move the broken code, or a function that calls an undeclared package, into a file under `R/` instead, and run the check again. It now fails or warns, because that file is part of what gets built.

Draft, November 13, 2026. Content may change.

> **Pitfall**
>
> A new compendium that omits `^analysis$` from `.Rbuildignore`, or deletes the line while editing the file by hand, does not fail silently. `R CMD build` then tries to fold the raw data, the derived files, and the rendered report into the installable tarball. This can make the build slow or bloat the package beyond what CRAN or a reviewer will tolerate. It can also fail outright on a binary data file the build process cannot parse as package data. The fix is to treat `.Rbuildignore` as part of the package skin, checked into version control like `DESCRIPTION`, not as an incidental dotfile.

The R-package structure is therefore part of what makes a directory a compendium. Every archetype we shall meet below, however different in outward shape, carries the same minimal R-package skin. It follows that `R CMD check` is the one gate that applies to all of them. A manuscript compendium adds a render gate on top; a simulation compendium adds a seeding discipline; a package compendium leans on the check most heavily of all; but the universal floor is the same. It is, however, a floor under `R/` and the tests, not under the analysis. This is one reason the analyst's third judgment, what becomes a function, matters: code moved into `R/` comes under the check, and code left in `analysis/` does not. We return to this point when we tabulate the archetypes, because it is the thread that unifies them.

> Check 5: why an R package?
>
> A colleague proposes to organize a compendium as a plain folder with sensible subdirectory names, arguing that the R-package files (`DESCRIPTION`, `NAMESPACE`) are empty ceremony for an analysis that is not a package anyone will install. What is lost by this choice?
> *The answer is on page 206.*

5.6. Building and maintaining the package skeleton

The R-package skin need not be written by hand. Two packages automate it: `usethis` writes the files of the skin, and `devtools` runs the development

Draft, November 13, 2026. Content may change.

loop on top of them. The chunk below lists the commands a compendium needs, in the order a new project meets them.

```
lock <- jsonlite::read_json('../../renv.lock')
lock_pkgs <- names(lock$Packages)
skeleton_tools <- c('usethis', 'devtools', 'pkgdown',
                    'pkgload', 'tinytest')
in_lock <- skeleton_tools %in% lock_pkgs
names(in_lock) <- skeleton_tools
in_lock
#>  usethis devtools  pkgdown  pkgload tinytest
#>    FALSE    FALSE    FALSE     TRUE     TRUE
```

```
# Not run here: several of these packages are outside
# renv.lock (see the text).
usethis::create_package('cohortstudy')
usethis::use_r('cohort')
usethis::use_gpl_license(version = 3)
usethis::use_package('survival')
tinytest::setup_tinytest('.')
usethis::use_pkgdown()

devtools::load_all()
devtools::document()
devtools::check()
pkgdown::build_site()
```

We do not run the second chunk. Of the five tools it calls, `usethis`, `devtools` and `pkgdown` are not recorded in this repository's `renv.lock`, so running them would step outside the pinned library; only `pkgload` and `tinytest` are recorded. That absence is defensible for development tools, since what they produce (`DESCRIPTION`, `NAMESPACE`, the help files in `man/`) is committed. A reproducer uses those files without rerunning the tools, and can run `R CMD check` from the shell with none of them installed.

The first six calls run once, or once per new file. `create_package()` writes `DESCRIPTION`, `NAMESPACE`, and `R/`, the minimal skin of the previous section. `use_r()` creates or opens a file in `R/`, the place where a function

Draft, November 13, 2026. Content may change.

comes under the check. The license call writes the `License` field and a license file. This compendium's `DESCRIPTION` records GPL-3. The call that writes GPL-3 is `use_gpl_license(version = 3)`; others, such as `use_mit_license()` or `use_cc0_license()`, write other licenses, and the choice is argued in Chapter 13. `use_package()` adds a package to `Imports`, which is how the undeclared-import WARNING of the previous section is cleared. Because the book tests with `tinytest` (Loo, 2020), its `setup_tinytest()` takes the place of the `use_testthat()` most tutorials show. It creates `inst/tinytest/` and the runner `tests/tinytest.R`, and adds `tinytest` to `Suggests`, where this compendium's `DESCRIPTION` has tinytest (>= 1.4.0).

The next three calls form the loop that is repeated while the code in `R/` changes. `load_all()` makes the package's functions available in the session by simulating a build, an install, and an attach, and Wickham & Bryan (2023) recommend it over `source()` for package work. It is supplied by `pkgload`, which this chapter's own first chunk calls directly. `document()` turns the roxygen comments above each function into help files and regenerates `NAMESPACE`. `check()` builds the tarball and runs `R CMD check` on it, the gate of the previous section, with the same blind spot for `analysis/`. A useful habit is to run `check()` before any commit that touches `R/`, `DESCRIPTION`, or the tests.

`pkgdown` builds a website from the help files, the README, and any vignettes. `use_pkgdown()` runs once and writes `_pkgdown.yml`; `build_site()` is rerun to refresh the site. A site is documentation, not a gate. It earns its place in the R-package archetype, whose users read reference pages, and is seldom worth maintaining for a data analysis. The full treatment of each step, and of the many `usethis` helpers not shown, is *R Packages* (2e) by Wickham and Bryan (Wickham & Bryan, 2023). Its chapter 'The Whole Game' runs the loop end to end on a toy package, its 'Website' chapter covers `pkgdown`, and an appendix treats `R CMD check` in detail.

5.7. The Five Pillars

Having settled the container, we turn to what it holds. A compendium is defined by convention, but it *manages* a specific and small set of artifacts.

Draft, November 13, 2026. Content may change.

The `zzcollab` framework organizes these as the Five Pillars. These are the five things a compendium reads, writes, and version-controls, one for each point at which reproducibility can be captured (team, 2026). An analysis is reproducible only to the extent that all five are captured. Moreover, the pillars are layered rather than independent, each resting on the ones beneath it.

1. **Computational environment**, captured by the `Dockerfile`. This pillar fixes the operating system, the R version itself, the system libraries, and the document toolchain, all the software beneath the R packages. It is the subject of Chapter 7.

2. **Package versions**, captured by `renv.lock`. This pillar records the exact version of every R package the analysis loads, so that the same set can be restored on another machine. It is the subject of Chapter 6.

3. **Session configuration**, captured by the project `.Rprofile`. This pillar fixes the R options, the locale and timezone, and the thread counts that shape a session before any user code runs. A profile can also override the random-number-generator kind, but such an override is secured only by an explicit `RNGkind()` call in the analysis code, and the seed belongs there too (Chapter 3, Chapter 11).

4. **Source code**, captured by the version-control history. This pillar holds the scripts, the functions, the manuscript source, and the tests that implement the analysis.

5. **Data**, captured either by inclusion in the compendium, for small redistributable files, or by reference to an external repository with a resolvable identifier, for large or restricted data.

Two of the pillars are not coordinate with the other three. Source code and data, pillars four and five, are the inputs whose re-execution is the goal. The environment, packages, and session, pillars one through three, are the conditions under which that re-execution occurs. This chapter is about the container that holds all five and the shape that container takes. The individual pillars are developed in the chapters named against them.

Draft, November 13, 2026. Content may change.

Check 6: layered, not independent

Why does the chapter say the pillars are 'layered rather than independent,' and what would it mean to capture pillar two, package versions, while leaving pillar one, the computational environment, uncaptured? *The answer is on page 206.*

! The honest level

This chapter's own practice stops at naming and separating the five artifacts. It leaves their actual capture, and the rung that capture buys, to the chapters named against each pillar above.

5.8. The directory layout, concretely

We now make the layout concrete. The following is the top-level structure of a `zzcollab` compendium. The block is illustrative and is not executed.

```
penguin-analysis/
├── DESCRIPTION           # package metadata and dependencies
├── NAMESPACE             # generated; exported functions
├── LICENSE
├── R/                    # reusable, documented functions
├── man/                  # generated function documentation
├── tests/                # tinytest runner
├── inst/tinytest/        # unit tests (test-*.R)
├── vignettes/            # long-form prose
├── analysis/
│   ├── data/
│   │   ├── raw_data/       # immutable inputs
│   │   ├── derived_data/   # regenerable script outputs
│   │   └── metadata/       # data dictionaries, provenance
│   ├── scripts/           # numbered pipeline steps
│   ├── report/            # report.Rmd / index.qmd manuscript
│   ├── figures/           # generated figures
│   └── tables/            # generated tables
```


Draft, November 13, 2026. Content may change.

```
├── Dockerfile            # Pillar 1: environment
├── renv.lock             # Pillar 2: package versions
├── .Rprofile             # Pillar 3: session configuration
├── Makefile              # workflow entry points
├── zzcollab.yaml         # configuration: base image, archetype
├── .zzcollab-state       # generated state record; never hand-
edited
└── .github/workflows/    # continuous-integration definitions
```

Above the `analysis/` directory sits the standard R-package machinery, and this is the skin that makes `R CMD check` apply. Within `analysis/` sits the research proper, in the Marwick convention. Its raw-and-derived data separation is the analyst's second judgment made physical. At the root sit the three environment artifacts, one per pillar. Alongside them sit the `Makefile` that provides the commands and the `zzcollab.yaml` that records, among a small set of parameters, the archetype. That last file is our bridge to the centerpiece of the chapter. The archetype is a first-class parameter recorded there, elicited at creation time alongside the choice of base image.

Check 7: finding the boundary in the tree

Using the directory listing above, name the one line that marks the boundary `R CMD check` actually respects, and state which side of that line `renv.lock` falls on.
The answer is on page 207.

5.9. The README and the data dictionary

The directory listing above names a `README` nowhere, and that absence is not an oversight to copy. A compendium that captures all five pillars perfectly but ships without a `README` still fails the first test a stranger applies to it, because the stranger has no entry point. The NHS RAP Community of Practice makes a README one of the criteria of its baseline level, alongside code in an open-source language, version control, peer review of the code, and open publication. A repository meets the README criterion only once

it 'includes a README.md file (or equivalent) that clearly details steps a user must follow to reproduce the code' (NHS England RAP Community of Practice, 2024). A `README.md` at the project root should state, in order, what the project is for, how to reproduce it, and where the data dictionary lives for anyone who needs the columns rather than the narrative. The reproduction instructions name the `make` target or `Rscript` command that runs the pipeline end to end, and the archetype-specific gate from Chapter 3 that passing it claims.

The data dictionary is a second, separate document, usually in `analysis/data/metadata/` or beside the extract it describes. It answers a question the README should not try to: what is each column, exactly. At minimum it gives, per column, the name, its type, a one-line meaning, its units where numeric, and its provenance. Provenance here means whether the column was measured, derived, or simulated and by what script. This book's own `analysis/data/README.md` is a worked, if unusually candid, example. Because the extract it describes is synthetic, the dictionary can also say outright which rows are planted defects and why. A real data use agreement would instead replace that disclosure with an access-control statement. The columns table there (`mrn`, `patient_name`, `hba1c`, and seven more, each with a type and a one-line meaning) is the shape to imitate. On the other hand, the provenance section above it, naming the exact generator script and seed, is the part a real project cannot write the same way. A real extract's provenance is the originating system and the extraction date, not a committed script.

> Check 8: README versus data dictionary
>
> A compendium's `README.md` says 'run `make all` to reproduce every figure in the report' and nothing else about the data. Is this README sufficient on its own, and what, if anything, is still missing from the compendium?
> *The answer is on page 207.*

5.9.1. The README's computational requirements

A README that names the command still leaves a reproducer guessing what the command needs. The template README of the Social Science

Draft, November 13, 2026. Content may change.

Data Editors, written for replication packages, gives this a section of its own, 'Computational requirements' (Vilhuber et al., 2022). Adapted to an R compendium, that section asks for the first four items below, and the template's 'List of tables and programs' supplies the fifth.

- **Software and versions.** The R version and every package, each at the version the author used. In a compendium the lockfile carries the list, and the README names the R version, points to `renv.lock`, and names the image digest once there is one.
- **Randomness.** A fixed seed, not one taken from the clock, and a statement of where each seed is set, or a statement that no random numbers are drawn. The template warns that parallel work and the order in which programs run both need care.
- **Runtime.** The wall-clock time of a full run, in a coarse category such as under ten minutes, and any storage needed beyond the repository itself.
- **Hardware.** The operating system, processor, and memory of the machine on which that time was measured.
- **A map from outputs to code.** Every table and figure, by its number in the manuscript, beside the program (and, where useful, the line) that produces it.

This book's cohort compendium makes a compact example. Its repository does not yet have a `README.md` at the root, so the excerpt below is the section such a file would carry. Every value in it, from the R version to the seed's line number and the runtime, is read from the repository or measured when this chapter is rendered, which is the same rule the README itself should follow.

```
## Computational requirements

- Software: R 4.6.1, packages as in renv.lock
  (restore with renv::restore()); key packages
  survival 3.8-6, withr 3.0.3, tinytest 1.4.3.
- Randomness: simulate_cohort.R calls
  set.seed(20260908) once, at line 19;
  bootstrap_cox_ci() sets seed 20260908 (B = 1000)
  with withr::with_seed(). Sampler kinds:
```


Draft, November 13, 2026. Content may change.

```
  Mersenne-Twister, Inversion, Rejection.
- Runtime: 3.6 s for the analysis and bootstrap
  (under 10 min).
- Hardware: aarch64-apple-darwin23, 8 logical cores.

| Output                  | Produced by |
|-------------------------|-------------|
| Synthetic extract       |
 ↪  analysis/scripts/simulate_cohort.R |
| Cox HR for BMI, 95% CI | R/cohort.R:
 ↪  run_canonical_analysis() |
| Bootstrap interval      | R/cohort.R: bootstrap_cox_ci() |
| Recorded baseline       |
 ↪  analysis/scripts/generate_baseline.R |
| Baseline check (L3)     |
 ↪  inst/tinytest/test-baseline-cohort.R |
```

Two points in the excerpt are easy to miss. The extract and the bootstrap draw from separate seeds, and the README says where each is set, because a reader who finds only one may assume the other draws from the clock. The runtime and hardware are those of the machine that last rendered this chapter, not of the reader's; they tell a reproducer whether a run is expected to take seconds or days, not what their own run will take. The excerpt also omits memory, which base R does not report in a portable way; an author measures it and adds the figure by hand.

5.10. Project-oriented paths: `here`, not `setwd()`

A script that opens with `setwd('/Users/jane/project')` reproduces on exactly one machine: Jane's. The fix taught by the `here` package is to mark the project root once, with a file such as `.here` or the marker RStudio and Git already leave behind. Every path is then built with `here::here()`, which walks up from the working directory until it finds that marker and returns paths relative to it from there on (Bryan & Hester, 2024). Consider a script that reads `here::here('analysis', 'data', 'raw_data', 'cohort_raw.csv')`. It resolves the same file whether R

Draft, November 13, 2026. Content may change.

is launched from the project root, from `analysis/report/`, or from a continuous-integration runner that checks the repository out to a different absolute path each time. This holds because the path is computed from the marker rather than typed as a string.

This book's own chapters, inspected directly, show why the discipline matters even without `setwd()`. Several chapters reach the cohort extract with `'../data/raw_data/...'`. However, `08-testing-quality.qmd` reaches the same file with `'../../analysis/data/raw_data/...'`, which climbs to the project root and back down again. From `analysis/report/` the two strings name one file by different routes, and nothing but the author's care keeps the two forms in agreement. Every one of these paths works only because the author tracked, chunk by chunk, how many `../` steps the renderer's working directory is from the target file. By contrast, `here::here('analysis', 'data', 'raw_data', 'cohort_raw.csv')` would resolve identically from any of these chunks, because it does not depend on the renderer's starting point at all. The relative-path convention is not wrong, exactly. It is fragile in a way that is invisible until a chunk is moved, copied into a different chapter, or run from a different tool.

5.11. The five archetypes

We come now to the centerpiece of the chapter. In this section we make the point this book returns to in every chapter: a research compendium need not be a manuscript. Once we have seen the `analysis/report/` directory, it is easy to assume that a compendium always exists to render a paper. That is the common case, but it is one case among five.

`zzcollab` treats the research archetype as a first-class initialization parameter, orthogonal to the choice of computational environment (team, 2026). Where the base image fixes the environment, the archetype determines the starting scaffolding: the directory emphasis, whether a primary report exists, and which validation gates apply on top of the universal one. Five archetypes are advertised. Because every one of them carries the minimal R-package skin, `R CMD check` remains the universal gate across all five. The framework's own whitepaper stresses this point in its account of the archetype axis. The reproducibility machinery is identical; what varies is

Draft, November 13, 2026. Content may change.

the shape of the work and the gates it warrants. We take each in turn, with a health-sciences example, and then tabulate the differences.

5.11.1. 1. Data analysis (a cohort study)

The first archetype is the exploratory and confirmatory data analysis with no manuscript as its primary organizing document. This is the closest to the original `rrtools` use case, and it is the archetype of the epidemiology student of our orientation. The work lives in `analysis/scripts/`, as numbered pipeline steps, and in `analysis/figures/`, with the derived datasets in `analysis/data/derived_data/`. Consider an incident-diabetes cohort study. A cleaning script reads the registry extract from `raw_data/` and writes an analysis-ready table to `derived_data/`. A modeling script fits a Cox proportional-hazards model and writes the coefficient table to `tables/`, and a plotting script writes the Kaplan-Meier curves to `figures/`. There may be a short results document, but the deliverable is the pipeline and its outputs.

The applicable gates are the universal `R CMD check` and the unit tests. The latter guard the cleaning logic and the data invariants. For example, they check that the cohort has the expected number of subjects. They also check that a derived rate matches a value recorded in a baseline file within a stated tolerance, rather than merely falling in a plausible range. A render gate is optional here, because there may be no report to render. The reproducibility machinery, environment and packages and session, is nonetheless present in full.

5.11.2. 2. Scholarly manuscript (a comorbidity paper)

The second archetype makes the manuscript the organizing principle. A `report.Rmd` or `index.qmd` file in `analysis/report/` is created on day one and written in parallel with the analysis, so that the prose and the numbers never drift apart. Every figure in the paper is regenerated on each render, rather than pasted in by hand. A representative deployment is a study of the comorbidity of PTSD, depression, and diabetes among Black women drawn from a trauma-study cohort. The study uses variable selection, multiple imputation, and causal mediation analysis. The

Draft, November 13, 2026. Content may change.

compendium organizes the manuscript in `analysis/report/report.Rmd`, the statistical functions in `R/` with `roxygen2` documentation, the analysis pipeline in `analysis/scripts/`, and the imputed datasets in `analysis/data/derived_data/`.

Here the render gate is central, not optional. A validation run rebuilds the report from source, and a failure to render is a failure of the compendium. The gates are therefore `R CMD check`, the unit tests, and the render of `report.Rmd` to PDF. This is the archetype most readers imagine when they hear the word compendium, and it is only the second of five.

5.11.3. 3. R package (a summary-table package)

The third archetype arises when the reusable code matures into something of general interest. At that point `R/`, `man/`, and `tests/` become the first-class content, and the compendium is shareable as a CRAN-style package. What distinguishes this archetype is that *there is no report to render at all.* A package that generates publication-quality clinical-trial summary tables has no manuscript as its deliverable; its deliverable is the exported, documented, tested function. A representative deployment is exactly such a summary-table package, hosting its source in `R/`, its examples in `vignettes/`, and its documentation as a generated website.

The gates for this archetype are `R CMD check`, run at its full strength, and the unit tests. There is no render gate, because there is nothing to render. This case is the clearest refutation of the manuscript-only view. An entirely valid, fully reproducible compendium can produce no report whatsoever, and its integrity rests on the universal check and the tests alone.

5.11.4. 4. Simulation study (a power analysis)

The fourth archetype is the simulation study, such as a power calculation or an assessment of the operating characteristics of a proposed trial design. In such a study the compendium enumerates a grid of parameter configurations, runs Monte Carlo replicates at each, and stores the results. The layout leans on `analysis/scripts/` for the simulation driver and on `analysis/data/derived_data/` for the stored replicate results. These are treated

Draft, November 13, 2026. Content may change.

as derived data because they can be regenerated from the code and the seeds.

But what makes a simulation reproducible at all? The distinguishing discipline of this archetype is *seeding*. A simulation whose random draws are not pinned by an explicit `set.seed()` is not reproducible even in principle, because a second run produces different numbers by construction. The archetype therefore emphasizes explicit seeding and parallel execution. A recommended practice is to store a seed column alongside the parameter grid so that each replicate is individually reproducible. Consider a power simulation for a longitudinal trial. The driver builds a grid over sample size and effect size and assigns each row an integer seed. It then simulates a dataset for each row, fits the planned model, and records whether the null was rejected. The empirical power is then the rejection rate within each cell. The gates are `R CMD check`, the unit tests, and, as an emphasis particular to this archetype, a baseline test that compares the seeded output with a value recorded in an earlier session and committed (Chapter 11). A second call in the same session is not such a test, because it agrees with the first even when the generator has changed. A render gate is optional, since the deliverable is often the stored results rather than a paper.

5.11.5. 5. Blog (a teaching notebook)

The fifth archetype is the blog or lab notebook, in which the compendium renders, in the general case, not one report but many documents. A teaching blog carrying a series of methods posts, or a running lab notebook of analyses, is a single compendium. Its `analysis/` tree may hold many `index.qmd` files, each rendered to a static site by the same machinery that renders a manuscript. The framework's representative deployment is the limiting case of a single post. Integration with Quarto (Allaire et al., 2024) enables publication to the site from the same compendium that produced the underlying analyses. As a result, the full five-pillar reproducibility stack applies to web-published teaching content without any change to the workflow.

The gate here is again the universal `R CMD check`, joined by a render gate that must succeed for *each* post the compendium carries. The multiplicity is the distinguishing feature of the general case. That is, a blog compendium

Draft, November 13, 2026. Content may change.

is a render-many, where the manuscript is a render-one and the package is a render-none.

The five archetypes and their differences are summarized in Table 5.1, and Figure 5.2 shows which validation gates apply to each. Both are best read across the rows, to see how the layout and the gates vary, and down the two universal columns, to see how much stays fixed.

```
library(tibble)
library(knitr)

archetypes <- tibble::tibble(
  archetype = c(
    'Data analysis', 'Scholarly manuscript', 'R package',
    'Simulation study', 'Blog'
  ),
  directory = c(
    'analysis/scripts/, analysis/figures/',
    'analysis/report/',
    'R/, man/, tests/',
    'analysis/scripts/, derived_data/',
    'many analysis/ index.qmd posts'
  ),
  report = c(
    'Optional', 'Yes, to PDF', 'No',
    'Optional', 'Yes, many'
  ),
  gate = c(
    'R CMD check and tests',
    'R CMD check plus render',
    'R CMD check and unit tests',
    'R CMD check plus seed determinism',
    'R CMD check plus render of each post'
  ),
  example = c(
    'Incident-diabetes cohort analysis',
    'PTSD and diabetes comorbidity paper',
    'Clinical-trial summary-table package',
    'Power and operating-characteristics simulation',
```


Draft, November 13, 2026. Content may change.

```
    'Teaching blog or lab notebook'
  )
)

knitr::kable(
  archetypes,
  col.names = c(
    'Archetype', 'Directory emphasis', 'Renders a report?',
    'Primary validation gate', 'Health-sciences example'
  )
)
```

Table 5.1.: The five research-compendium archetypes. The reproducibility machinery is identical across all five; only the directory emphasis, the presence of a rendered report, and the gates that apply on top of the universal R CMD check differ.

Archetype	Directory emphasis	Renders a report?	Primary validation gate	Health-sciences example
Data analysis	analysis/scripts/, analysis/figures/	Optional	R CMD check and tests	Incident-diabetes cohort analysis
Scholarly manuscript	analysis/report/	Yes, to PDF	R CMD check plus render	PTSD and diabetes comorbidity paper
R package	R/, man/, tests/	No	R CMD check and unit tests	Clinical-trial summary-table package
Simulation study	analysis/scripts/, derived_data/	Optional	R CMD check plus seed determinism	Power and operating-characteristics simulation
Blog	many analysis/index.qmd posts	Yes, many	R CMD check plus render of each post	Teaching blog or lab notebook

Draft, November 13, 2026. Content may change.

```
library(ggplot2)

archs <- c(
  'Data analysis', 'Scholarly manuscript', 'R package',
  'Simulation study', 'Blog'
)
gates <- c(
  'R CMD check', 'Unit tests', 'Render report',
  'Seed determinism'
)
grid <- expand.grid(
  archetype = archs, gate = gates,
  stringsAsFactors = FALSE
)

renders <- c('Scholarly manuscript', 'Blog')
seeded <- c('Simulation study')

grid$status <- 'not emphasised'
grid$status[grid$gate %in% c('R CMD check', 'Unit tests')]
 ↪  <-
  'universal'
grid$status[grid$gate == 'Render report' &
              grid$archetype %in% renders] <- 'applies'
grid$status[grid$gate == 'Seed determinism' &
              grid$archetype %in% seeded] <- 'applies'

grid$archetype <- factor(grid$archetype, levels =
 ↪  rev(archs))
grid$gate <- factor(grid$gate, levels = gates)
grid$status <- factor(
  grid$status,
  levels = c('universal', 'applies', 'not emphasised')
)

# Non-color cue: a point shape per status, so universal and
# selective cells differ by shape as well as hue (WCAG
 ↪  1.4.1).
```

```
# The two fills are only 1.34:1 apart, so the shape carries
 ↪  the
# distinction; the white markers are 7.1:1 and 5.3:1
 ↪  against them.
# Point shapes rather than a text glyph, because the PDF
 ↪  device
# cannot encode a Unicode circle. The shape scale shares
 ↪  the fill
# scale's labels so the two merge into one legend, and a
 ↪  reader who
# cannot separate the hues can still read the key by shape.
marked <- grid[grid$status != 'not emphasised', ]
status_labels <- c(
  'universal'      = 'Universal',
  'applies'        = 'Selective',
  'not emphasised' = 'Does not apply'
)

ggplot(grid, aes(x = gate, y = archetype, fill = status)) +
  geom_tile(colour = 'white', linewidth = 1.2) +
  geom_point(data = marked, aes(shape = status), colour =
   ↪  'white',
             size = 3.5) +
  scale_shape_manual(
    values = c('universal' = 16, 'applies' = 17,
               'not emphasised' = NA),
    labels = status_labels, drop = FALSE
  ) +
  scale_fill_manual(
    values = c(
      'universal'      = '#1a5c8c',
      'applies'        = '#1e7a50',
      'not emphasised' = '#7a8a9c'
    ),
    labels = status_labels
  ) +
  guides(fill = guide_legend(override.aes = list(size =
   ↪  2.5))) +
```


Draft, November 13, 2026. Content may change.

```
  labs(x = NULL, y = NULL, fill = NULL, shape = NULL) +
  theme_minimal(base_size = 11) +
  theme(
    panel.grid = element_blank(),
    legend.position = 'bottom'
  )
```

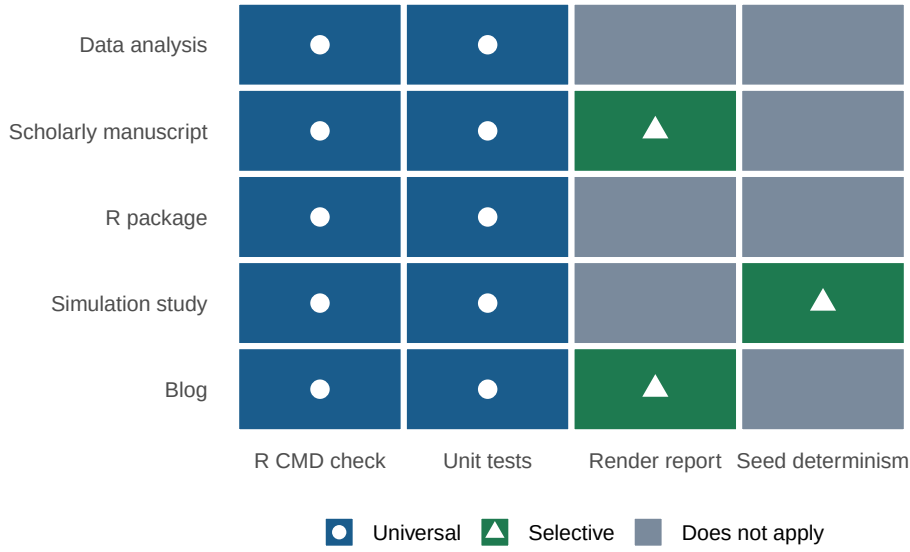


Figure 5.2.: Validation gates by archetype. The R-package check and the unit tests apply universally; the render and seeding gates apply selectively. Where a render gate is only optional, as for a data analysis or a simulation study without a report, the cell is left unmarked. The underlying reproducibility machinery, the Five Pillars, is identical across every row.

Check 9: the render-none case

A reviewer objects that a compendium producing no rendered document cannot be a reproducible research object, since there is no result to reproduce. Which archetype answers this objection, and how?

The answer is on page 207.

5.12. How each archetype treats the chapters ahead

The reproducibility machinery is identical across the five archetypes; only its weight shifts. Table 5.2 is the reference every later chapter points back to rather than restates. For each archetype, it records how the ladder reads it (Chapter 3) and how heavily it leans on a sealed render (Chapter 9). It also records which kind of test carries its weight (Chapter 11) and how its continuous-integration gates are configured (Chapter 12). Lastly, it records how much a pipeline tool earns its keep (Chapter 10) and how the compendium is ultimately shared (Chapter 13).

Draft, November 13, 2026. Content may change.

Table 5.2.: How each archetype treats the topics of the chapters ahead. The reproducibility machinery under every cell is the same; the table records only where the emphasis falls.

Archetype	Ladder (Chapter 3)	Literate programming (Chapter 9)	Testing (Chapter 11)	CI gates (Chapter 12)	Pipeline (Chapter 10)	Sharing (Chapter 13)
Data analysis	Same rungs as a manuscript; L3 via a baseline on a derived rate	Light; an internal HTML summary, not a typeset report	Leans on data-validation and reproducibility tests	Render gate only if a report exists	Strong case for `targets`, since it is re-run constantly and skipping unchanged steps saves time	Deposited as itself
Scholarly manuscript	Full apparatus; L3 compares tables and figures with recorded baselines	Center of gravity; the sealed render this chapter is built around	Adds a report-renders-and-matches-baseline test	Render gate is central, not optional	Modest; often a one-target build around the sealed report	Deposited beside the article it supports

Draft, November 13, 2026. Content may change.

Archetype	Ladder (Chapter 3)	Literate programming (Chapter 9)	Testing (Chapter 11)	CI gates (Chapter 12)	Pipeline (Chapter 10)	Sharing (Chapter 13)
R package	L1 and L2 as for any archetype, since `R CMD check` is a gate, not a rung; L3 via a baseline on function output after a clean rebuild	Largely absent; help pages and vignettes stand in	Native home of unit testing; `R CMD check` runs the suite	No render gate unless a vignette exists	None; `R CMD check` plays the corresponding role	Also released to CRAN or an R-universe, not only archived
Simulation study	L2 is nominal without the seed discipline; RNG kind is the load-bearing determinant	Thin; a presentation layer over precomputed, stored results	Leans hardest on the seed-determinism test	A reduced, fixed-seed grid runs under CI; the full grid runs elsewhere	Strongest case of the five; `targets` branching maps the parameter grid	Recorded seeds and the union lockfile are what a re-runner needs

Draft, November 13, 2026. Content may change.

Archetype	Ladder (Chapter 3)	Literate programming (Chapter 9)	Testing (Chapter 11)	CI gates (Chapter 12)	Pipeline (Chapter 10)	Sharing (Chapter 13)
Blog	Applies post by post; L3 needs a baseline per post	Central but multiplied across many rendered posts	Tests the code each post's examples depend on	Render gate iterates over every post	Rebuilds only the posts whose sources changed	Deployed as a website; the compendium behind it is still deposited

5.13. Progressive disclosure

But must a project choose one archetype and keep it? The five archetypes can also be stages a single project passes through. The framework's most useful structural property is that the same layout supports the whole trajectory without reorganization. This is what the framework calls *progressive disclosure*: the same directory layout, the same `Makefile`, and the same Five Pillars carry a project from one archetype to the next.

The trajectory is easy to picture, and it is the one our epidemiology student is likely to walk. The project begins as a data analysis: scripts and figures, no manuscript. As the findings firm up, a `report.Rmd` is added to `analysis/report/` and the project becomes a manuscript compendium, with the render gate now active. Nothing already written needs to move. In the course of the analysis a cleaning routine and a modeling helper prove useful enough to document and test, so they migrate from `analysis/scripts/` into `R/`. The package archetype's gates begin to bite on them without any change to the layout. Should one of those functions reach general interest, the compendium can be shared as a package, with

Draft, November 13, 2026. Content may change.

its `R/` and `man/` and `tests/` now first-class. The `analysis/` directory then recedes to a development notebook. The same compendium has been, in turn, a data analysis, a manuscript, and a package, and at no point was it reorganized.

The archetype therefore describes a project's current emphasis; it is not a permanent category. The reproducibility machinery is invariant across the archetypes and across the transitions between them. It is this invariance that lets a project evolve without leaving the reproducible state it started in. The worked example that follows traces the first part of this trajectory in detail.

> Check 10: what moves, what does not
>
> In the student's trajectory above, name one thing that changes at each transition and one thing that never does.
> *The answer is on page 207.*

5.14. Worked example

To make the progressive-disclosure trajectory concrete, we follow a single compendium through its first two archetypes. We stay at the level of the directory and the gates rather than the analysis. Suppose our student begins the incident-diabetes study. The compendium is created and its archetype recorded as a data analysis. The shell session below is illustrative and is not executed.

```
$ mkdir diabetes-cohort && cd diabetes-cohort
$ zzc analysis  # tidyverse base-image profile; archetype
 ↪  set separately
```

The student places the registry extract under `raw_data/` and writes `analysis/scripts/01_clean.R` to produce a derived cohort table. The student then writes `analysis/scripts/02_model.R` to fit the Cox model. A unit test in `inst/tinytest/` asserts the cohort size and that the fitted hazard ratio matches a value recorded in a committed baseline file, within

Draft, November 13, 2026. Content may change.

a stated tolerance. This is a test that can actually fail if the model's inputs or the `survival` version drift. By contrast, a bound such as 'no hazard ratio is negative' would be satisfied by every correct and every broken fit alike. At this point the only gates that apply are the universal check and the tests, run in a clean environment:

```
$ make check-renv     # dependencies declared and locked
$ make docker-test    # tinytest only
$ make docker-check   # R CMD check
```

Some weeks later the results are firm enough to write up. The student adds `analysis/report/report.Rmd`, moving the figure code out of `02_model.R` and into report chunks that regenerate the figures on render. The compendium is now a manuscript, and the archetype line in `zzcollab.yaml` is updated to reflect it. However, not a single existing file has moved. A render gate now joins the others:

```
$ make docker-render  # rebuild report.Rmd to PDF
```

The cleaning routine, meanwhile, has grown into a general helper worth documenting. It therefore migrates from `analysis/scripts/01_clean.R` into `R/prepare_cohort.R`, gains a `roxygen2` block, and picks up its own unit tests. The package archetype's discipline now protects it, and `R CMD check` verifies that the documentation matches the function signature. The same compendium is now doing the work of three archetypes at once, on one unchanged layout. Each gate applies to the part of the project that warrants it.

Check 11: the manuscript transition

When `report.Rmd` is added and the project becomes a manuscript compendium, what single gate becomes active that was not before, and what about the compendium's Five Pillars status is unaffected by this transition?
The answer is on page 207.

Draft, November 13, 2026. Content may change.

5.15. The cohort, continued

The examples so far have all described a hypothetical student's compendium. We now perform the same classification exercise on this book's own running cohort compendium, as it sits in this repository when the chapter is rendered.

```
raw <- load_cohort('../data/raw_data/cohort_raw.csv')
cohort <- clean_cohort(raw)
pillars <- c(
  environment_dockerfile = file.exists('../../Dockerfile'),
  packages_renv_lock = file.exists('../../renv.lock'),
  session_rprofile = file.exists('../../.Rprofile'),
  source_code_r_cohort = file.exists('../../R/cohort.R'),
  data_generator =
   ↪  file.exists('../scripts/simulate_cohort.R')
)
pillars
#> environment_dockerfile     packages_renv_lock
 ↪  session_rprofile
#>                  FALSE                   TRUE
 ↪  TRUE
#>   source_code_r_cohort         data_generator
#>                   TRUE                   TRUE
sum(pillars)
#> [1] 4
nrow(raw)
#> [1] 338
nrow(cohort)
#> [1] 337
pillar_label <- c(
  environment_dockerfile = 'the environment pillar
   ↪  (`Dockerfile`)',
  packages_renv_lock = 'the package-versions pillar
   ↪  (`renv.lock`)',
  session_rprofile = 'the session pillar (`.Rprofile`)',
  source_code_r_cohort = 'the source-code pillar
   ↪  (`R/cohort.R`)',
```


Draft, November 13, 2026. Content may change.

```
  data_generator = 'the data pillar (`simulate_cohort.R`)'
)
pillars_absent <- if (any(!pillars)) {
  paste(pillar_label[!pillars], collapse = ' and ')
} else {
  'no pillar'
}
rung <- if (pillars[['packages_renv_lock']] &&
              pillars[['environment_dockerfile']]) {
  'L2'
} else if (pillars[['packages_renv_lock']]) {
  'L1'
} else {
  'L0'
}
```

In this repository, 4 of the Five Pillars have an artifact. The source-code pillar is `R/cohort.R`, under version control. The data pillar is the committed generator `analysis/scripts/simulate_cohort.R`, which writes the synthetic extract from a fixed seed; the CSV it writes is gitignored, so the chunk checks the generator, not the extract. Of the five, the compendium still lacks the environment pillar (`Dockerfile`). Against the archetype table above, this compendium is a **data analysis**. Its deliverable is the pipeline of `R/cohort.R` and the chunks this book's chapters run against it, not a rendered manuscript of its own. The applicable gates are therefore the universal `R CMD check` and the unit tests, with no render gate required. Measured against the Five Pillars, the compendium is at **L1**, the rung Chapter 3 computes for it from the same files. The lockfile entry, TRUE, and the Dockerfile entry, FALSE, decide the rung. The source-code and data pillars make a compendium locatable, and the session pillar fixes the options a session starts with, but none of the three pins a package version or an environment.

> Check 12: pillars present, rung reached
>
> This compendium currently has 4 of the Five Pillars captured by an artifact, with the environment pillar (`Dockerfile`) absent. Does

Draft, November 13, 2026. Content may change.

holding 4 of five pillars, 80 percent of them, place it partway between two rungs of the ladder of Chapter 3?
The answer is on page 208.

5.16. Collaborating with an LLM

A language model is a capable assistant for scaffolding and organizing a compendium. However, it is an unreliable one for the judgments that define it. The three triples below mark the division.

! Restricted data

One rule comes first (habit 3). Never paste or upload protected health information, identifiable records, or data held under a data use agreement, nor outputs derived from them, into a model or a coding agent unless three conditions all hold. The institution has approved that endpoint for that class of data. The IRB protocol and the consent permit the processing. The data use agreement, or the provider's terms, permits it. If any of the three is unclear, the answer is no.
The third condition is the one most often missed. Institutional approval of a tool does not override the terms under which a data provider released the data. NIH, for example, treats sharing controlled-access data through prompts with public generative AI tools, which it glosses as third-party tools, as a breach of the data use certification. It treats a model trained on such data as a data derivative, to be shared only with other approved users and not kept after the project closes (National Institutes of Health, 2025a).
Describe the structure instead: the column names and types, or a small synthetic example generated from the data dictionary, never edited from real rows. A coding agent sees whatever files are in its working tree, so run it in a checkout or container where the real data is not mounted.
An agent also acts, so confine it to a branch and keep merges, pushes, and deposits for yourself (Section 2.10.1).

A compendium's own directory listing can be enough to tempt a reader into

Draft, November 13, 2026. Content may change.

pasting what it names. The prompts below describe the structure instead, with file names rather than file contents.

Prompt. 'I have a folder of R scripts and CSV files for a cohort analysis. Propose a research-compendium directory structure and tell me which files go where.'

Watch for. A plausible layout that silently guesses the raw-and-derived boundary, placing a file the model cannot distinguish as input or output into the wrong directory. Watch also for an omission of the R-package skin (`DESCRIPTION`, `NAMESPACE`) that quietly forfeits the `R CMD check` gate.

Verification. Confirm by hand that every file in `raw_data/` is a genuine immutable input and every file in `derived_data/` is regenerable from code, because this is the analyst's judgment and not the model's. Confirm the R-package skin is present, and run `R CMD check` to establish that the proposed structure is in fact checkable, remembering that the check covers `R/` and the tests, not the scripts under `analysis/`.

A better prompt. 'Below is a list of the file names in my cohort analysis, with no contents, and a note of which files the data office sent me. Propose a compendium layout with an R-package skin (`DESCRIPTION`, `NAMESPACE`, `R/`, `analysis/`). For each file, say raw or derived and why; mark any file you cannot classify rather than guessing. Then list the commands I should run to check the layout.'

Prompt. 'Which archetype should this project be, and which validation gates apply?'

Watch for. Check whether the answer defaults to the manuscript archetype when the project has no report and is really a package or a simulation study. Check also whether it lists, with confidence, a render gate for a project that renders nothing.

Verification. Decide the archetype yourself from the project's actual deliverable, a document, a function, or a set of stored simulation results, and then check that the gates the model lists match Table 5.1 for that archetype. A render gate on a render-none package is the tell that the model has guessed the archetype wrongly.

Prompt. 'Write the unit tests that should gate this compendium.'

Draft, November 13, 2026. Content may change.

Watch for. Tests that assert only that functions exist or that a script runs without error, which are structural checks rather than correctness tests. For a simulation compendium, watch also for an absence of any test that the seeded output is reproducible.

Verification. Read each generated test and confirm it checks a value the analysis actually depends on, not merely that code executes. For a simulation study, add or verify a test that compares the seeded output with a baseline recorded in an earlier session and committed, within a stated tolerance (Chapter 11). That is the property the archetype exists to protect, so check for it first. A second call in the same session agrees with the first even when the generator has changed, so it does not test this. If an agent writes the tests, it can also run them and edit them until they pass. The expected values and the recorded baseline stay with the analyst, who reads every change under `inst/tinytest/` and `inst/baseline/` before it is merged (Section 2.10.1).

5.17. Exercises

1. Take an existing analysis of your own, currently a scatter of scripts and data files, and classify it as one of the five archetypes. State the deliverable that determines the classification, and list the validation gates that should apply on top of `R CMD check`.

2. For that same analysis, draw the raw-and-derived boundary explicitly: list every data file and assign it to `raw_data/` or `derived_data/`, justifying each assignment by whether the file is an immutable input or a regenerable product. Identify any file whose status is genuinely ambiguous and explain the ambiguity.

3. Explain, in a paragraph, why `R CMD check` is described in this chapter as a universal gate. What does dressing a compendium in the R-package skin buy that a plain, well-named folder does not?

4. Sketch a progressive-disclosure trajectory for a project that begins as a simulation study and ends as an R package. At each transition, state which gate becomes active or inactive, and confirm that no directory needs to be reorganized.

Draft, November 13, 2026. Content may change.

5. Consider a simulation compendium whose driver script contains no `set.seed()` call. At which rung of the reproducibility ladder can such a compendium sit at best, and why can it not, even in principle, reach L3? Write the one-line test you would add to gate the seeding.

6. The chapter claims that the manuscript is only one of five archetypes. Construct the strongest argument you can for the opposing view, that every serious research compendium should ultimately produce a rendered document, and then state where the argument fails.

7. On this book's own running cohort compendium, run `load_cohort()` and `clean_cohort()` from `R/cohort.R` against `analysis/data/raw_data/cohort_raw.csv`, as in 'The cohort, continued' above, and separately check `file.exists()` for the five artifacts the `pillars` chunk checks: `Dockerfile`, `renv.lock`, and `.Rprofile` at the project root, `R/cohort.R`, and the generator `analysis/scripts/simulate_cohort.R`. Report three numbers a grading script could check without fitting any model: the raw row count, the cleaned row count, and how many of the Five Pillars have an artifact present when you run the check.

8. This exercise returns to the reproducibility ladder of Chapter 3 (Chapter 3). Using the pillar count and the `file.exists()` results from Exercise 7, state which single rung of the L0-L3 ladder this book's own cohort compendium occupies when you run the check, and explain why the number of pillars captured does not place it partway between two rungs.

5.18. Solutions

Attempt an exercise before expanding its solution. The first two work on an analysis of your own, so what you produce will vary; the boxes say what a good answer establishes.

Exercise 1

No single correct answer. What a good answer establishes is that the classification follows from the **deliverable**, not from the subject

Draft, November 13, 2026. Content may change.

matter or the size of the project.
The discriminating question is what the project is *for*. If the answer is a paper, it is a scholarly manuscript and the render gate applies. If it is a set of results with no document, it is a data analysis and the render gate may or may not apply. If it is a function other people will call, it is an R package and the render gate does not apply at all. If it is an estimate of an operating characteristic obtained by repeated sampling, it is a simulation study and the seed-determinism gate applies. If it is a post, it is a blog.
The gates on top of the universal `R CMD check` are drawn from a short list. Unit tests apply universally; the render gate applies where a document is the deliverable; and seed determinism applies where the result is stochastic. Naming a gate that does not apply is as much the point of the exercise as naming one that does, since the chapter's claim is that the machinery is identical and only the gates differ.
If you find that your analysis is genuinely two archetypes, a manuscript that also ships a reusable function, you have found the ordinary case rather than an anomaly, and you should apply the union of the gates.

Exercise 2

No single correct answer. What a good answer establishes is the operative test, which is not 'is this file original' but **'if I deleted this file, could the code regenerate it?'**
If yes, it is derived. If no, it is raw, and it needs a provenance record and a checksum, because nothing in the compendium can reconstruct it.
Two ambiguous cases arise often enough to expect them. The first is a **cleaned extract received from a data coordinating center**. It is derived from someone's raw data, but not from anything in this compendium, so it is raw here: the compendium's code cannot regenerate it. The rule is that the boundary is drawn at the edge of *this* compendium, not at the edge of the world. The second is a **manually corrected file**, where an analyst fixed a handful of values by hand. It is not regenerable, because the corrections live nowhere but in the file, so it must be treated as raw. The better resolution is to make it derived by recording the corrections as code, a lookup

Draft, November 13, 2026. Content may change.

table of overrides applied by a script, which restores regenerability and makes the corrections reviewable.
A third case worth naming is a **cached expensive computation**, a model that took a day to fit. It is derived by the test, since the code can regenerate it, but storing it is sensible. It belongs in `derived_data/` and should be regenerable on demand rather than trusted indefinitely.

Exercise 3

`R CMD check` is universal because it is the one gate that applies to every archetype regardless of deliverable. A manuscript, a simulation, a blog, and a package all carry the same skin. That is, each has functions in `R/` that must load, declared dependencies that must resolve, and tests that must pass. `R CMD check` verifies exactly those three things without knowing or caring what the project produces. The archetype-specific gates, rendering a document and checking seed determinism, apply to some archetypes and not others; this one applies to all of them.
A complete answer also states the limit. The check runs on the built package, and `.Rbuildignore` excludes `analysis/`, so it never parses or runs the analysis scripts. A script there can carry a syntax error or an undeclared package and the check still passes. 'Universal' means it applies to every archetype, not that it covers every file. The analysis code is gated by the render gate and the dependency scan of Chapter 12.
What the R-package skin buys over a well-named folder is **an existing verification apparatus that the project becomes eligible for**. A plain folder can hold all five artifacts and can in principle reach any rung of the ladder, so the skin buys nothing in terms of capture. What it buys is that `DESCRIPTION` and `NAMESPACE` are the interface `R CMD check` expects. Supplying that interface makes a large, mature, externally maintained body of checks apply to the analysis for free. Without it there is no single standard command that verifies the code loads and the dependencies resolve. The analyst must then assemble an equivalent by hand, differently, for every project.
This is the same bargain as any other standard format: the structure is arbitrary in itself and useful because other things already understand

Draft, November 13, 2026. Content may change.

it.

Exercise 4

A serviceable trajectory has three transitions.
Simulation study. The seed-determinism gate is active, since the results are draws and their reproducibility depends on the generator. `R CMD check` and unit tests are active, as always. The render gate is inactive: there is no document yet.
Simulation study plus a report on its findings. The render gate becomes active. Seeding stays active, since the simulation still runs. Nothing else changes.
R package, the simulation logic extracted into exported functions. The render gate becomes inactive again if the report is dropped, or stays active if a vignette replaces it, which is the usual outcome. Seed determinism becomes inactive as a *project* gate and reappears inside the unit tests, as an assertion that a seeded function returns a recorded value. `R CMD check` becomes more demanding rather than newly applicable: it now checks documentation completeness and export consistency, which were vacuous when nothing was exported. No new structure is invented at any transition; the scaffold's directories already exist. `R/` held the functions throughout; it merely gains `@export` tags. `analysis/` held the driver and the report, and its files may move into `vignettes/`, which is a move of files between existing directories rather than a reorganization. `tests/` gains cases. The Five Pillars are untouched, because the reproducibility machinery does not depend on the archetype. What changes is which gates fire, and gates are configuration rather than structure.

Exercise 5

At best L2. The compendium can pin its packages with a lockfile and its environment with a container, so nothing stops it reaching the second rung. What it cannot do is reach L3.
Why not, even in principle. L3 means the recorded outputs have been shown to regenerate, within a stated tolerance, from a clean rebuild. Without a seed, the generator is initialized from a source

Draft, November 13, 2026. Content may change.

that varies between sessions, so every run draws a different sample and produces different numbers. There is nothing stable to compare a rerun against. An unseeded simulation has no recorded output that a rerun could match, so the L3 check is undefined rather than merely unperformed.

Note that the analysis may be perfectly good statistics. With enough replicates the Monte Carlo error is small and two runs agree to within it. But 'agrees to within Monte Carlo error' is a different and weaker claim than 'produces the recorded outputs to numerical tolerance', and it is the latter that L3 names.

A one-line gate. The cheapest form asserts that the driver sets a seed at all. `tinytest` runs each test file with `inst/tinytest/` as the working directory, so the path climbs two levels to the project root. The guard lets the file exit quietly under `R CMD check`, where `analysis/` is not part of the built package:

```
driver <- '../../analysis/scripts/run_sim.R'
if (!file.exists(driver)) exit_file('analysis/ not
 ↪  present')
expect_true(
  any(grepl('set\\.seed\\(', readLines(driver))),
  info = 'the simulation driver sets a seed'
)
```

That is a structural check, and it is worth saying so. It confirms the call is present, not that the results are stable. Running the simulation twice in one session and asserting the results agree is not the stronger gate either. Both runs share the same R and the same generator defaults, so a change to those moves both runs together and the assertion still holds. The stronger gate is a baseline test (Chapter 11). We run the reduced simulation once and save its summary to a committed baseline file. A `test-baseline-*.R` file then recomputes the summary in each later session and compares it with `expect_equal(..., tolerance = 1e-8)`. That is the comparison L3 requires, and it costs a great deal more to run than the one-line check.

Draft, November 13, 2026. Content may change.

Exercise 6

The strongest version of the opposing argument. A rendered document is the only artifact that forces the analyst to state what the analysis *found*. Code and tests establish that the computation runs and is internally consistent; neither compels an interpretation. A document binds a claim to the numbers that support it, and because it is rendered rather than typed, the binding cannot silently decay. Requiring one of every compendium would therefore catch a whole class of failure that gates cannot see. That failure is an analysis that runs perfectly and that nobody has ever had to summarize, in which an implausible number sits unremarked because no sentence was ever written next to it. The render gate would become universal, which is simplifying, and every project would carry a human-readable entry point, which is exactly what an adopter of someone else's compendium most wants.

Where the argument fails. It conflates the deliverable with the interpretation. For an R package, the deliverable *is* the behavior of the exported functions. That behavior is bound to its claims by unit tests, which assert what the function should return in a form that is checked on every build. The tests are the binding artifact, and they are a stronger binding than prose because they are executable. Requiring a rendered document in addition would add an artifact that no consumer of the package reads and that must be maintained in parallel with the tests, which is how documents drift out of date. The argument also proves too much. Taken seriously, it makes reproducibility a property of documents rather than of re-execution. Yet a package with no document but a complete, passing test suite is plainly more reproducible than a beautifully rendered report with no tests.

The synthesis the chapter would offer is that every compendium needs an artifact binding claims to computation, and that a rendered document is one such artifact rather than the only one.

Exercise 7

The solution for this exercise is in the instructor guide, which is distributed separately to course instructors. Self-study readers who

Draft, November 13, 2026. Content may change.

want to check their answer can compare the three autogradable values against those reported in the chapter's own 'The cohort, continued' section.

Exercise 8

The compendium sits at **L1**, as the `pillars` chunk in 'The cohort, continued' prints: the lockfile entry is TRUE and the Dockerfile entry is FALSE.
The pillar count from Exercise 7, 4 of five present, does not translate into a fraction of a rung. The ladder of Chapter 3 is not a tally across the Five Pillars; it is a cumulative scale keyed to two specific artifacts. L1 requires `renv.lock` specifically, and L2 requires an environment recipe specifically, such as a `Dockerfile`, in addition. The session-configuration pillar (`.Rprofile`), the source-code pillar, and the data pillar being captured is what L0 already asks for, namely locatable code and data under version control, and adds nothing beyond it. With the lockfile present and the Dockerfile absent, four of five pillars is L1, not 80 percent of the way to L2. A compendium cannot be partway to a pinned environment; either the recipe exists or it does not.

Answers to the checks

Check 1. *from earlier chapters* (page 163)

For the 2018 bootstrap interval that no longer matches, the most likely layer is the R runtime. R 3.6.0, released in 2019, changed the default sampler behind `sample()`, so the same seed now draws different resamples, as Chapter 2 shows. The tag is not at fault. It fixes the analysis code and makes it locatable, which is L0 and no more (Chapter 4). The default RNG kind comes with the R release, so it belongs to the runtime and is fixed at L2 (Chapter 3). Because the rungs are cumulative, L2 also needs a lockfile, which would rule out package churn, the other candidate. An explicit `RNGkind()` call in the analysis code would have fixed the sampler as well.

Draft, November 13, 2026. Content may change.

Check 2. *what the scatter is missing* (page 165)

The statistics already have correctness. What the compendium adds is the property of being a single, locatable object. That is, it is a thing that can be handed to a colleague, cited, archived, and re-run. This is possible because every part of it (the code, the data, and eventually the environment record) lives in one self-describing directory rather than scattered across a laptop's folders and a manuscript's pasted figures.

Check 3. *three judgments, one tool-free* (page 166)

None of them. The framework's commands create the directories (`raw_data/`, `derived_data/`) and enforce the gates once the boundary is drawn. However, drawing the boundary, like naming the archetype and deciding what becomes a function, is a judgment about the science. It is not a setting the tooling can infer from the files themselves.

Check 4. *why the R package, not a new format* (page 168)

A build system, a validation tool, and tens of thousands of developers already fluent in the convention, all for free. A bespoke format would have needed its own checker written from scratch and its own community taught to use it. The R package, on the other hand, came with `R CMD check` and a working ecosystem already attached.

Check 5. *why an R package?* (page 170)

The plain folder can still be locatable and can still hold all five artifacts, so it can in principle reach the higher levels of the ladder. What is lost is the universal gate. Without the R-package skin, `R CMD check` has nothing to check, so there is no single, standard, pre-built command that verifies the functions load, their dependencies resolve, and the tests pass. Without the skin, the analyst must then assemble an equivalent set of checks by hand for every project. (The analysis scripts are a separate matter; with or without the skin, they need a render gate and a dependency scan.) The ceremony is precisely the interface that makes an existing verification apparatus apply.

Check 6. *layered, not independent* (page 174)

A lockfile restored outside any pinned environment still depends on whatever R version and system libraries the host machine happens to supply. Consequently, pillar two's guarantee, 'the same packages,' is only as strong

Draft, November 13, 2026. Content may change.

as pillar one's, 'the same environment,' lets it be. Capturing packages without the environment is exactly Chapter 3's L1 without L2. It is a real step, but one whose benefit a different R release or a missing system library can still erase.

Check 7. *finding the boundary in the tree* (page 175)

The boundary is the `analysis/` entry itself, the one path `.Rbuildignore` excludes from the build. `renv.lock` sits outside `analysis/`, at the project root, so it is not excluded by that line. However, it is also not parsed by `R CMD check`, which reads `DESCRIPTION` for declared dependencies rather than the lockfile. The two artifacts answer different questions about the same package skin.

Check 8. *README versus data dictionary* (page 176)

The sentence satisfies the README criterion of the RAP baseline, since it states the one command a user must run; the baseline's other criteria are separate. What is missing is not part of the README's job: a data dictionary naming each column's type, meaning, units, and provenance. A reproducer who can rerun the pipeline but cannot tell what `hba1c` means or whether it was measured or simulated can regenerate the numbers but cannot interpret them.

Check 9. *the render-none case* (page 187)

The R-package archetype answers it. A package that exports a summary-table function has no report to render, yet it is fully reproducible. Its result is the behavior of the function, and that behavior is protected by `R CMD check` and by the unit tests. These regenerate and verify the function's outputs on every check. The absence of a render gate does not weaken reproducibility. It reflects only that the deliverable is a tested procedure rather than a document.

Check 10. *what moves, what does not* (page 192)

What changes: the archetype recorded in `zzcollab.yaml`, and the set of gates that apply (a render gate joins at the manuscript transition; `R CMD check`'s documentation and export checks start to bite once functions migrate into `R/`). What never changes: the directory layout itself. No file already written is moved or renamed at either transition, and the Five Pillars sit in the same three root-level artifacts throughout.

Draft, November 13, 2026. Content may change.

Check 11. *the manuscript transition* (page 193)

The render gate becomes active: a validation run must now rebuild `report.Rmd` to PDF, where before there was no report to render. The Five Pillars are unaffected, because adding a manuscript changes which gates apply, not which artifacts capture the environment, the packages, the session, the code, or the data. Those five pillars are the same five files or records before and after the archetype changes.

Check 12. *pillars present, rung reached* (page 196)

No. The ladder is not a count of pillars. It is a cumulative scale in which L1 specifically requires the package-versions pillar (`renv.lock`) and L2 specifically requires the environment pillar (`Dockerfile`) as well. With the lockfile entry TRUE and the Dockerfile entry FALSE, the compendium is at L1, and the other pillars present add nothing to that placement. Four of five pillars with the lockfile present and the Dockerfile absent is L1, not 80 percent of the way to L2. No rung is reached by partial credit across unrelated pillars.

5.19. Cheat sheet

Command or rule	What it does or checks
`DESCRIPTION`, `NAMESPACE`, `R/`	The minimal R-package skin; makes a directory checkable by `R CMD check`
`.Rbuildignore` entry `^analysis$`	Excludes `analysis/` from the built tarball; `R CMD check` never sees it
The Five Pillars	`Dockerfile` (environment), `renv.lock` (packages), `.Rprofile` (session), version control (source code), inclusion or a resolvable identifier (data)
'If I deleted this file, could the code regenerate it?'	The operative test for the raw-versus-derived boundary

Draft, November 13, 2026. Content may change.

Command or rule	What it does or checks
Five archetypes	Data analysis, scholarly manuscript, R package, simulation study, blog: one skin, differing gates
Progressive disclosure	One directory layout carries a project through every archetype with no reorganization
Rule of thumb	Name the archetype from the deliverable (a document, a function, stored results, a post), never from the subject matter or the project's size
Habit 7	Write the honest level down: a tidy compendium is a precondition for reproducibility, not evidence of it
`usethis::create_package()`, `use_r()`, `use_package()`	Write the package skin, a file in `R/`, and an `Imports` entry
`tinytest::setup_tinytest('.')`	Adds `inst/tinytest/`, `tests/tinytest.R`, and `tinytest` in `Suggests`
`devtools::load_all()`, `document()`, `check()`	The development loop: load, regenerate help and `NAMESPACE`, run `R CMD check`
README computational requirements	Software and versions, where each seed is set, runtime, hardware, and an output-to-code map

5.20. Further reading

The foundational proposal is Gentleman and Temple Lang's account of the compendium as an executable document combining text, code, and data (Gentleman & Temple Lang, 2007). It reads well alongside Knuth's original statement of literate programming, from which the idea descends (Knuth, 1984). The operationalization that this chapter builds on is Marwick, Boettiger, and Mullen's proposal that an R package is the natural compendium container, together with their `rrtools` tool (Marwick et al.,

Draft, November 13, 2026. Content may change.

2018). The reader who wants the R-package mechanics in full should consult Wickham's book on R packages, which is the standard reference for the skin that makes `R CMD check` apply (Wickham, 2015). For the complementary skill of *reading* a compendium one has received rather than writing one, Nust, Boettiger, and Marwick offer a short and practical guide (Nüst et al., 2018). Finally, Peng's argument that reproducibility is a minimum standard for computational science supplies the motivation that the compendium is built to serve (Peng, 2011). The individual pillars named in this chapter are developed in Chapter 6, Chapter 7, and the validation chapters that follow.

Draft, November 13, 2026. Content may change.

Part II.

Capturing the Environment

Draft, November 13, 2026. Content may change.

6. Pinning Packages with renv

In order to exactly reproduce a given result, it may be necessary to use programs in the exact versions used originally.

Sandve, Nekrutenko, Taylor, and Hovig, *Ten Simple Rules for Reproducible Computational Research*, Rule 3 (2013)

By the end of this chapter you will be able to pin the exact version of every package an analysis uses. A colleague who restores the lockfile a year from now will then run the same code against the same packages as the original run, rather than whatever happens to be current on their machine. You will also be able to say precisely which layer of the determinant stack that pin does and does not reach.

Before you start

Reading time. About 45 minutes.
Prerequisites. Chapter 3, for the L0-L3 ladder, the determinant stack, and the capture-versus-validation vocabulary this chapter relies on throughout. Chapter 5, for the `DESCRIPTION` file and the R-package archetype the dependency triad draws its middle term from. Statistically, nothing beyond reading a package version number is required. R knowledge assumed: calling `library()`, installing a package, and reading a JSON file as a nested list; no prior exposure to `renv` itself is assumed.
Objectives. By the end of this chapter, can you:

- Explain the package-churn failure mode, and say why pinning package versions is the step that lifts an analysis from locatable (L0) to package-pinned (L1)?
- Describe what an `renv.lock` file records, and carry out the init, snapshot, and restore workflow that produces and consumes it?

- State the dependency-triad invariant relating the code, the `DESCRIPTION`, and the lockfile, and run the automated check that keeps the three in agreement?
- Configure a dated snapshot of a package repository so that a restore draws fixed versions, as pre-compiled binaries where the platform is served?
- Delimit what `renv` does and does not pin, and thus explain why a lockfile alone cannot reach the pinned-environment level?

Outline. The package-churn failure mode and the L0-to-L1 transition it motivates; the three judgments that stay with the analyst; packrat to renv and what a lockfile records; the init, snapshot, restore workflow; the dependency-triad invariant and the automated check that enforces it; dated snapshots of a package repository; the limits of renv; lighter alternatives; the cohort's own dependencies, scanned for real; collaborating with an LLM; exercises, a cheat sheet, and further reading.

Check 1: from earlier chapters

A colleague clones your cohort repository at its submission tag and runs `analysis/scripts/01-clean.R` on the unchanged raw extract. The cleaned table has fewer rows than the committed `analysis/data/derived_data/cohort_clean.csv`. The history shows that the outlier threshold in the script was tightened after that file was last committed. On the same machine, the script as it stood when the file was committed reproduces the file's row count. Drawing on Chapter 3, Chapter 4, and Chapter 5, name the failure and the rule it breaks. *The answer is on page 247.*

6.1. Where this sits on the stack

This chapter's whole business is the lockfile, so it is the clearest capture in the book so far. It pins, for the first time, the package-versions layer of the determinant stack that Chapter 3 laid out. That is exactly the step from L0 to L1. Figure 6.1 marks that single layer against the ambient nine, with

Draft, November 13, 2026. Content may change.

every other layer left faded as secured, or left unpinned, elsewhere.

```
plot_determinant_stack(
  captures = c('Package versions'),
  title = 'Pinning packages captures one layer, and one
   ↪  layer only')
```

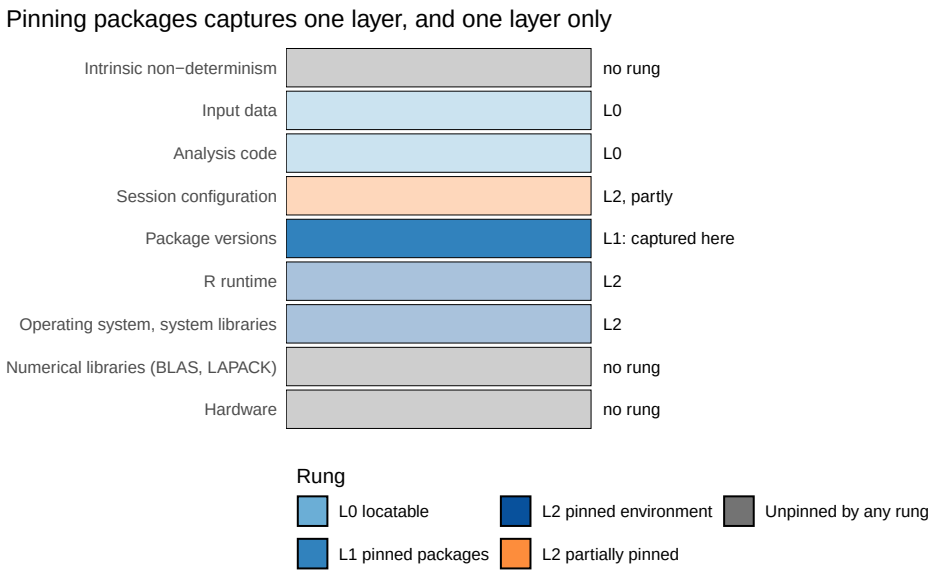


Figure 6.1.: The determinant stack of Chapter 3, with package versions marked as captured here. Pinning this one layer with a committed `renv.lock` is the L0-to-L1 transition this chapter teaches; the R runtime, the operating system, and the system libraries beneath it remain unpinned until Chapter 7 adds the container.

6.2. Orientation

To illustrate, consider a familiar situation. A doctoral student completes the analysis for a registry study of hospital readmissions, renders a clean report, and deposits the code alongside the manuscript. The work is under version control, so a reader can locate every script; on the ladder of Chapter 3

Draft, November 13, 2026. Content may change.

the analysis has reached L0, locatable. Eighteen months later a reviewer asks for a minor revision. The student pulls the repository onto a new laptop, installs the packages afresh, and reruns the analysis. The model now reports a different coefficient. Nothing in the code changed. What changed is that `dplyr` had moved from version 1.1.0 to 1.2.1, a grouping default had been refined, and one intermediate summary was computed over a slightly different partition. The number in the published table can no longer be regenerated, and the student cannot say with certainty which number was correct.

This is the package-churn failure mode. It is the specifically computational failure that the empirical studies of Chapter 2 isolate. The R package ecosystem is healthy precisely because it changes: maintainers fix bugs, sharpen defaults, and deprecate awkward interfaces. Every such improvement is, however, a potential threat to the exact reproduction of an analysis written against the older behavior. A script names the packages it loads but almost never the versions it loaded them at. It therefore inherits whatever versions the next machine happens to provide.

The remedy for this one layer is a lockfile, a file that records the exact version of every package the analysis used so that the same set can be reinstalled later. Recording that file is the transition from L0 to L1, from locatable to package-pinned, and it is the whole business of this chapter. We shall be equally careful about the transition's limit. A lockfile pins the packages and nothing beneath them. An analysis at L1 is therefore pinned at the package layer and exposed at every layer below, a boundary that motivates the container of Chapter 7.

The chapter proceeds as follows. We begin with the three judgments that stay with the analyst, then trace the line from packrat to renv and examine what a lockfile records. We next work through the init, snapshot, and restore workflow, the dependency-triad invariant and the automated check that enforces it, and dated snapshots of a package repository. We then set out the limits of renv and some lighter alternatives. Lastly, we scan the cohort's own dependencies for real and consider how to collaborate with an LLM on this material.

Draft, November 13, 2026. Content may change.

Check 2: naming the failure

The student's code did not change, and the published table can no longer be regenerated. What single determinant changed, and what artifact would have prevented the drift?
The answer is on page 247.

6.3. The analyst's contribution

The tooling in this chapter automates a great deal. However, three judgments remain irreducibly the analyst's.

1. **Deciding when to snapshot, and what a snapshot claims.** A lockfile is a dated assertion that these versions produced this result. The analyst chooses the moment to make that assertion, typically once an analysis is stable, and must understand that a snapshot taken mid-exploration pins whatever half-installed state the library happens to be in.
2. **Distinguishing a result dependency from a development tool.** No scanner can know that `languageserver` powers the editor while `dplyr` powers the analysis. The analyst must keep the lockfile an honest, minimal record of what determines the result, and keep mere tooling out of it, a distinction we develop below and that the accompanying whitepaper treats at length.
3. **Matching the reproducibility claim to the layer actually pinned.** A lockfile secures the package layer and no other. Presenting a lockfile-equipped analysis as fully reproducible, when the R version, the operating system, and the system libraries remain unpinned, overstates what has been achieved, and only the analyst can prevent that.

The vocabulary of this chapter

- **Lockfile.** A machine-generated file, `renv.lock`, recording the R version, the package repositories, and the exact version and

Draft, November 13, 2026. Content may change.

source of every package the project uses, together with metadata copied from each package's `DESCRIPTION`.

- **Package library.** The directory into which R installs packages. A project-local library belongs to one project rather than being shared across all of a user's work.
- **Snapshot.** The act of scanning a project, resolving the versions currently installed, and writing them into the lockfile.
- **Restore.** The reverse act of reading the lockfile and reinstalling exactly the packages it records.
- **`DESCRIPTION`.** The human-authored file that declares a project's direct dependencies by role (`Imports`, `Suggests`, `Depends`), with optional version floors.
- **Package record.** The lockfile entry for one package: its name, version, and source, plus fields such as `Imports` and `License` copied from its `DESCRIPTION`. It contains no checksum of the package's files.
- **Package repository.** A server, such as CRAN or the Posit Package Manager, from which packages are installed.
- **Dated snapshot.** A view of a repository frozen as it stood on a particular calendar day, so that an install run today and one run next year fetch the same versions.

Check 3: tool versus judgment

`renv::snapshot()` can scan a project's code for packages and write them to the lockfile automatically. Name one of the three judgments above that this automation does not make for the analyst.
The answer is on page 247.

6.4. From packrat to renv

We begin with some history. The idea of a project-local library governed by a lockfile is not new to R. The `packrat` package, begun in 2013 and on CRAN from 2014, was the first widely adopted attempt to give each project its own isolated set of packages recorded in a manifest. Its mechanics were sound but its ergonomics were not, and it acquired a reputation

Draft, November 13, 2026. Content may change.

for surprising behavior that discouraged routine use. The `renv` package, introduced by Ushey & Wickham (2026) and now the community standard, superseded it in 2019. It kept the essential idea, a project-local library plus a lockfile. However, it rebuilt the surface around a small, predictable set of verbs, a human-readable JSON lockfile, and a shared package cache. The cache lets many projects share one physical copy of a given package version.

The design separates two things. The **library** is where packages physically live; the **lockfile** is the written record of what should live there. The two are kept in agreement by two complementary operations. A snapshot reads the library and updates the lockfile to match it; a restore reads the lockfile and updates the library to match it. Everything `renv` does is a variation on keeping these two representations in step.

Check 4: library or lockfile

A teammate deletes their project's library folder by accident, but the lockfile is untouched and committed. Which operation restores their ability to work, and which of the two representations was lost?
The answer is on page 248.

6.4.1. What the lockfile records

The lockfile is the artifact that does the pinning, so it repays close reading. The top of the file records the R version and the repositories in force. Below that, each package has a record giving its exact version and the source it came from. Fields copied from the package's own `DESCRIPTION` follow: its title, license, declared dependencies, whether it contains compiled code, and more. The excerpt below comes from a lockfile that `renv` 1.2.3 wrote on 2026-10-11, under R 4.6.1, for a one-line script that calls `library(dplyr)`. It is shown as a static block because `renv` does not run while this book renders. It is abbreviated only by deleting fields and records; nothing has been added or reordered. The record for `dplyr` omits, among others, its `Type`, `Authors@R`, `Description`, `URL`, and `Suggests` fields.

Draft, November 13, 2026. Content may change.

```
{
  "R": {
    "Version": "4.6.1",
    "Repositories": [
      {
        "Name": "CRAN",
        "URL": "https://packagemanager.posit.co/cran/2026-
        ↪  06-29"
      }
    ]
  },
  "Packages": {
    "dplyr": {
      "Package": "dplyr",
      "Version": "1.2.1",
      "Source": "Repository",
      "Title": "A Grammar of Data Manipulation",
      "License": "MIT + file LICENSE",
      "Depends": [
        "R (>= 4.1.0)"
      ],
      "Imports": [
        "cli (>= 3.6.2)",
        "generics",
        "glue (>= 1.3.2)",
        "lifecycle (>= 1.0.5)",
        "magrittr (>= 1.5)",
        "methods",
        "pillar (>= 1.9.0)",
        "R6",
        "rlang (>= 1.1.7)",
        "tibble (>= 3.2.0)",
        "tidyselect (>= 1.2.0)",
        "utils",
        "vctrs (>= 0.7.1)"
      ],
      "NeedsCompilation": "yes",
      "Repository": "CRAN"
```


Draft, November 13, 2026. Content may change.

```
    }
  }
}
```

Four features deserve comment. First, the version is recorded exactly, `1.2.1` and not `>= 1.2`, so there is no latitude for a later install to drift upward. The version floors inside `Imports`, such as `rlang (>= 1.1.7)`, are a different thing. They are `dplyr`'s own statement of what it tolerates, copied from its `DESCRIPTION`. The pin for `rlang` is the exact version in `rlang`'s own record.

Second, the source is named. A package that came from CRAN, from a Posit Package Manager mirror, from Bioconductor, or from a specific GitHub commit is recorded as such. By default `renv::restore()` looks for it in the same place, since the same version number can denote different code on different repositories. The `Repository` value is the registered name of the repository the package was drawn from, matching one of the names in the `Repositories` block above. This is why the excerpt records `CRAN` even though that entry points at a Posit mirror.

Third, the closure is complete. Every contributed package that `dplyr` needs, directly or through another package, has a record of its own: the full file also holds records for `rlang`, `vctrs`, `tibble`, and the rest, and one for `renv` itself. Base packages such as `methods` and `utils` come with R and have none.

Fourth, and easily misread, the record contains no checksum of the package's files. Lockfiles written by `renv` before version 1.1.0 carried a `Hash` field and a `Requirements` list, and older tutorials still show them. The change is recorded in the NEWS file that ships with `renv`. That `Hash` was a fingerprint of selected `DESCRIPTION` fields, which `renv` uses to key its package cache. It was never a hash of the installed files, so it could not confirm that a restored package was byte-identical to the one recorded. A lockfile in either format therefore cannot detect a repository that serves altered contents under an unchanged version string. That protection, such as it is, rests with the repository.

Draft, November 13, 2026. Content may change.

> **Pitfall**
>
> A reader who has worked with `renv` before 1.1.0 may look for the old `Hash` field and, finding it absent in a current lockfile, worry that integrity checking has been removed. Nothing has been lost, because the old `Hash` never checked file integrity in the first place. It only fingerprinted `DESCRIPTION` fields for the package cache. The actual pitfall is believing either format's record proves the installed bytes match what was recorded. Neither does, and a lockfile's protection against a repository silently serving different code under the same version string is exactly nil.

6.4.2. The init, snapshot, restore workflow

Three commands drive the ordinary lifecycle. None of them can execute while this book renders, so we show them as static R blocks. The reader runs them at an interactive prompt inside the project.

Initialization creates the project-local library and a first lockfile.

```
renv::init()
```

From this point, packages installed into the project go into its private library rather than the shared system-wide one. One project's `dplyr` upgrade therefore cannot disturb another's. As the analysis develops and new packages are installed, a snapshot writes the current state into the lockfile.

```
renv::snapshot()
```

The snapshot does not blindly record every installed package. It first scans the project's source for the packages the code actually references, through `library()`, `require()`, and `pkg::fun()` calls, and records those together with their transitive dependencies. A package sitting in the library but called nowhere in the code is, by default, left out.

Draft, November 13, 2026. Content may change.

On another machine, or the same machine a year later, a restore reads the lockfile and rebuilds the library to match.

```
renv::restore()
```

A collaborator who clones the repository and calls `renv::restore()` obtains the exact package set the analysis was written against, irrespective of what versions are current.

In the framework this book uses, this lifecycle is partly automated so that an analyst does not forget the snapshot. The project's generated `.Rprofile` installs an exit hook that snapshots the library when an interactive session ends, and restores missing packages when a session begins. A package installed during analysis is thus captured. The automation removes the two most common ways the library and the lockfile drift apart.

Verification

To see the pin actually hold: note the installed version of one package (`packageVersion('dplyr')`), delete the project library, run `renv::restore()`, and confirm the same version comes back. To make it fail on purpose, open `renv.lock` in a text editor, change one package's `Version` field to a release that does not exist, and rerun `renv::restore()`. It reports that the requested version cannot be found, rather than silently installing something else.

Check 5: snapshot versus restore

A colleague clones your project, opens R, and is puzzled that the packages are not yet installed. They ask whether they should run `renv::snapshot()` to get them. What do you advise, and why?
The answer is on page 248.

6.5. The dependency-triad invariant

A lockfile solves the pinning problem. However, it introduces a bookkeeping problem in its place: the lockfile can fall out of step with the code it is

supposed to describe. A contributor installs a package for an exploratory notebook and forgets to snapshot, so the code now uses a package the lockfile does not record. Or a package is dropped from the analysis but left in the project's `DESCRIPTION`, so the declared dependencies name something the code no longer touches. Either drift silently degrades the pin.

But how are we to know that the lockfile still describes the code? The framework this book uses formalizes the correct relationship as the dependency triad. Three sources of information describe the packages an analysis uses, and in a healthy compendium they satisfy a nesting relation.

- The **code** references some set of packages, through its `library()`, `require()`, and `pkg::fun()` calls.
- The **`DESCRIPTION`** declares a set of direct dependencies.
- The **`renv.lock`** records a set of pinned packages.

The invariant is the composition of two atomic inclusions, each set contained in the next:

$$\text{Code} \subseteq \text{DESCRIPTION} \subseteq \text{renv.lock}.$$

Read left to right, it says that every package the code calls must be declared in `DESCRIPTION`, and every declared dependency must be pinned in the lockfile. The lockfile is the largest set because it also records the transitive dependencies that the analyst never named directly. Base packages, which every R installation provides at the R version's own version, are excluded from the code set. Recommended packages such as `survival` also ship with R, but they are versioned independently of it and `renv` records them in the lockfile, so they stay in. The first inclusion therefore concerns every package that is not base.

The invariant needs one qualification. Its strict form, with `DESCRIPTION` as the binding middle term, holds cleanly for the R-package archetype. There `DESCRIPTION` is the package's declared dependency interface, and every package the code calls belongs there by definition. For a research or analysis compendium the middle term is looser. In that case `DESCRIPTION` declares only the API of the reusable code in `R/`, while packages used solely by analysis scripts or reports legitimately live in `renv.lock` without appearing in `DESCRIPTION`. The operative invariant for a compendium is therefore

Draft, November 13, 2026. Content may change.

the weaker Code ⊆ renv.lock. That is, the strict `DESCRIPTION` inclusion is exact for the package archetype but archetype-conditional in general. As a matter of placement an analysis-script dependency may still be listed in `DESCRIPTION` under `Suggests`. However, the framework does not enforce the strict `DESCRIPTION` inclusion as a gate on a compendium, precisely because that inclusion does not hold for one. The gate it enforces on a compendium is the single inclusion Code ⊆ renv.lock, that nothing the code uses goes unpinned. The strict double inclusion is applied only in the package archetype, where `DESCRIPTION` is the authoritative dependency interface.

Figure 6.2 draws the nesting, with the archetype qualification marked. A package may sit legitimately in any of the three bands; the only broken position is outside the lockfile altogether.

```
library(ggplot2)
library(tibble)

bands <- tibble(
  label = c('renv.lock: everything pinned',
            'DESCRIPTION: declared',
            'Code: actually called'),
  xmin = c(0.4, 1.5, 2.5), xmax = c(9.6, 8.5, 7.5),
  ymin = c(0.5, 1.3, 2.1), ymax = c(6.2, 5.4, 4.6),
  fill = c('#eef4f4', '#dcebea', '#c3ddda')
)

items <- tibble(
  x = c(5.0, 5.0, 2.6, 7.4),
  y = c(3.3, 1.75, 0.92, 0.92),
  lab = c('dplyr\n(called, declared, pinned)',
          'an unused declaration',
          'rlang, vctrs\n(transitive)',
          'ggplot2, used only by\nanalysis scripts')
)

ggplot() +
  geom_rect(data = bands,
            aes(xmin = xmin, xmax = xmax,
```


Draft, November 13, 2026. Content may change.

```
              ymin = ymin, ymax = ymax, fill = fill),
          color = 'grey40', linewidth = 0.4) +
scale_fill_identity() +
geom_text(data = bands, aes(x = xmin + 0.15, y = ymax -
 ↪  0.25,
                          label = label),
          hjust = 0, fontface = 'bold', size = 3.1) +
geom_text(data = items, aes(x = x, y = y, label = lab),
          size = 2.9, lineheight = 0.95, color =
           ↪  'grey20') +
annotate('point', x = 3.55, y = 0.02, shape = 4,
         size = 2.6, stroke = 1.1, color = '#a33') +
annotate('text', x = 3.75, y = 0.02, hjust = 0,
         label = 'survival: used but unpinned',
         size = 3.1, fontface = 'bold', color = '#a33') +
coord_cartesian(xlim = c(0.2, 9.8), ylim = c(-0.35, 6.4))
 ↪  +
theme_void()
```


Draft, November 13, 2026. Content may change.

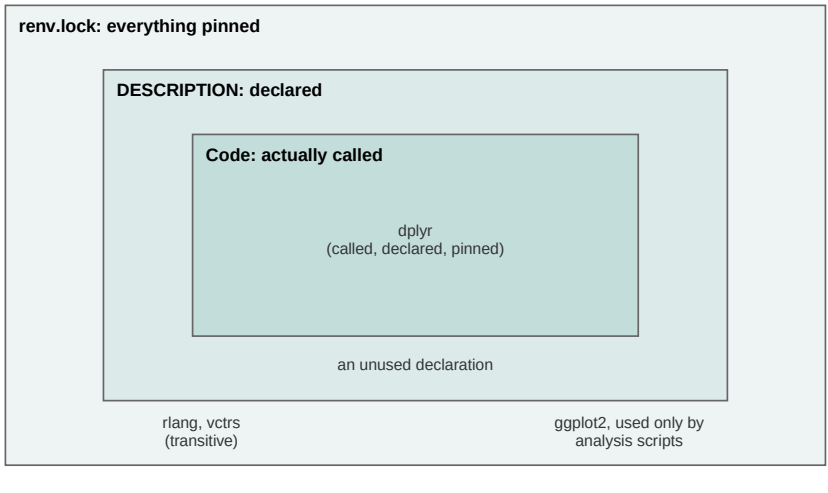


Figure 6.2.: The dependency triad as nested sets, with example packages placed in each region. Everything the code calls must be pinned, so a package outside the lockfile is the one broken position and is marked with a cross. The band between `DESCRIPTION` and the lockfile holds the transitive dependencies no analyst names directly, together with, in a compendium, the packages used only by analysis scripts. That second case is the archetype qualification discussed above: it is a violation in an R package and legitimate in a compendium.

The invariant can be checked by machine. The framework's `check-renv` gate, backed by the `zzrenvcheck` package, scans all three sources and tests the two inclusions. It reports any violation: a package used in code but undeclared, or declared but unpinned. It can also repair a violation on request. It writes a missing declaration into `DESCRIPTION` through the `desc` package. For a package absent from the lockfile, it writes a record of the version installed in the analyst's library, the version the code was actually run against. Only when the package is not installed at all does it fall back to the current CRAN version. It then prints a warning saying so, since that version may not be the one that produced any result. The

Draft, November 13, 2026. Content may change.

repair does not call `renv::snapshot()` or `install.packages()`; it edits the two manifests as text. Two limits follow. First, the record it writes is minimal, a name, version, and source without the copied `DESCRIPTION` metadata. Second, it covers the missing package alone, not that package's own dependencies. The repair is therefore a prompt to review, and a later `renv::snapshot()` remains the authoritative record of the library.

```
$ make check-renv
```

In the framework this book uses, this target runs the check with repair enabled before a commit. Outside the framework the same call is `zzrenvcheck::check_packages(auto_fix = TRUE)`. Under continuous integration the check must run with repair disabled and must be able to fail the build:

```
Rscript -e "zzrenvcheck::check_packages(auto_fix = FALSE,
 ↪  strict = TRUE, error_on_fail = TRUE)"
```

When `error_on_fail` is true, a failed check stops with an error and `Rscript` exits with a nonzero status. That is what turns the job red. Current `zzrenvcheck` already defaults to `error_on_fail = !interactive()`, so a script or CI job fails while an interactive console only reports. Nevertheless, the CI call above passes the argument explicitly so that the gate does not depend on how R was started. We confirmed the failure on a scratch project whose code called `dplyr` without declaring it: the command above printed 'Validation failed' and exited with status 1. `zzrenvcheck` is not on CRAN, so the CI job installs it first. The gate tool is itself a dependency of the evidence it produces, so the job pins it to one commit rather than to whatever the default branch holds on the day of the run:

```
remotes::install_github(
  'rgt47/zzrenvcheck@6f53235668cb4048c1ae793d64b10d6505bf6
   ↪  9a2'
)
```

That commit is `zzrenvcheck` 0.10.0, the version whose behavior this section describes. A drift that escaped the developer's machine therefore fails the

Draft, November 13, 2026. Content may change.

build rather than being silently repaired. This forces the contributor to commit the fix instead of letting the clean build paper over it. The gate is a presence check. That is, it guarantees that the three sets nest, that nothing used is unpinned. It does not verify that the version floors in `DESCRIPTION` agree with the exact pins in the lockfile, which is a separate concern.

The gate is also archetype-aware. When it finds an `analysis/` directory at the project root it enters compendium mode automatically (`compendium = "auto"` is the default). In that mode only packages used in `R/` are required in `DESCRIPTION`, while packages used only in `analysis/` or `scripts/` need only appear in `renv.lock`. On an R-package project, which has no `analysis/` directory, the strict double inclusion applies. Passing `compendium = TRUE` or `compendium = FALSE` overrides the auto-detection when the directory layout does not reflect the archetype.

6.5.1. Scanning the documentation, not only the code

The most common way for a dependency check to fail is subtle. It scans the analysis code, pronounces the project consistent, and overlooks a package that is used only in a vignette or in the rendered manuscript. That is exactly where a documentation build breaks on a fresh machine. `zzrenvcheck` deliberately scans more widely than a plain code scan on this point.

The scan does not look at `.R` scripts alone. It parses the literate and dynamic documents as well, R Markdown (`.Rmd`), Quarto (`.qmd`), and Sweave (`.Rnw`). A package loaded inside a report chunk or a vignette is thus seen just as one loaded in `R/` is. Within every such file it recognizes each of the forms by which R code reaches for a package. These are the explicit `library(pkg)` and `require(pkg)`, the namespaced `pkg::function()`, and the roxygen2 directives `@import` and `@importFrom` from which a package's `.Rd` help files are generated. In its strict mode it widens the search beyond the analysis directories, into the `tests/`, `vignettes/`, and `inst/` directories. It is correspondingly careful about what it ignores. `README` files and the example directories (`man/examples`, `inst/examples`) are skipped, since the install-and-demonstrate code they usually carry would otherwise raise false alarms. A separate pass reads the reproducibility files themselves, the `Dockerfile`, the `Makefile`, `install.sh`, and `.Rprofile`, for version-

Draft, November 13, 2026. Content may change.

pinned installs such as `pak('survival@3.5-8')`. This pass catches a version fixed in the container recipe that has drifted from the lockfile.

Consider a manuscript compendium whose analysis in `R/` restores cleanly from the lockfile. Its report, however, draws a forest plot with a plotting package that appears nowhere except the `.qmd` itself. A scanner that read only the code would certify the project as consistent. The report would then fail to render on the reviewer's machine for want of an unpinned package. Because `zzrenvcheck` reads the document, the missing dependency is caught and pinned before it can break the build. The wider reach helps most with the scholarly manuscript and the blog, whose rendered outputs are the deliverable. Their package needs sit in documents, where a naive check does not look.

Check 6: what a code-only scan misses

A manuscript's `R/` directory restores cleanly from the lockfile, but its `.qmd` report loads a plotting package nowhere referenced in `R/`. Would a dependency check that scans only `.R` files catch the missing pin, and why does `zzrenvcheck` catch it where that scan would not?
The answer is on page 248.

! The honest level

This chapter's own practice, the `check-renv` gate and the lockfile it enforces, reaches L1 package-pinned and nothing further. It leaves the R version, the operating system, and the system libraries beneath the package layer entirely exposed.

6.5.2. The triad across archetypes

The lockfile discipline is identical across the archetypes of Chapter 5, because every archetype records its packages in the same `renv.lock` by the same snapshot and restore. What varies slightly is how natural the declared-dependency middle term feels. In the R-package archetype, `DESCRIPTION` is the package's defining interface: `R CMD check` reads it, an installer reads it, and CRAN reads it. The triad is therefore especially natural there, since

Draft, November 13, 2026. Content may change.

the declared set already exists for reasons independent of reproducibility. A data-analysis project, a manuscript compendium, a simulation study, and a blog also carry a `DESCRIPTION`. There, however, it declares only the API of the reusable code in `R/`. Their binding invariant is instead the single Code $\subseteq$ renv.lock. The `check-renv` gate applies that weaker rule automatically when it detects an `analysis/` directory. A subtlety follows for the R-package archetype. A package used only by an analysis script or report belongs in `Suggests` rather than `Imports`. In that way `R CMD check` does not hard-require a package the package's own exported functions never call.

> Check 7: locating a drift
>
> Continuous integration on an R-package archetype fails with the message that `survival` is used in `R/model.R` but is not declared in `DESCRIPTION`. Which inclusion of the triad has been violated, and what single command would most likely repair it?
> *The answer is on page 248.*

6.6. Dated snapshots of a package repository

A lockfile records which versions to install. But a plain CRAN mirror will happily serve only whatever is current. Moreover, older source packages must often be recompiled, which is slow and can fail when a system library has since moved on. The Posit Package Manager resolves both problems by offering dated snapshots of CRAN. A snapshot URL freezes the repository as it stood on one calendar day. A request for a package then returns the version that was current on that date, and, on a platform the manager serves, returns it as a pre-compiled binary rather than as source to be built. On Linux that requires the URL form described below.

The mechanism is a URL that embeds a date.

```
options(repos = c(
  CRAN = "https://packagemanager.posit.co/cran/2026-06-29"
))
```


Draft, November 13, 2026. Content may change.

With repositories pointed at a dated snapshot, an install run today and the same install run next year draw from an identical, frozen catalog. The lockfile above records exactly this URL in its `Repositories` block, so that a restore reproduces both the versions and the source they came from. The gain is twofold. The first gain is fixity, because the catalog cannot drift, and the second is speed, because binaries install in seconds where source builds take minutes. On Linux the binary path is more specific. Precompiled binaries are served from a distribution-specific form of the URL, such as `https://packagemanager.posit.co/cran/__linux__/noble/2026-06-29`, and R may need to send its version in the HTTP user agent. The plain dated path shown above should be expected to deliver source to be built. In the framework this book uses, the project's generated configuration points R at a dated snapshot automatically. The two benefits thus accrue without the analyst assembling the URL by hand. The date of the snapshot becomes, in effect, part of the environment's identity. We take up this topic again when the container fixes a snapshot date at image-build time in Chapter 7.

Check 8: fixity versus speed

A colleague points their `repos` option at `https://packagemanager.posit.co/cran/2026-06-29` and reports that installs are both faster and identical a year later. Are these the same benefit or two different ones, and what does each depend on?
The answer is on page 249.

6.7. The limits of renv

A lockfile is easy to overclaim, and the gap it leaves is the subject of the next chapter. But what, exactly, does a lockfile leave unpinned? `renv` pins the R packages. It does not pin the R interpreter itself, so a lockfile produced under R 4.6.0 may restore under R 4.2.1 with different results. It may also fail to restore at all when a recorded version requires a newer R. It does not pin the operating system, so a lockfile assembled on macOS carries no guarantee on Linux. And it does not pin the system libraries that many packages link against, the GDAL behind a spatial package or the `libxml2` behind an XML parser. A restore on a machine lacking a required system

Draft, November 13, 2026. Content may change.

library may therefore fail to compile a package or, more insidiously, may install a differently behaving one.

`renv` does its one job well, but the package layer is one layer of a taller stack. Pinning it leaves every layer beneath it as the next machine happens to provide. Ushey & Wickham (2026) designed `renv` to solve the package problem and no more; the environment problem is out of its scope by design. An analysis equipped with a faithful lockfile has reached L1, pinned packages, and every layer beneath the packages remains exposed. Closing that remaining gap is the work of the container, and it is where Chapter 7 begins.

Check 9: what a lockfile cannot promise

A lockfile produced under R 4.6.0 is restored on a colleague's machine running R 4.2.1, and the restore succeeds without error. Does this mean the analysis will produce the same result on both machines?
The answer is on page 249.

6.8. Lighter and alternative approaches

The lockfile-and-restore idiom is not the only way to pin the package layer. The reader should know the principal alternatives, because some suit particular settings better and because the tradeoffs illuminate what `renv` is doing.

The most important shift of idiom is from preserving a lockfile to reconstructing an environment from a date. The `rang` package of C. Chan & Schoch (2023) takes a set of packages and a target date, and resolves the versions that were current then. It reaches back through CRAN's archive to reconstruct a historical environment, even for a project that never recorded a lockfile at all. Where `renv` requires foresight, a committed lockfile written while the analysis was live, `rang` offers hindsight, recovering an approximate environment after the fact. It is the natural instrument for a legacy analysis that one wishes to resurrect.

The `groundhog` package of Simonsohn (2026) pursues the same date-based idea but at a different point in the workflow. Rather than restoring a library

Draft, November 13, 2026. Content may change.

before the script runs, it loads version-specific packages at run time. A `groundhog.library()` call carries a date, and the packages current on that date are loaded for that session. It needs no project library and no snapshot step. In effect, it trades `renv`'s explicit committed lockfile for a date-stamped call evaluated against the repository's history. The cost of that convenience is that the pin now lives inside the script's control flow rather than in a separate, inspectable manifest. A reviewer must read the code to discover it.

Finally, `capsule` (McBain, 2023) is best understood as a lazier discipline layered over `renv` itself. It defers building the project library until it is actually needed. The analyst develops against their ordinary user library and materializes the pinned library only when reproducing or checking the work. For analysts who find `renv`'s always-on project library intrusive during exploratory work, it lowers the friction of adoption while preserving the lockfile as the eventual source of truth.

Each of these approaches pins the package layer and only the package layer. Whether the pin is a committed lockfile, a reconstructed date, or a run-time call, none reaches down to the R version, the operating system, or the system libraries. The choice among them is a choice of ergonomics and of foresight versus hindsight, not a choice of how far down the determinant stack one has pinned.

> Check 10: foresight or hindsight
>
> A legacy analysis from four years ago never recorded a lockfile. Which of the three alternatives is suited to recovering an approximate environment for it now, and which two require the discipline to have been in place from the start?
> *The answer is on page 249.*

6.9. Worked example

To make the package-churn failure mode concrete, and to generate a computed artifact at build time, we construct a small synthetic lockfile diff. That is, we compare two snapshots of the same project taken some months apart. This is the kind of table a version-control diff of `renv.lock` would

Draft, November 13, 2026. Content may change.

summarize. Reading it is the routine skill of noticing which pins moved. The data are synthetic and no packages are installed; the computation uses only `tibble`, `dplyr`, and `knitr`.

```
library(tibble)
library(dplyr)

lock_diff <- tibble(
  package    = c('dplyr', 'ggplot2', 'tidyr',
                 'survival', 'lubridate', 'rlang'),
  version_a  = c('1.1.0', '3.4.4', '1.3.0',
                 '3.5-7', '1.9.3', '1.1.2'),
  version_b  = c('1.2.1', '3.4.4', '1.3.1',
                 '3.5-8', '1.9.3', '1.1.4')
) |>
  mutate(changed = if_else(
    version_a == version_b, '', 'changed'
  ))

knitr::kable(lock_diff)
```

Table 6.1.: A synthetic lockfile diff between two snapshots of one project. The changed column flags every package whose pinned version moved between the earlier and later snapshot.

package	version_a	version_b	changed
dplyr	1.1.0	1.2.1	changed
ggplot2	3.4.4	3.4.4	
tidyr	1.3.0	1.3.1	changed
survival	3.5-7	3.5-8	changed
lubridate	1.9.3	1.9.3	
rlang	1.1.2	1.1.4	changed

Between the two snapshots, 4 of the 6 pins moved. Three of the moves are patch-level (`tidyr`, `survival`, and `rlang`), the kind that ordinarily preserve behavior but occasionally do not. One is a minor version bump

Draft, November 13, 2026. Content may change.

(`dplyr`, from 1.1.0 to 1.2.1) of the sort that can refine a default. The `rlang` move, from 1.1.2 to 1.1.4, deserves attention despite its size because of its transitive position. Almost nothing calls `rlang` directly, yet almost everything depends on it, so the move happened beneath the whole stack. A diff of the lockfile is a reviewable record of this churn. Without a committed lockfile, none of these movements would be visible, and the analyst confronted with a changed result would have no way to see which package had shifted. The lockfile cannot stop packages from changing, but it makes the change legible and lets the analyst restore the earlier column deliberately.

> Check 11: reading the diff
>
> In Table 6.1, `ggplot2`'s version is identical in both snapshots and `rlang`'s moved by only a patch release, yet the chapter singles out `rlang`'s move as the more consequential one to notice. Why?
> *The answer is on page 249.*

6.10. The cohort's own dependencies

In this section we apply the chapter's method to the running cohort compendium. `renv::dependencies()` scans a file for the packages its code actually references and returns one row per package found. That is exactly the scan `renv::snapshot()` runs before writing a lockfile.

```
deps <- renv::dependencies(
  c('../../R/cohort.R',
    '../../analysis/scripts/simulate_cohort.R')
)
#> Finding R package dependencies ... Done!
deps_tbl <- unique(deps[, c('Source', 'Package')])
deps_tbl$Source <- basename(deps_tbl$Source)
found_pkgs <- sort(unique(deps$Package))
base_pkgs <- rownames(installed.packages(priority =
 ↪  'base'))
found_base <- intersect(found_pkgs, base_pkgs)
```


Draft, November 13, 2026. Content may change.

```
contributed <- setdiff(found_pkgs, base_pkgs)
has_lockfile <- file.exists('../../renv.lock')
lock_pkgs <- if (has_lockfile) {
  names(renv::lockfile_read('../../renv.lock')$Packages)
} else {
  character(0)
}
unpinned <- setdiff(contributed, lock_pkgs)
knitr::kable(deps_tbl, row.names = FALSE)
```

Table 6.2.: Packages `renv::dependencies()` finds referenced in the two files that produce the running cohort, scanned directly rather than asserted. `simulate_cohort.R` contributes no rows because it calls base R functions (`rnorm`, `sample`, `as.Date`) without `library()` or a `::` prefix, so the scan finds no package name to report.

Source	Package
cohort.R	parallel
cohort.R	stats
cohort.R	survival
cohort.R	tibble
cohort.R	utils
cohort.R	withr

`R/cohort.R` references `parallel`, `stats`, `survival`, `tibble`, `utils`, and `withr`. Of these, `parallel`, `stats`, and `utils` are base packages: they ship with every R installation, their version is the R version's, and a lockfile records no entry for them. The counting rule for the rest is whether `renv` records the package in the lockfile. A recommended package such as `survival` also ships with the standard R binaries, but it is versioned and updated independently of R, so `renv` pins it like any other, and we count it as contributed. Under that rule the scan finds 3 contributed packages, `survival`, `tibble`, and `withr`. The code is locatable and its dependencies are *known*, from a real scan rather than from memory. Whether they are also pinned is a question for the lockfile. `file.exists('renv.lock')` at the

Draft, November 13, 2026. Content may change.

project root is TRUE, and of the contributed packages the scan finds, the lockfile leaves 0 unrecorded. The package-versions layer is therefore pinned, and the cohort compendium has reached at least L1. For a compendium without a lockfile, `renv::init()`, which scans the code and writes the first lockfile, and a commit of that lockfile, is the single step that lifts it from L0 to L1, exactly as the orientation above describes for the hypothetical registry study.

Check 12: why no rows for the simulator

`renv::dependencies()` returns no row for `analysis/scripts/simulate_cohort.R`, even though that script is essential to the cohort and uses functions such as `rnorm()` and `sample()`. Is this a bug in the scan?
The answer is on page 250.

6.11. Collaborating with an LLM

A language model is a capable assistant for the mechanics of `renv`. However, a fluent, plausible-looking lockfile instruction can pin the wrong thing or nothing at all. Each of the following triples pairs a useful prompt with what the analyst must watch for and verify.

! Restricted data

One rule comes first (habit 3). Never paste or upload protected health information, identifiable records, or data held under a data use agreement, nor outputs derived from them, into a model or a coding agent unless three conditions all hold. The institution has approved that endpoint for that class of data. The IRB protocol and the consent permit the processing. The data use agreement, or the provider's terms, permits it. If any of the three is unclear, the answer is no.
The third condition is the one most often missed. Institutional approval of a tool does not override the terms under which a data provider released the data. NIH, for example, treats sharing controlled-access data through prompts with public generative AI tools, which it glosses

Draft, November 13, 2026. Content may change.

as third-party tools, as a breach of the data use certification. It treats a model trained on such data as a data derivative, to be shared only with other approved users and not kept after the project closes (National Institutes of Health, 2025a).
Describe the structure instead: the column names and types, or a small synthetic example generated from the data dictionary, never edited from real rows. A coding agent sees whatever files are in its working tree, so run it in a checkout or container where the real data is not mounted.
An agent also acts, so confine it to a branch and keep merges, pushes, and deposits for yourself (Section 2.10.1).

Dependency questions rarely need data at all.

Prompt. 'My collaborator cloned our project and the packages are missing. Write the R command they should run.'

Watch for. Check for advice to reinstall packages by hand with `install.packages()`, or to run `renv::snapshot()`, either of which is wrong for a fresh clone. The correct answer is `renv::restore()`.

Verification. Confirm that the suggested command reads the lockfile rather than writing it, and that after running it the installed versions match the lockfile. Run the check gate and confirm the triad holds.

With a coding agent the suggestion becomes an action. An agent that runs `renv::snapshot()` in a fresh clone can overwrite the lockfile with the state of whatever library it finds there. Treat `renv.lock` as a protected path: read its diff line by line before any commit that touches it, and keep the decision to change it with the analyst (Section 2.10.1).

Prompt. 'Explain why my `renv.lock` includes `roxygen2` and `tinytest` when my analysis never calls them, and whether I should remove them.'

Watch for. Check for a flat assertion that any unused package should be purged, which misses the deliberate convention by which test infrastructure is pinned while pure development tooling is not. The distinction is a judgment, not a rule the model can read off the file.

Verification. Decide the placement yourself against the two-test procedure: does the package's version affect the result, and does the code reference it?

Draft, November 13, 2026. Content may change.

Confirm that development-only tools such as `languageserver` are in the Dockerfile and absent from the lockfile, and that test packages are handled by your stated convention.

Prompt. 'Set my repository to a dated Posit Package Manager snapshot for 2026-06-29.'

Watch for. Check the URL for the wrong path structure, an implausible date, or a mirror that does not offer binaries for the analyst's platform. Check also for a Linux distribution codename assumed without comment that does not match the analyst's system.

Verification. Check the constructed URL against the Posit Package Manager documentation, confirm the date is the one you intend, and confirm that an install actually draws binaries rather than falling back to source. The recorded URL should then appear in the lockfile's `Repositories` block.

A better prompt. 'My analysis runs on Ubuntu 24.04 (codename `noble`) with the R version recorded in the attached `renv.lock`. Write the `options(repos = ...)` call that points at the Posit Package Manager CRAN snapshot dated 2026-06-29, in the form that serves Linux binaries for that codename. Change no package version. Tell me how to confirm that an install draws a binary rather than building from source.'

6.12. Exercises

1. Initialize `renv` in a throwaway project, install two packages, and snapshot. Open `renv.lock` in a text editor and locate, for one package, its `Version`, its `Source` and `Repository`, and its `Imports`. Explain in a sentence what each tells a later restore, and name one thing the record does not let a restore check. Report the `renv` version you used, since the lockfile format changed in `renv` 1.1.0.

2. With the project of Exercise 1, upgrade one package to a newer version, snapshot again, and inspect the diff of `renv.lock` under version control. Which fields changed, and which stayed fixed? Relate the diff to the synthetic table of Table 6.1.

Draft, November 13, 2026. Content may change.

3. Deliberately break the dependency triad: add a `library()` call for a package you have not declared or pinned. Run the project's dependency check and record which inclusion it reports as violated and how it repairs the violation. Then remove the call and confirm the check passes.

4. Point your repository at a dated Posit Package Manager snapshot and reinstall one package. Time the install, then compare it with an install of the same package from source. Report the difference and explain, in terms of binaries versus source, why it arises.

5. Take a small analysis that has no lockfile and use `rang` to reconstruct the package environment as it stood on a chosen past date. Compare the resulting version set with what a current `install.packages()` would provide. Which packages differ, and by how much?

6. Argue, in a short paragraph, why an analysis that passes its dependency check has reached L1 and not L2. Name one concrete determinant that a lockfile leaves unpinned and describe a plausible way it could change a result across two machines.

7. Run `renv::dependencies()` on `R/cohort.R` and `analysis/scripts/simulate_cohort.R` from the running cohort compendium, as Table 6.2 does. Report the number of distinct contributed (non-base) packages the scan finds and name them, counting a recommended package such as `survival` as contributed because `renv` pins it, and treating a package as base when `installed.packages(priority = 'base')` lists it. Then explain, from the scan alone and without reading `simulate_cohort.R`, why that file is permitted to contribute zero rows.

8. This exercise returns to the version-control chapter's discipline of distinguishing what must be committed from what must never be. `renv::init()` writes an `renv/` directory containing `activate.R`, a `settings.json`, and (once packages are installed) a `library/` subdirectory holding the actual installed package files. State which of these `renv` itself places in a generated `.gitignore`, and explain why committing the `library/` subdirectory would defeat the purpose of the lockfile rather than merely waste space.

Draft, November 13, 2026. Content may change.

6.13. Solutions

Attempt an exercise before expanding its solution. Two of the eight have no single correct answer, because what they report depends on the machine and the date. For those the box says what a good answer establishes rather than supplying a number. Exercise 7 does have a checkable numeric answer, run against the extract as committed.

Exercise 1

The answer below assumes `renv` 1.1.0 or later. With `renv` 1.2.3, the version used for this chapter, a snapshot writes records in the format excerpted in the chapter. The fields answer different questions a later restore has to settle.

The **version** is recorded exactly, `1.2.1` and not `>= 1.2`, so a restore installs the release the analysis actually used and has no latitude to drift upward. This is the field that defends against the ordinary failure, in which a package acquires a new default between the original run and the rerun.

The **source** names where the package came from: CRAN, a Posit Package Manager mirror, Bioconductor, or a specific GitHub commit. It matters because the same version number can denote different code on different repositories, so the version alone does not identify the package. Note that the `Repository` value is the registered name of the repository rather than its URL. That is why a lockfile can record `CRAN` while the corresponding entry in the `Repositories` block points at a Posit mirror.

The **`Imports`** list is copied from the package's own `DESCRIPTION`. It tells a reader which other packages this one needs, with the version floors its authors declared, such as `rlang (>= 1.1.7)`. It is not itself a pin: each of those dependencies is pinned by the exact `Version` in its own record elsewhere in the file.

What the record does not let a restore check is whether the files it installs are the files that were installed before. There is no checksum of the package contents. A reader working from an older tutorial may expect a `Hash` field. `renv` before 1.1.0 wrote one, but it was a fingerprint of `DESCRIPTION` fields used to key the package cache, never a hash of the installed files. Current `renv` does not write it.

Draft, November 13, 2026. Content may change.

A useful way to hold the fields together: the version says which release, the source says whose copy of it, and the copied metadata says what that release declared about itself. None of them says whether the bytes match.

Exercise 2

For the upgraded package, `Version` changes, and so does any field copied from `DESCRIPTION` that the new release altered: commonly `Imports` and its version floors, the `RoxygenNote`, and the author or maintainer fields. A diff of one upgrade can therefore show many changed lines for a single moved pin. The line to look for first is `Version`. `Package`, `Source`, and `Repository` stay fixed, since the package is the same package drawn from the same place. A reader whose lockfile shows a `Hash` field moving alongside the version is running `renv` older than 1.1.0, which wrote the older format.

Two further effects in the diff are easy to miss on a first reading. If the upgraded package gained a dependency, new package entries appear in the lockfile that the reader never installed by name. This is because the lockfile captures the full transitive closure rather than the packages the reader asked for. And the `R` block at the top of the file is untouched. It records the R version under which the snapshot ran, but it does not install or enforce that version (Exercise 6), and upgrading a package does not change it.

The relation to Table 6.1 is that the table shows the same structure in synthetic form. Each row is a package, the earlier and later columns hold the pinned versions, and the flag marks where they differ. A real `git diff renv.lock` is that table expressed as JSON, with the additional detail that the copied `DESCRIPTION` metadata moves with the version.

Exercise 3

The violated inclusion is the first atomic one, Code $\subseteq$ DESCRIPTION. The code now references a package that `DESCRIPTION` does not declare. This in turn leaves it unpinned in the lockfile, so the transitive composition Code $\subseteq$ renv.lock fails as a consequence. The breach to

Draft, November 13, 2026. Content may change.

report is the atomic one, not the consequence.
Running `make check-renv` invokes `zzrenvcheck::check_packages()` with automatic repair. It writes the package into `DESCRIPTION` through the `desc` package and adds a record of its installed version to `renv.lock`, restoring both inclusions. If the package is not installed, the repair records the current CRAN version instead and warns that it has done so. The honest response to that warning is to install the package, rerun the analysis, and repair again. This edits the two manifests as text and is deliberately distinct from an `renv::snapshot()`, which would instead re-derive the whole lockfile from whatever is currently installed.
Removing the `library()` call and re-running the check does make the check pass. However, it does not return the manifests to their prior state: the repair already wrote the package into `DESCRIPTION` and the lockfile, and nothing removes it. A passing check means the inclusions hold, not that the project is unchanged. Restoring the prior state requires deleting the entries by hand or discarding the changes under version control, which is itself a small argument for making the repair on a branch.

Exercise 4

This exercise has no single correct answer, because the times depend on the package, the machine, and the network. What a good answer establishes is the mechanism, and it should report both timings and name the package.
The expected result is that the snapshot install is substantially faster. The explanation is that Posit Package Manager serves a prebuilt binary for the declared platform. Installing a binary is unpacking an archive; installing from source is compiling it. For a package carrying compiled code, typically anything depending on `Rcpp`, the difference is commonly an order of magnitude or more.
The instructive case is the one that does not show a large difference. A pure-R package has little to compile, so the two paths do similar work and the timings converge. A reader who happens to pick such a package and reports a small difference has the right finding, provided they identify why. Two other conditions can flatten the difference and

Draft, November 13, 2026. Content may change.

are worth checking before concluding anything. First, the `renv` cache may already hold the package, in which case neither install does real work. Second, PPM does not serve binaries for every platform, so a machine whose platform is unserved falls back to source in both arms.

Exercise 5

This exercise also has no single correct answer, since what differs depends on the date chosen and on the size of the dependency tree. A good answer reports the count of differing packages, gives one or two concrete version pairs, and draws the general conclusion.

The general conclusion is that `rang` reconstructs what was *current on a date*, which is not the same thing as what an analysis *actually used*. The two coincide only if the original analyst installed everything on that date and never updated anything afterward. That is not how most projects proceed. Reconstruction by date is therefore an approximation, and a lockfile is a record; the approximation is what one falls back on when the record does not exist.

The number of differing packages will generally grow with the age of the date and with the depth of the dependency tree, since a difference anywhere in the transitive closure counts. A package whose version happens to be unchanged is not evidence that the reconstruction succeeded, only that that particular package was quiet over the interval.

Exercise 6

A passing dependency check establishes that the triad holds: every package the code uses is declared, and every declared package is pinned in the lockfile. That is exactly the L1 property, pinned packages, and the check is the evidence for it.

It is not L2, because the lockfile pins packages and nothing beneath them. The R version is recorded in the lockfile but not supplied by it. The operating system, the system libraries, the locale, the time zone, and the numerical libraries are all outside its scope. L2 requires an artifact that fixes the operating system, the R runtime, and the system libraries, which is what Chapter 7 is about. Even L2 does not

Draft, November 13, 2026. Content may change.

guarantee the numerical libraries: an image determines which BLAS is installed, but the thread count and the CPU still vary.
For a concrete unpinned determinant, the numerical-library layer is the cleanest example of what a lockfile misses, though no rung of the ladder fixes it completely. Two machines can hold identical pinned package sets and still link against different BLAS implementations, reference BLAS on one and Apple's Accelerate or Intel MKL on the other. Because these reorder the floating-point summations inside a matrix decomposition, a principal-components analysis or a mixed-model fit can return results that agree to several decimal places and differ in the last. A sign flip on a component can make the difference visible in a plot. The lockfile is silent on all of it, and setting a seed does not help, because the difference is not stochastic.
A second acceptable answer names the locale. Two machines under different collation orders sort character vectors differently. This changes which row is taken first in a tie and can therefore change a reported result through a `slice()` or a `first()`, with no error raised anywhere.

Exercise 7

The solution for this exercise is in the instructor guide, which is distributed separately to course instructors. Self-study readers who want to check their answer can compare the autogradable values against those reported in the chapter's own 'The cohort's own dependencies' section.

Exercise 8

`renv::init()` writes a generated `.gitignore` inside `renv/`. Under `renv` 1.2.3's default settings it excludes `library/`, `local/` (local package sources), `cellar/`, `lock/`, `python/`, `sandbox/`, and `staging/`. It leaves `renv/activate.R` and `renv/settings.json` to be committed, along with the lockfile `renv.lock` at the project root. The rule follows the same line the version-control chapter draws between what must be committed and what must never be. Here it is applied to a different kind of large, regenerable, and machine-specific content

Draft, November 13, 2026. Content may change.

rather than to restricted data.
Committing `library/` would not merely waste space; it would defeat the lockfile's purpose outright. `renv.lock` exists to be a small, portable, human-readable description from which the library can be *rebuilt* on any machine. If the library itself were committed, a collaborator on a different operating system or architecture would receive compiled binaries built for someone else's platform. Because their versions match the lockfile, `renv::restore()` would likely treat them as already installed, and a package with compiled code would then fail when it loads. Moreover, the repository would grow by orders of magnitude for content that `renv::restore()` already knows how to regenerate from the one file that actually matters.

Answers to the checks

Check 1. *from earlier chapters* (page 214)

The cleaned cohort table with fewer rows exposes a stale derived file, not a package or environment change. On one machine the old script reproduces the old count and the current script does not, so only the analysis code differs. The committed file breaks the promise of Chapter 5 that a file in `derived_data/` can always be regenerated from the raw inputs and the code. The history of Chapter 4 locates the change: `git log` on the script lists the commit, and `git show` displays its diff. The tag secures L0 on the ladder of Chapter 3. It fixes which code a reader finds, not that a stored output still matches that code. The remedy is to regenerate the file from the raw extract and the current script.

Check 2. *naming the failure* (page 217)

The package-versions layer changed. `dplyr` moved from 1.1.0 to 1.2.1 between the original run and the revision, and a refined grouping default altered an intermediate summary. A committed `renv.lock`, restored rather than freshly installed, would have reinstalled the exact version the student originally ran against. That would have closed the gap the bare script left open.

Check 3. *tool versus judgment* (page 218)

Draft, November 13, 2026. Content may change.

Any one of the three. The first is the moment at which a snapshot is taken, and therefore what state of the library it asserts as correct. The second is the distinction between a package the result depends on and a package that merely serves the analyst's editor or workflow. The third is the honest match between the reproducibility claim made and the layer the lockfile actually pins. The scan makes none of these three decisions.

Check 4. *library or lockfile* (page 219)

`renv::restore()` restores it, by reading the lockfile and reinstalling the packages it records into a fresh library. What was lost was the library, where packages physically live. The lockfile, the written record of what should live there, was never at risk, because it is a small committed text file rather than a directory of installed code.

Check 5. *snapshot versus restore* (page 223)

They should run `renv::restore()`, not `renv::snapshot()`. Restore reads the existing lockfile and installs the recorded versions into the local library, which is exactly what a fresh clone needs. Snapshot runs in the opposite direction. It would overwrite the lockfile with whatever is currently installed, which on a fresh clone is nothing relevant. It would thereby destroy the very record the colleague needs. The mnemonic is that snapshot writes the lockfile and restore reads it.

Check 6. *what a code-only scan misses* (page 230)

No. A code-only scan sees only `R/`, finds every package it references already pinned, and certifies the project consistent while the report's own dependency goes unseen. `zzrenvcheck` catches it because it also parses literate documents, `.Rmd` and `.qmd` among them, for `library()`, `require()`, and `pkg::fun()` calls. A package loaded only inside a report chunk is therefore scanned exactly as one loaded in `R/` would be.

Check 7. *locating a drift* (page 231)

The invariant is the composition of two atomic inclusions. The one violated here is the first, Code $\subseteq$ DESCRIPTION: the code references `survival` while `DESCRIPTION` does not declare it. This in turn leaves it unpinned in the lockfile. The transitive composition Code $\subseteq$ renv.lock therefore fails as a consequence, but the atomic breach is at the code-to-`DESCRIPTION` step. The likely cause is that `survival` was installed interactively but

Draft, November 13, 2026. Content may change.

never declared. Running `make check-renv`, which invokes `zzrenvcheck::check_packages()` with automatic repair, would write `survival` into `DESCRIPTION` through the `desc` package. It would also add a record of the installed `survival` version to `renv.lock`, restoring both inclusions. This edits the two manifests as text and is distinct from an `renv::snapshot()`, which would instead re-derive the lockfile from the installed library. The fix must then be committed so the clean CI build sees it.

Check 8. *fixity versus speed* (page 232)

Two different benefits. The speed comes from the snapshot serving a pre-compiled binary rather than source to be built. It depends on the platform-specific binary path being used (the distribution-codename form on Linux) and on a binary existing for that platform. The identical-a-year-later fixity comes from the date embedded in the URL freezing the catalog. It holds regardless of whether a binary or a source install is served. A snapshot with no binaries for a given platform still delivers the fixity without the speed.

Check 9. *what a lockfile cannot promise* (page 233)

Not necessarily. `renv` pins the package versions, not the R interpreter itself. A successful restore under a different R version means the packages installed, not that they behave identically under that R. The lockfile's `R` block records which R version wrote it, but it neither installs nor enforces that version. When the project loads under a different minor release of R, `renv` prints a one-line notice naming both versions, and then proceeds under the R it was given. Closing this exposure is the work of the container in Chapter 7, which pins the R runtime along with the packages.

Check 10. *foresight or hindsight* (page 234)

`rang` is suited to the legacy case, because it reconstructs a past environment from a target date through CRAN's archive, needing no prior lockfile at all. `groundhog` and `capsule` both require the discipline in place beforehand. `groundhog` needs the date-stamped `groundhog.library()` call written into the script at the time. `capsule` still layers over `renv`'s own snapshot, so without one there is nothing for it to materialize later.

Check 11. *reading the diff* (page 236)

Draft, November 13, 2026. Content may change.

Because consequence tracks transitive position, not the size of the version jump. `rlang` sits beneath almost every other package in the tidyverse closure. A change to its behavior therefore propagates upward into everything that depends on it, even though almost nothing in the project's own code calls `rlang` directly. A patch-level move in a package most analysts have never heard of can matter more than a minor-version move in a package they call by name every day.

Check 12. *why no rows for the simulator* (page 238)

No. `rnorm()` lives in the `stats` package and `sample()` in the `base` package. Both, like `utils`, are base packages bundled with every R installation rather than contributed packages a lockfile needs to pin. The scan is working as intended. It reports the packages a file names, through `library()`, `require()`, or `pkg::`, and this script names none. A base package named that way would appear, as `stats` and `utils` do for `R/cohort.R`, but the lockfile would still record no entry for it.

6.14. Cheat sheet

Command or rule	What it does
`renv::init()`	Creates the project-local library and a first `renv.lock`.
`renv::snapshot()`	Scans the code for referenced packages and writes their installed versions to `renv.lock`.
`renv::restore()`	Reads `renv.lock` and reinstalls exactly the versions it records.
`renv::dependencies(files)`	Scans one or more files for the packages they name (`library()`, `::`, and similar), base packages included, without touching the lockfile.
`make check-renv` / `zzrenvcheck::check_packages(auto_fix = TRUE)`	Checks and repairs the code-to-`DESCRIPTION`-to-lockfile triad.

Draft, November 13, 2026. Content may change.

Command or rule	What it does
`zzrenvcheck::check_packages( auto_fix = FALSE, strict = TRUE, error_on_fail = TRUE)`	The CI form: checks only, and fails the build on a violation.
`options(repos = c(CRAN = '.../cran/YYYY-MM-DD'))`	Points installs at a dated, fixed snapshot of the repository.
Rule of thumb	A lockfile pins the package layer and nothing beneath it; L1, not L2.
Rule of thumb	An oracle, a declared dependency, or a pinned version must come from outside the thing it checks; a lockfile is no exception for file integrity.
Habit 3	Never let restricted data reach an unapproved model endpoint, even to ask a dependency question.

6.15. Further reading

The authoritative reference for the tool is Ushey & Wickham (2026), the package documentation for `renv`. Its introductory and collaboration vignettes are the fastest route to fluent daily use, and its treatment of the cache and of custom repositories repays study once the basics are in hand. The glossary entry `renv` and `packrat` summarizes the relationship of `renv` to its predecessor.

For the date-based alternatives, C. Chan & Schoch (2023) presents `rang` and its strategy of reconstructing a historical environment from CRAN's archive. It gives a clear account of the cases where hindsight succeeds and where it cannot. Simonsohn (2026) documents `groundhog` and its run-time, date-stamped loading model, and McBain (2023) describes the lazy discipline that `capsule` layers over `renv`. Reading the three together clarifies that pinning the package layer admits several idioms with different demands on foresight.

On the discipline of where a package belongs, the placement whitepaper accompanying the framework develops the two-test decision procedure and the two-layer model that this chapter summarizes. It is the place to turn

Draft, November 13, 2026. Content may change.

when a package sits awkwardly between the lockfile and the Dockerfile. Finally, the boundary this chapter insists on, that the lockfile pins the packages and nothing beneath them, is taken up directly in Chapter 7. There the container fixes the R version, the operating system, and the system libraries that `renv` by design does not.

Draft, November 13, 2026. Content may change.

7. Pinning the Environment with Docker

Though computational reproducibility seems more straight forward than replicating physical experiments, the complex and rapidly changing nature of computer environments makes being able to reproduce and extend such work a serious challenge.

Carl Boettiger, *An Introduction to Docker for Reproducible Research* (2015)

By the end of this chapter you will be able to write a Dockerfile that reaches L2, pinned environment, rather than one that merely looks reproducible. Its base will be named by an immutable digest, and its installer will fetch from a dated snapshot of the package repository. Its locale, time zone, and thread count will be declared in the recipe instead of inherited from whichever machine happens to run it.

Before you start

Reading time. About 55 minutes.
Prerequisites. Chapter 3, for the L0-L3 ladder and the capture-versus-validation vocabulary this chapter relies on throughout; Chapter 6, for the lockfile this chapter's container restores inside itself and the L0-to-L1 step the container builds on. No R fluency beyond reading a short script is assumed; no Docker or systems-administration background is assumed, since the chapter builds the idea from first principles.
By the end of this chapter you will be able to answer:

- Can you explain which determinants of a computational result a package lockfile leaves unpinned, and why a container is required to fix them?
- Can you distinguish a Docker image from a container, and state

the precise sense in which an image 'runs the same everywhere it runs'?
- Can you pin a base image by its immutable content digest, point its in-image installer at a dated repository snapshot, and say what each step secures?
- Can you name the environment variables that govern locale, time zone, and thread determinism, and predict a concrete failure that each one prevents?
- Can you select a Rocker profile appropriate to a given research archetype, and state honestly which rung of the reproducibility ladder a given image reaches and which it does not?

Outline.

- Where this sits on the stack: the layers a container pins.
- Orientation: a lockfile that restored and still computed a different number.
- The analyst's contribution: three judgments no tool automates.
- What the lockfile leaves standing: the operating system, the system libraries, the R runtime.
- Images and containers: the vocabulary, and the Rocker project.
- Making the image itself reproducible: pinning by digest and by dated snapshot.
- Apple Silicon and multi-architecture images: why a digest needs a fixed platform.
- Environment-variable determinism: locale, time zone, thread count.
- Profiles as base images, and a first look at the two-layer team model.
- Worked example: one project walked from L1 to L2.
- The cohort, continued: what the compendium's own Dockerfile would still need.

Check 1: from earlier chapters

You restore the same `renv.lock` on two laptops that run the same R release, and three things go wrong. In case (a), on one laptop

Draft, November 13, 2026. Content may change.

the restore stops because a package cannot find a system library to compile against. In case (b), a regression coefficient differs in its sixth decimal place. In case (c), a bootstrap interval differs although the seed and code are identical, and one laptop's project `.Rprofile` calls `RNGkind(sample.kind = 'Rounding')`. For each, name the determinant layer of Chapter 3 and say what closes it, drawing on Chapter 2 and Chapter 6.
The answer is on page 294.

7.1. Where this sits on the stack

This chapter captures, for the first time in the book, the two layers beneath the packages. These are the operating system with its system libraries, and the R runtime itself. Figure 7.1 marks both as captured on the shared determinant stack of Chapter 3. The chapter captures nothing above those two layers. A container that pins the operating system and the R version still restores whatever the lockfile says about packages, which is the layer Chapter 6 already captures.

```
plot_determinant_stack(
  captures = c('Operating system, system libraries', 'R
   ↪  runtime'),
  title = 'Containers capture the operating system and the
   ↪  R runtime')
```

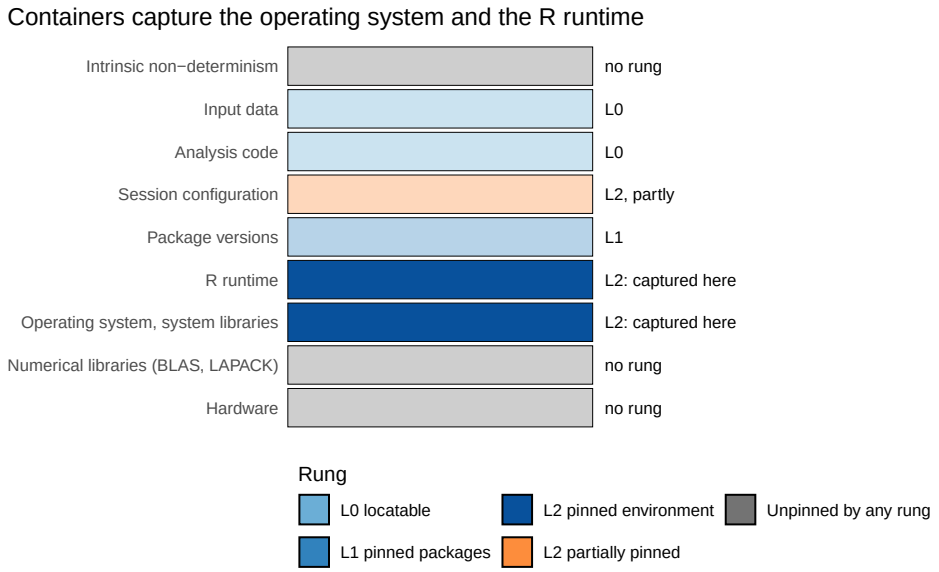


Figure 7.1.: The determinant stack of Chapter 3, with the operating system and system libraries, and the R runtime, marked as captured by this chapter. These are the two layers a lockfile cannot reach, because neither is an R package; pinning them is the L1-to-L2 step this chapter teaches.

7.2. Orientation

Consider an epidemiologist who has done everything the previous chapter asked. Her cohort analysis is under version control, and every R package it uses is fixed to an exact version in a lockfile. She has reached L1, pinned packages, and she is entitled to some confidence. She sends the project to a collaborator, who runs `renv::restore()` and watches it fail. One of her packages links against the geospatial library GDAL, which is present on her machine and absent on his, and the package will not install without it. On a third machine the restore succeeds, but the analysis returns a subtly different figure, because that machine runs an older release of R in which a numerical default differs. The lockfile did its job perfectly.

The lockfile pins the R packages and nothing beneath them. It says nothing

Draft, November 13, 2026. Content may change.

about the operating system, the system libraries such as GDAL or `libxml2` that many packages link against, or the R interpreter itself. As Ushey observed in documenting `renv`, the tool records the packages an analysis used, not the platform it ran on (Ushey & Wickham, 2026). A lockfile restored under a different R version, or on an operating system missing a required system library, may fail to install or, more insidiously, may install and behave differently. This is the gap the present chapter closes. Its subject is the container, the technology that fixes the operating system, the system libraries, and the runtime, and so carries an analysis from L1 to L2, pinned environment.

We shall build the container from first principles, assuming no prior contact with it. The reason is that the health scientist who most needs reproducibility is rarely the one with a systems-administration background. We begin with what a container is, and proceed to the Rocker project that supplies R-specific containers. We then take up the two refinements that make a container itself reproducible rather than merely convenient, and the complication that multi-architecture images bring to a recent Mac. The environment variables that govern determinism follow. We close with the profiles and the team model, then work one project from L1 to L2 and check this book's own compendium.

Check 2: what the lockfile missed

The epidemiologist's lockfile restored cleanly on a third machine but produced a subtly different number. Her collaborator's restore, on a second machine, failed outright. Name the determinant responsible for each failure, and say why the lockfile pinned neither.
The answer is on page 294.

7.3. The analyst's contribution

A container is a tool, and like every tool in this book it automates the mechanical part of the work and leaves the judgment to the analyst. Three decisions belong to the analyst and to no script.

1. **The choice of base environment must match the archetype, not the maximum.** A container can carry anything, and it is

Draft, November 13, 2026. Content may change.

tempting to reach for the largest base image that surely contains whatever one might need. That instinct is wrong. A manuscript archetype needs a base carrying a LaTeX toolchain; an R-package archetype needs only R and the check tools; a simulation archetype may want more cores than it wants software. Provisioning far above the need inflates the image, slows every rebuild, and obscures what the analysis actually depends on. The analyst chooses the smallest base that carries the real requirement.

2. **How far to pin decides whether the reproducibility claim is honest.** Consider a container built on a mutable tag such as `latest`, which installs whatever the package repository serves on the day of the build. Such a container looks reproducible and is not. Two builds from the same recipe on different days produce different images. Fixing this costs a little effort now, in pinning the base by digest and the installer by date, and the analyst must decide to spend it. The chapter will argue that the effort is the difference between claiming L2 and reaching it.

3. **The environment variables that govern determinism are the analyst's to set.** A container inherits a locale, a time zone, and a default thread count. These silently shape sorting, date arithmetic, and the last bits of a floating point sum. No tool infers what the analysis needs; the analyst declares it, in a few lines that a later reader can inspect and trust.

> The vocabulary of this chapter
>
> - **Image.** A frozen, read-only filesystem together with metadata, bundling an operating system, R, a set of packages, and configuration. It is analogous to a snapshot of a fully installed computer, packaged so it can be shipped.
> - **Container.** A running instance of an image. One image can start many containers; each begins from the same frozen filesystem.
> - **Layer.** Each instruction in an image's recipe produces one filesystem layer. An image is the stack of its layers, and identical layers are shared and cached across images.

Draft, November 13, 2026. Content may change.

- **Base image.** The starting image a recipe builds on, named on the `FROM` line. Here, a Rocker image providing a fixed R version on a fixed operating system.
- **Tag and digest.** A tag, such as `rocker/tidyverse:4.4.0`, is a human-readable pointer that may later be repointed. A digest, `@sha256:...`, is an immutable hash of the content it names, and the same digest always denotes the same content. For a multi-architecture image that content is an index listing one image per platform, so a digest fixes one filesystem only together with a fixed platform, here `--platform linux/amd64`.
- **Rocker.** The community project that publishes the standard R Docker images.
- **Dated snapshot.** A frozen view of a package repository on a particular day, so that installing 'the current version' resolves to the same version on every build.
- **Profile.** The base-image bundle a project is built from, such as `minimal`, `tidyverse`, `rstudio`, or `publishing`. It selects the `FROM` image.

Check 3: whose decision

A junior analyst asks why the base image is not simply 'the biggest one available', since a bigger image can only contain more of what might be needed. Using the first analyst judgment of this section, explain what is wrong with that reasoning.
The answer is on page 294.

7.4. What the lockfile leaves standing

The boundary of what a lockfile fixes is the whole motivation for the container. A lockfile records, for every R package the analysis code uses, the exact version and the repository it came from, and `renv::restore()` reinstalls that precise set (Ushey & Wickham, 2026). That is the L0-to-L1 transition, and it is genuine progress. Beneath the packages, however, sits a stack of determinants the lockfile never touches.

Draft, November 13, 2026. Content may change.

Three of these matter most. The first is the operating system, which supplies the C library and the conventions for locale and sorting against which every package was compiled. The second is the layer of system libraries, the compiled non-R software that packages link against: GDAL for spatial data, `libxml2` for XML parsing, the numerical libraries that implement linear algebra. A package that needs GDAL will not install on a machine that lacks it, and the lockfile cannot supply it, because GDAL is not an R package. The third is the R interpreter itself. A lockfile pins the packages but not the R version: its `R` block records the version, and nothing in it can install that version. As we saw in the introduction, a change between R releases can alter a result silently: the `sample()` default that changed in R 3.6.0, the `stringsAsFactors` default that flipped in R 4.0.0. Restore the same lockfile under a different R, and the packages match while their behavior does not.

`renv` is therefore necessary and, by itself, insufficient. It closes the package layer cleanly and leaves the three layers beneath it exposed. Fixing those takes a tool that packages the whole computer the packages ran on, which is the container.

> Check 4: three layers, one tool
>
> Name the three determinants beneath the package layer that a lockfile cannot pin, and say, for each, why it is not an R package and so cannot appear in a lockfile.
> *The answer is on page 294.*

7.5. Images and containers

The idea of the container is straightforward, and it is easiest to grasp through the distinction between an image and a container. The two are as different as a recipe from a meal, or a class from an object.

An **image** is a frozen filesystem plus metadata. It is, quite literally, a snapshot of a fully installed computer: it holds an operating system, the system libraries, a particular R, a particular set of R packages, and the configuration that ties them together. All of this is captured as a single

Draft, November 13, 2026. Content may change.

read-only artifact. The image does not run. It is inert, like a disk image or a virtual-machine template, waiting to be started.

A **container** is what one gets by starting an image. It is a running instance. The frozen filesystem is made live, and a process begins inside it. That process sees exactly the operating system and libraries the image froze, regardless of what the host machine outside the container happens to have. One image can start many containers, each an independent running copy that begins from the same frozen state. When a container exits, any change it made to its own filesystem is ordinarily discarded, and the next container starts again from the pristine image. The image is the durable, shippable thing; the container is the ephemeral, running thing.

Images are built in **layers**, which explains both the speed of the tool and the way we shall pin it. An image is built from a recipe, the Dockerfile, whose instructions execute in order, and each instruction adds one layer to a stack. A layer that installs the operating system sits at the bottom; a layer that installs R sits on top of it; a layer that installs packages sits on top of that. Because layers are content-addressed and cached, an unchanged lower layer is never rebuilt, and two images that share a base share its layers on disk. This is why building a second project from the same base is fast, and why, later, we shall be careful about which layer a given change invalidates.

The property that earns the container its place in this book is often summarized as 'it runs the same everywhere it runs'. The claim is precise, and the repetition of 'runs' is deliberate. Wherever a container runtime is present, the same image starts a container with the same operating system, the same system libraries, and the same R. A result computed inside it therefore does not depend on the host. The qualification is that the container must actually run: the guarantee holds across every machine with a compatible runtime, and it is silent about machines that have none. This is a real boundary, and we return to it, and to the runtimes that widen it, in Chapter 8. For present purposes the container gives us what the lockfile could not: a fixed operating system, fixed system libraries, and a fixed R, traveling together as one artifact.

Draft, November 13, 2026. Content may change.

Check 5: image or container

A colleague says, ‘I deleted a file inside the container by accident, so I restarted it and the file was back. Docker must have a recovery feature.’ What is actually happening, in the vocabulary of this section? *The answer is on page 295.*

7.6. The Rocker project

A Dockerfile could, in principle, begin from a bare operating system and install R by hand. Almost no one does, because the R community maintains a set of ready base images that have become the standard starting point. This is the Rocker project. Boettiger introduced Docker to the reproducible-research audience and argued that a container is the natural unit for shipping a computational environment (Boettiger, 2015). Boettiger and Eddelbuettel then built and documented the Rocker images themselves, a family of R-specific base images maintained in step with R releases (Boettiger & Eddelbuettel, 2017). The fuller account, including the tooling that has grown around the images, is given by Nüst and colleagues in their survey of what they call the Rockerverse (Nüst, Eddelbuettel, et al., 2020).

The practical value of Rocker is that it supplies, for each R version, a tested base image with R already installed on a known operating system. The analyst’s recipe can then build on that shared foundation. The images form a small hierarchy. At the bottom sits `rocker/r-ver`, R on Ubuntu and little else. Above it, `rocker/rstudio` adds the RStudio Server development environment. `rocker/tidyverse` builds on `rocker/rstudio`, adding the tidyverse packages and the `pandoc` document converter, so it carries RStudio Server as well and is the larger of the two. And `rocker/verse` adds a LaTeX toolchain for rendering manuscripts to PDF. Each higher image is built on the one below, so they share layers, and each fixes a larger environment than the last. We shall meet these again as the profiles.

Draft, November 13, 2026. Content may change.

Check 6: why not start bare

Starting a Dockerfile from a bare operating-system image and installing R by hand is possible. Explain why almost no one does, and what is lost in reproducibility terms if two analysts each hand-roll their own R installation rather than build on the same Rocker base.
The answer is on page 295.

7.7. Making the image itself reproducible

A container is reproducible only if the recipe that builds it is, and the naive recipe is not. Two refinements separate a container that merely runs today from one that will rebuild to the same image next year. Nüst and colleagues, in their ten simple rules for writing Dockerfiles for reproducible data science (Nüst, Sochat, et al., 2020), argue for pinning software versions explicitly. They also argue for naming version-specific image tags rather than `:latest`. At the same time they caution that even a version-specific tag is not a guarantee against change, since the maintainer can still rebuild and repoint it. (Inspected: the published rules recommend version tags, not cryptographic digests, and do not discuss dated package-repository snapshots.) This chapter takes that caution one step further than their rules do, with two refinements that do the most work here:

The first refinement pins the base image by its **digest** rather than by a **tag**. A tag such as `rocker/tidyverse:4.4.0` is a mutable pointer. It names an image today, but the maintainers may rebuild and repoint it tomorrow, for instance to patch a security flaw in the operating system. A recipe that named the tag would then build on different bytes, without any change to the recipe itself. A digest is an immutable hash of the exact image content, so that

```
FROM rocker/tidyverse@sha256:1c2f...e8a9
```

names not 'whatever `4.4.0` points to' but one specific piece of content that the digest will denote forever. For the Rocker images this chapter uses, that content is an image index, a short table with one image per CPU

Draft, November 13, 2026. Content may change.

architecture, from which a build receives the entry matching the platform it requests. A digest therefore fixes one filesystem only together with a fixed platform, so this book pins the index digest and always builds with `--platform linux/amd64`. The section on Apple Silicon below explains why that platform, and the alternative of pinning the platform-specific digest.

Figure 7.2 follows one unchanged recipe across a year to show what the difference costs. The recipe is not edited at any point in either row; the only thing that varies is whether its `FROM` line names the tag or the digest. (The digest strings shown here and in the figure, such as `sha256:1c2f...e8a9` and `sha256:6f05...`, are truncated for display and illustrative only. A digest copied for actual use must be the full 64-character hash, such as the one the showcase Dockerfile below carries.) Nothing in the upper row is an error or a misconfiguration on the maintainer's part. Repointing a tag to patch a security flaw is the correct thing for them to do, which is exactly why a tag cannot carry a reproducibility claim.

```
library(ggplot2)
library(tibble)

spans <- tibble(
  row = c('FROM rocker/r-ver:4.4.0',
          'FROM rocker/r-ver:4.4.0',
          'FROM rocker/r-ver:4.4.0',
          'FROM rocker/r-ver@sha256:6f05...'),
  xmin = c(0, 3.5, 8.5, 0),
  xmax = c(3.5, 8.5, 12, 12),
  build = c('build 1', 'build 2', 'build 3', 'build 1')
)
spans$row <- factor(spans$row,
                    levels = c('FROM
                     ↪  rocker/r-ver@sha256:6f05...',
                               'FROM rocker/r-ver:4.4.0'))

ggplot(spans) +
  geom_rect(aes(xmin = xmin, xmax = xmax,
                ymin = as.numeric(row) - 0.26,
                ymax = as.numeric(row) + 0.26,
```


Draft, November 13, 2026. Content may change.

```
              fill = build),
          color = 'white', linewidth = 1.4) +
scale_fill_manual(values = c('build 1' = '#14746f',
                             'build 2' = '#7fb1ad',
                             'build 3' = '#c9e2df'),
                  guide = 'none') +
scale_y_continuous(breaks = 1:2,
                   labels = levels(spans$row),
                   limits = c(0.45, 2.95)) +
scale_x_continuous(breaks = c(0, 3.5, 8.5, 12),
                   labels = c('Jan', 'Apr', 'Sep',
                    ↪  'Dec')) +
annotate('text', x = 3.5, y = 2.42, label =
 ↪  'maintainer\nrebuild',
         size = 2.6, lineheight = 0.95, color = 'grey25')
          ↪  +
annotate('text', x = 8.5, y = 2.42, label =
 ↪  'maintainer\nrebuild',
         size = 2.6, lineheight = 0.95, color = 'grey25')
          ↪  +
annotate('text', x = 12.15, y = 2, hjust = 0,
         label = 'three environments\nfrom one recipe',
         size = 2.7, lineheight = 0.95, color = '#a33') +
annotate('text', x = 12.15, y = 1, hjust = 0,
         label = 'one environment,\nalways',
         size = 2.7, lineheight = 0.95, color = 'grey25')
          ↪  +
coord_cartesian(xlim = c(0, 16.6), clip = 'off') +
labs(x = NULL, y = NULL) +
theme_minimal(base_size = 9.5) +
theme(
  panel.grid.major.y = element_blank(),
  panel.grid.minor = element_blank(),
  axis.text.y = element_text(family = 'mono', size = 8,
   ↪  hjust = 1)
)
```


Draft, November 13, 2026. Content may change.

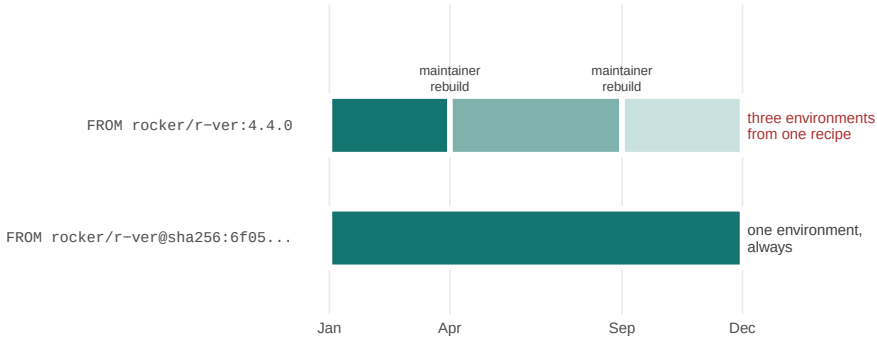


Figure 7.2.: One unchanged recipe built repeatedly over a year, under a tag and under a digest. The upper row builds on whatever the tag currently points at, so a maintainer rebuild in April and another in September silently change the ground the analysis stands on, giving three different environments from identical text. The lower row names a digest, builds for one fixed platform, and receives the same bytes at every build. The maintainer rebuilds are ordinary maintenance, not mistakes: the point is that a mutable pointer cannot carry an immutable claim.

The second refinement concerns what happens inside the image when it installs packages. An installer pointed at a live package repository resolves 'the current version' to whatever is current on the day of the build. An image built in March and an image built in June from the same recipe therefore carry different packages. The remedy is to point the in-image installer at a **dated snapshot** of the repository, a frozen view of the repository as it stood on a chosen day. With the snapshot fixed, a request for 'the current version' resolves to the same version on every build. In sum, the digest fixes the ground the image is built on. The lockfile, restored inside the image, fixes which package versions are installed onto it. The dated snapshot fixes where those packages are fetched from, and what any install that names no version resolves to.

A small Dockerfile with both refinements in place, and with the environment variables of the next section, follows. It is shown as static text, but we tested it. On 2026-10-11 we built it with `docker build --platform linux/amd64` against a lockfile written by `renv` 1.2.3 that pins one package, `jsonlite` 2.0.0. The lockfile's recorded R version was set to 4.4.0 to match

Draft, November 13, 2026. Content may change.

the image, and the built image reported `jsonlite` 2.0.0 and `renv` 1.2.3. The digest is the one the `rocker/r-ver:4.4.0` tag resolved to on that date. A reader adapting the recipe should look the digest up afresh, as Exercise 1 shows, rather than copy one from a book.

```
# rocker/r-ver:4.4.0 is built on Ubuntu 22.04 (jammy), so
 ↪  the
# binary snapshot below must name jammy. Digest as of
 ↪  2026-10-11.
FROM rocker/r-ver:4.4.0@sha256:73adabf378930f8744f56bac1fe
 ↪  5e99870532eb4ced1a15075991bac909d21f3

ARG PPM_SNAPSHOT=https://packagemanager.posit.co/cran/__li
 ↪  nux__/jammy/2026-06-27
ARG RENV_VERSION=1.2.3

LABEL org.opencontainers.image.base.name="docker.io/rocker
 ↪  /r-ver:4.4.0" \
      org.opencontainers.image.base.digest="sha256:73adabf
 ↪  378930f8744f56bac1fe5e99870532eb4ced1a15075991bac909d2
 ↪  1f3" \
      cran.snapshot="2026-06-27"

ENV LANG=en_US.UTF-8 \
    LC_ALL=en_US.UTF-8 \
    TZ=UTC \
    OMP_NUM_THREADS=1 \
    OPENBLAS_NUM_THREADS=1 \
    RENV_CONFIG_REPOS_OVERRIDE=${PPM_SNAPSHOT}

# Later lines win, so this replaces Rocker's own snapshot
 ↪  date.
RUN echo "options(repos = c(CRAN = '${PPM_SNAPSHOT}'))" \
      >> /usr/local/lib/R/etc/Rprofile.site

WORKDIR /project
COPY renv.lock renv.lock
```


Draft, November 13, 2026. Content may change.

```
RUN Rscript -e "install.packages('renv')" \
 && Rscript -e "stopifnot(packageVersion('renv') ==
 ↪  '${RENV_VERSION}')"

RUN Rscript -e "renv::restore(library = .libPaths()[1],
 ↪  prompt = FALSE)"
```

Several lines guard against failures that are easy to miss.

- **The base and the binary URL must agree.** Rocker builds `rocker/r-ver:4.4.0` `FROM ubuntu:jammy`, that is, Ubuntu 22.04, as its published Dockerfile for the release shows. Posit Package Manager serves Linux binaries compiled for a named distribution release, so the snapshot URL must say `jammy`. A URL naming a newer release, such as `noble` (24.04), would install binaries built against a different set of system libraries.
- **`WORKDIR` gives the project a directory of its own.** Without it the build runs in `/`, the root of the file system, which `renv` would then treat as the project. That directory contains the home directory, and `renv::init()` refuses to initialize a project in that position. A dedicated directory avoids the problem and keeps `renv`'s files out of the root.
- **`renv` is installed at the version the lockfile records.** The lockfile names the `renv` that wrote it, here 1.2.3. On the snapshot date that was the current release, so a plain `install.packages('renv')` fetches it, and the `stopifnot()` line turns any mismatch into a failed build. Passing `--build-arg RENV_VERSION=1.2.2` makes the build stop at that line with `packageVersion("renv") == "1.2.2" is not TRUE`, which is how we confirmed the check can fail.
- **The snapshot governs the restore only because `RENV_CONFIG_REPOS_OVERRIDE` says so.** By default `renv::restore()` ignores the session's repository option and downloads from the repositories recorded in the lockfile. We confirmed both directions by building variants. With the override removed and a different date in `Rprofile.site`, the restore still fetched from the lockfile's date. With the override set and a different date written into the lockfile, the restore fetched from the image's snapshot, as `jammy` binaries.

Draft, November 13, 2026. Content may change.

- **`library = .libPaths()[1]`** installs the restored packages into the image's site library, so every R session in the container sees them without activating a project.

The build prints a warning from `renv` that `curl` does not appear to be installed and that downloads will fail. In this image the warning is harmless: the downloads go through R's built-in `libcurl` support, as the completed restore shows.

The snapshot does not choose the versions of the packages in the lockfile; the lockfile does, and a recorded version older than the one current on the snapshot date is sought in the repository's archive. What the snapshot fixes is the source of every download, the binary build that matches the image's Ubuntu release, and the version of anything installed without a lockfile entry. Here that means `renv` itself, and any later `install.packages()` call made inside the running container. Rocker's own image already points R at a dated snapshot, fixed for each R version at the last day that version was current (`2024-06-13` for 4.4.0), which is why the `echo` line appends rather than replaces: the later setting wins.

The container and the lockfile divide the labor. The `FROM` line fixes the operating system, the system libraries, and the R; the snapshot fixes where packages come from; the `COPY renv.lock` and `renv::restore()` fix the exact package versions inside that environment. Neither the container nor the lockfile alone is enough. A container without a lockfile fixes the environment but not which package versions live in it; a lockfile without a container fixes the packages but not the platform. An L2 analysis uses both. The honest reading of an image, then, is that it reaches L2 only when the base is pinned by digest, the snapshot is dated, and a lockfile is restored inside it.

Verification

See the pinning actually hold by building the image twice and confirming the two builds carry the same content: the same operating system and the same R and package versions. This can be checked, for example, by comparing `renv::status()` and the installed apt and R package versions across the two builds. Do not test this by comparing the two image digests. Docker digests are not reproducible across

Draft, November 13, 2026. Content may change.

rebuilds by default, because per-layer timestamps and file modification times differ, so two builds of a correctly pinned recipe carry different digests. Bit-identical digests require reproducible-build tooling, such as `SOURCE_DATE_EPOCH` and the BuildKit rewrite-timestamp option, which is a separate concern from whether the content is pinned. Make the check able to fail: replace the digest with a mutable tag and the snapshot with a live repository, rebuild on two different days, and confirm `renv::status()` or the package versions now disagree. If the content differs between two builds of a recipe believed to be pinned, the recipe is not yet pinned, and the honest level is still L1 whatever the `FROM` line claims.

! The honest level

When this chapter's own practice names a digest, a dated snapshot, and a restored lockfile together, as it does in the Dockerfile above, it reaches L2 for the two layers a lockfile cannot reach. These are the operating system and system libraries, and the R runtime. It leaves L3, verification against a recorded baseline, entirely to Chapter 12.

A pinned recipe must also run where the reviewer works. In the R Consortium's Pilot 4, a Shiny application submitted to the FDA with a Dockerfile, the agency's statistical reviewers installed WSL and Docker Desktop on Windows. They noted that agency-issued laptops may not permit WSL or an unlicensed Docker Desktop, and found that the R version still decided whether the packages restored (U.S. Food and Drug Administration, Center for Drug Evaluation and Research, Office of Biostatistics, 2026). Their review, dated May 2026, concluded that containers so far may not give the robustness a regulatory submission needs (R Consortium, 2026; U.S. Food and Drug Administration, Center for Drug Evaluation and Research, Office of Biostatistics, 2026); the setup appendix covers such locked-down machines.

Draft, November 13, 2026. Content may change.

Check 7: tag versus digest versus snapshot

A recipe pins the base image by digest but still installs packages with a plain `install.packages()` against the live repository. Is this recipe's claim to L2 honest? Say exactly what is and is not pinned.
The answer is on page 295.

7.8. Apple Silicon and multi-architecture images

The digest pinned in the previous section carries one complication, which matters to anyone who builds on a recent Mac. A single image tag, such as `rocker/r-ver:4.4.0`, need not name one fixed set of bytes. Docker images can be published as a **manifest list**, also called an image index. That is, one tag, or even one digest, points not directly at a filesystem but at a small table of entries. Each entry names a different CPU architecture built from the same recipe. When a client pulls the tag, the runtime inspects its own architecture and silently selects the matching entry. An `amd64` host receives the `amd64` build, an `arm64` host receives the `arm64` build, and the person who wrote `FROM rocker/r-ver:4.4.0` never chose between them. This is what a multi-architecture image is: one name that resolves to different, independently built images depending on who is asking.

Not every Rocker image offers that choice. Rather than trust a secondary source, we checked directly, with `docker buildx imagetools inspect`. The digest on the showcase Dockerfile's `FROM` line names an index with a `linux/amd64` and a `linux/arm64` image (verified). But `rocker/tidyverse:4.4.0` and `rocker/verse:4.4.0`, the bases of the tidyverse and publishing profiles, publish `linux/amd64` only (verified); neither carries an `arm64` entry. An Apple Silicon Mac is an `arm64` host. A plain `docker build` of the showcase Dockerfile on such a Mac therefore succeeds without any flag, natively, on the `arm64` image (verified), while a recipe built on `rocker/tidyverse` or `rocker/verse` has no matching entry to select and needs `--platform linux/amd64` to build at all.

The native build is the one to be wary of, because it succeeds quickly and yields a different image from the one a labmate on an Intel laptop, a CI runner, or a cluster node builds from the same `FROM` line. It starts from a

Draft, November 13, 2026. Content may change.

different base image, and in our `arm64` build of the showcase the packages compiled from source where an `amd64` build installs binaries. An image built natively on a Mac and pushed to a registry will also not run on an `x86` cluster. The flag is what makes a Mac build the same image the cluster runs, so this book passes `--platform linux/amd64` to every build, whatever the base.

The `--platform linux/amd64` flag requests the `amd64` entry explicitly, overriding the host-architecture default. On an Apple Silicon host, Docker Desktop satisfies that request through QEMU, a CPU emulator that translates each `amd64` instruction into the sequence of native `arm64` instructions that has the same effect. Docker's own documentation on multi-platform builds is direct about the cost: emulation with QEMU 'can be much slower than native builds', especially for compute-heavy work (Docker Inc., 2024b). Every package compile and every `renv::restore()` inside the image pays this tax; a build that takes two minutes natively can take much longer under emulation. This is a real cost, worth planning a build around, but it is a cost in time, not in correctness. A QEMU-translated `amd64` instruction still computes the `amd64` result, so a container built this way runs the same program, only more slowly.

Emulation is not where a numerical risk would come from; the native build is. A native `arm64` build of the showcase Dockerfile is not guaranteed to match an emulated `amd64` build bit for bit. The reason is that a different architecture's base image can, in principle, link a different BLAS, enable different vector instructions, or carry a different compiler default. Any of these can shift a floating-point result in its last few bits. None of that is new: it is the same category of hidden determinant, beneath the package layer, that the next section takes up for thread counts. The digest alone does not settle the question, because one index digest names both images. A digest together with a fixed platform does. The platform decides which entry of the index a build receives, and the digest decides that the entry is the same every time it asks. Pin both, or pin the digest of the platform-specific manifest, which names one filesystem by itself. This book pins the index digest and always passes `--platform linux/amd64`, the architecture of GitHub's standard CI runners and of most HPC clusters.

Draft, November 13, 2026. Content may change.

Check 8: emulation and correctness

A student on an Apple Silicon laptop builds this chapter's showcase Dockerfile with `--platform linux/amd64` and gets a working container. However, the build takes nine minutes instead of the ninety seconds a labmate on an Intel laptop reports. The student worries the slow build means the result cannot be trusted. Is that worry justified, and what, already taught in this chapter, actually decides whether the result is trustworthy?
The answer is on page 296.

7.9. Environment-variable determinism

Are a fixed operating system and a fixed R enough? Not quite, because the same R on the same operating system can behave differently according to a handful of environment variables that the container inherits. Three of these deserve concrete illustration, because the failures they cause are silent and the fixes are one line each.

The first is the **locale**, carried in `LANG` and `LC_ALL`, which governs, among other things, how text is sorted and how dates and numbers are formatted for display. R's own numeric parsing is not one of the things it governs. `LC_NUMERIC` is pinned to `"C"` inside R regardless of what `LANG` or `LC_ALL` say, so `as.numeric('1,5')` returns `NA` with a coercion warning under any locale the host or container sets. (We verified this by calling `Sys.setlocale('LC_ALL', 'fr_FR.UTF-8')` and confirming that `Sys.getlocale('LC_NUMERIC')` still reports `'C'` and `as.numeric('1,5')` is still `NA`.) The risk is not that R silently reparses a number under a different convention; it is that data arriving from *outside* R already carries a convention baked in. A locale that uses a comma as the decimal mark, common across much of Europe, writes the number one and a half as `1,5` in a spreadsheet or CSV export. An ingest step that assumes a period decimal mark will misread it without erroring; an example is a hand-written regex that strips what it takes to be thousands separators. To make the failure concrete, consider the same numeric vector written under each convention by an external tool such as a spreadsheet

Draft, November 13, 2026. Content may change.

application. The table below shows what a naive period-assuming cleaning step reads back from the comma-convention strings.

```
library(tibble)

x <- c(1.5, 2200.75, 0.333)

period <- formatC(x, format = 'f', digits = 3, big.mark =
 ↪  ',')

comma <- gsub(',', '#', period)
comma <- gsub('\\.', ',', comma)
comma <- gsub('#', '.', comma)

read_back <- as.numeric(gsub(',', '', comma))

decimal_demo <- tibble(
  original      = x,
  period_locale = period,
  comma_locale  = comma,
  parsed_value  = read_back
)

knitr::kable(decimal_demo)
```

Table 7.1.: The same numbers written under a period decimal mark and a comma decimal mark, as a spreadsheet export might write them, and what a naive period-assuming cleaning step reads back from each. The comma-convention values are silently misread, with no error raised.

original	period_locale	comma_locale	parsed_value
1.500	1.500	1,500	1500.00000
2200.750	2,200.750	2.200,750	2.20075
0.333	0.333	0,333	333.00000

The parsed values are wildly wrong, and nothing warns the analyst. The

Draft, November 13, 2026. Content may change.

number `1.5` returns as `1500`, the number `2200.75` as roughly `2.2`, and the number `0.333` as `333`, because the period-assuming cleaning step treats the European comma as noise and strips it. This is a failure of the ingest code, not of R's parser. Even so, pinning `LANG` and `LC_ALL` in the image still matters for the parts of R that genuinely do depend on locale: collation and formatted output. Sorting the strings `'cote'`, `'côte'`, `'coté'`, `'coter'` with `sort()` returns a different order under `LC_ALL=C` than under `LC_ALL=fr_FR.UTF-8` (verified). Likewise, `format(Sys.Date(), '%B')` prints an English month name under one and a French month name under the other (verified). A report that sorts a table of names, or prints a month in running text, will read differently on two machines that disagree about locale even though every number in it was parsed identically. Fixing `LANG` and `LC_ALL` in the image removes that variability, which is why the Dockerfile above sets both to `en_US.UTF-8`.

The second is the **time zone**, carried in `TZ`. Date and time arithmetic depends on it, and an analysis that buckets events by calendar day will assign a late-evening event to different days in different time zones. Fixing `TZ=UTC` in the image makes the calculation independent of where the container runs.

The third is the **thread count**, carried in variables such as `OMP_NUM_THREADS`. Numerical libraries that implement linear algebra may split a computation across several threads. And because floating-point addition is not exactly associative, the order in which partial sums are combined can change the last bits of a result. Two runs of the same bootstrap or the same mixed model, one on four threads and one on eight, may therefore differ in the final digits. Which variable actually governs the summation order depends on the BLAS implementation the image carries: `OPENBLAS_NUM_THREADS` for OpenBLAS (which falls back to `OMP_NUM_THREADS` when it is unset), `MKL_NUM_THREADS` for the Intel MKL, and `OMP_NUM_THREADS` for the OpenMP-parallel sections. The `rocker/r-ver` images link R against the multithreaded OpenBLAS, not the single-threaded reference BLAS; in the image built above, `sessionInfo()` reports `openblas-pthread/libblas.so.3`. That is why the Dockerfile sets `OPENBLAS_NUM_THREADS=1` alongside `OMP_NUM_THREADS=1`. OpenBLAS reads its own variable first, so a stray `OPENBLAS_NUM_THREADS` would override `OMP_NUM_THREADS=1`. Setting both to one together removes thread count as a source of variation in the

Draft, November 13, 2026. Content may change.

order of summation. The image determines which BLAS is installed, but no rung of the ladder guarantees this layer. An L2 image run on a different CPU, or with these variables overridden at run time, can still move the last digits. The cost is a little speed, traded for run-to-run stability on the same image. Whether that trade is worth making is the analyst's judgment, and a container lets that judgment be recorded and enforced instead of left to whatever the host defaults to.

Check 9: which variable

A collaborator in Berlin and a collaborator in San Diego run the same containerized script on the same image and obtain identical package versions. Yet their daily case-count tables disagree by one case on several days, always at the boundaries between days. Which environment variable is the likely culprit, and why does pinning it in the image resolve the disagreement?
The answer is on page 296.

7.10. Profiles as base images

The choice of base image is, in practice, a choice from a short menu, and the framework this book uses names the entries **profiles**. A profile is a base-image bundle, and selecting one sets the `FROM` line and little else. Four profiles cover most needs.

The **minimal** profile builds on `rocker/r-ver`: R on Ubuntu with the development essentials and nothing more. It is the smallest base, and the right one when the analysis pulls in whatever it needs through the lockfile. The **tidyverse** profile builds on `rocker/tidyverse`, adding the tidyverse packages and `pandoc`, and is the natural default for data analysis. The **rstudio** profile builds on `rocker/rstudio`, adding the RStudio Server environment for those who prefer to work in the browser-based IDE. The **publishing** profile builds on `rocker/verse`, which carries a LaTeX toolchain, and is the base to choose when the deliverable is a rendered PDF manuscript.

The design intent is that the profiles differ in as few lines as possible. Moving a project from one profile to another changes the `FROM` line and, at most, a

Draft, November 13, 2026. Content may change.

single system-installation line. The reproducibility-relevant configuration stays constant: the locale and time-zone variables, the dated snapshot, and the lockfile restore. The profile decides what the base carries, and the lockfile decides what is installed on top.

The archetypes of Chapter 5 map onto the profiles naturally, and the mapping is the concrete form of the first analyst judgment above. The same machinery, a digest-pinned base, a dated snapshot, a restored lockfile, and the determinism variables, serves every archetype; only the base changes.

- A **data-analysis** archetype is well served by the tidyverse or minimal base, according to how much of its stack it draws from the base rather than the lockfile.
- A **manuscript** archetype needs the publishing base, because rendering to PDF requires the LaTeX toolchain that `rocker/verse` carries and the lighter bases do not.
- An **R-package** archetype needs very little: R and the check tools suffice, so the minimal base is the honest choice, and provisioning a LaTeX toolchain it never uses would be waste.
- A **simulation** archetype often wants more cores than software, so the base may be minimal while the container is run with a larger core allocation. Here the analyst must weigh the determinism point above. More cores may speed the simulation while changing the last bits of a result, so the thread variables (`OMP_NUM_THREADS` and `OPENBLAS_NUM_THREADS`) and the seeding discipline of Chapter 5 deserve particular care.
- A **blog** archetype renders many posts rather than one report, and is well served by the publishing or tidyverse base according to whether any post needs PDF output.

Check 10: choosing a profile

An R-package archetype is built against the publishing profile because the author 'might as well have LaTeX available'. Name the concrete risk this choice creates, distinct from image size.
The answer is on page 296.

Draft, November 13, 2026. Content may change.

7.11. A first look at the two-layer team model

A container fixes an environment for one analyst. A team needs one shared environment, and this introduces a wrinkle we shall only sketch here and develop fully in Chapter 13. Building an image from a Rocker base and a full package set takes many minutes. It would be wasteful for every member of a lab, and every one of a lab's projects, to rebuild the same base from scratch.

The remedy is a two-layer model. A team lead builds a comprehensive **team image** once, carrying the lab's preferred packages, and pushes it to a shared registry. Each project then builds a thin second layer on top of that team image, rather than on the raw Rocker base. The first layer, the team image, is built once and reused everywhere; the second layer, the per-project image, adds only what the project needs and builds in seconds, where the full image takes minutes. Because the team image is itself pinned by digest, recorded in a committed file, and built and pulled for one platform (`--platform linux/amd64`), every member runs the same filesystem. 'Works on my machine' therefore cannot arise: there is, in the relevant sense, only one machine. A member who drops the flag on an Apple Silicon Mac falls outside that guarantee. The mechanics of building, pushing, digest-recording, and updating the team image are the business of Chapter 13.

> Check 11: why not rebuild every time
>
> Explain, to a lab member who asks why the team does not simply have every project build from the raw Rocker base, what the two-layer model buys beyond saved build minutes.
> *The answer is on page 297.*

7.12. Worked example

We now follow a concrete case through, in the manner of the framework this book uses, which generates the Dockerfile so that the analyst does not write one by hand. A biostatistician is analyzing a registry cohort. The deliverable is a set of tables and figures, not a typeset manuscript, so

Draft, November 13, 2026. Content may change.

the appropriate archetype is data analysis and the appropriate profile is minimal. That profile carries R and the essentials, with the analysis stack drawn from the lockfile. She scaffolds the project and builds the image. The shell session, shown as static text and never executed here, is short.

```
$ mkdir cohort-study && cd cohort-study
$ zzc minimal
$ make docker-build
$ make r
```

The `zzc minimal` step generates the project, including a Dockerfile pinned to a digest and a dated snapshot; `make docker-build` builds the image; and `make r` starts a container and drops her into an R session inside it. The generated Dockerfile is, in essence, the one shown earlier. It carries a digest-pinned `FROM rocker/r-ver`, the locale and time-zone and thread variables, and the dated snapshot wired into the repository options. It ends with a `renv::restore()` that installs the locked packages inside the fixed environment. The image records its own provenance in labels, the base image, its digest, and the snapshot date, so a later reader can see exactly what was pinned without leaving the image.

Now suppose a reviewer asks for a typeset manuscript with the analysis embedded. The archetype has shifted, and so should the profile. She regenerates against the publishing base:

```
$ zzc publishing
$ make docker-build
```

The `FROM` line now names `rocker/verse`, the LaTeX toolchain comes with it, and the manuscript renders to PDF inside the container. Almost nothing else in the recipe changed: the locale and thread variables, the dated snapshot, and the lockfile restore are identical. The same reproducibility machinery served both archetypes, and the move between them was a single changed line.

What has she gained, in the vocabulary of the ladder? Before the container, at L1, her collaborator's restore could fail for a missing system library, or succeed and compute a different number under a different R. After it, at L2,

Draft, November 13, 2026. Content may change.

the operating system, the system libraries, and the R are fixed and travel with the analysis, and the locale and time-zone and thread variables are declared rather than inherited. What she has not yet gained is verification, L3. An image that builds and a report that renders are not the same as a report whose numbers have been regenerated and shown to match a recorded baseline. That is the subject of Chapter 12.

Check 13: complete the Dockerfile

The worked example's generated Dockerfile is, in essence, the tested recipe of the section on making the image itself reproducible. It is reprinted below with four of its lines removed, and the two label values that would give lines 1 and 2 away masked. Write each missing line so that the recipe again reaches L2 when built with `--platform linux/amd64`.

Draft, November 13, 2026. Content may change.

```
# rocker/r-ver:4.4.0 is built on Ubuntu 22.04 (jammy),
 ↪  so the
# binary snapshot below must name jammy. Digest as of
 ↪  2026-10-11.
# (1) ____: name the base image so that it cannot move

# (2) ____: the repository the installer fetches from
ARG RENV_VERSION=1.2.3

LABEL org.opencontainers.image.base.name="docker.io/ro
 ↪  cker/r-ver:4.4.0" \
      org.opencontainers.image.base.digest="<the digest
 ↪  of line 1>" \
      cran.snapshot="<the date of line 2>"

ENV LANG=en_US.UTF-8 \
    LC_ALL=en_US.UTF-8 \
    TZ=UTC \
    # (3) ____: fix the thread count of OpenMP and of
 ↪  the BLAS
    RENV_CONFIG_REPOS_OVERRIDE=${PPM_SNAPSHOT}

# Later lines win, so this replaces Rocker's own
 ↪  snapshot date.
RUN echo "options(repos = c(CRAN = '${PPM_SNAPSHOT}'))"
 ↪  \
      >> /usr/local/lib/R/etc/Rprofile.site

WORKDIR /project
COPY renv.lock renv.lock

RUN Rscript -e "install.packages('renv')" \
 && Rscript -e "stopifnot(packageVersion('renv') ==
 ↪  '${RENV_VERSION}')"

# (4) ____: install the locked package versions
```

The answer is on page 297.

7.13. The cohort, continued

The running cohort compendium of this book is, itself, a small research compendium: a `cohort_raw.csv`, a `clean_cohort()`, and a model fit on top. Whether it has reached L2 yet can be checked directly.

```
files_needed <- c('Dockerfile', 'renv.lock')
present <- file.exists(file.path('../..', files_needed))
names(present) <- files_needed
present
#> Dockerfile  renv.lock
#>      FALSE       TRUE
```

Only `renv.lock` exists at the compendium's root, so by this chapter's standard the compendium is at L1. Reaching L2 needs nothing new. It needs only a Dockerfile that follows the showcase recipe built and verified earlier in this chapter, adapted to what the compendium's own `renv.lock` records:

- a `FROM` line naming a Rocker base appropriate to the compendium's archetype, data analysis, and carrying the R the lockfile records, `4.6.1` here: so `rocker/r-ver:4.6.1`, pinned by its index digest and built with `--platform linux/amd64`, rather than the showcase's `rocker/r-ver:4.4.0`. The two versions must match, because a lockfile restored under a different R is the mismatch this chapter opened with;
- a dated-snapshot `ARG`, naming the Ubuntu release the chosen base actually runs, wired into `RENV_CONFIG_REPOS_OVERRIDE` as the showcase Dockerfile does. The release changes with the R version: `rocker/r-ver:4.4.0` runs Ubuntu 22.04 (`jammy`), while `rocker/r-ver:4.6.1` runs Ubuntu 24.04 (`noble`), as the `CRAN` setting in its image configuration shows;

Draft, November 13, 2026. Content may change.

- the three determinism variables this chapter argued for, `LANG`/`LC_ALL` set to a fixed locale, `TZ=UTC`, and `OMP_NUM_THREADS`/`OPENBLAS_NUM_THREADS` set to a deliberate value, since the compendium's `bootstrap_ci()` refits the model many times, and a multithreaded BLAS could move the last digits of the interval it reports;
- a `WORKDIR`, a `COPY renv.lock renv.lock`, and the same two-step `renv` install and `renv::restore()` pattern, so the packages the compendium's scripts load are restored inside the pinned environment.

None of this is built here, because building and verifying an image is slow and the showcase Dockerfile earlier in this chapter already did that verification once. What the check above adds is specific to the compendium: no Dockerfile is in place yet, and the gap is these four concrete items, not a vague 'needs a Dockerfile'.

7.14. Collaborating with an LLM

A language model is a capable assistant for container work, and a Dockerfile it writes can look complete while lacking the pins this chapter requires. The analyst who delegates must verify, and the following three exchanges show what to watch for.

! Restricted data

One rule comes first (habit 3). Never paste or upload protected health information, identifiable records, or data held under a data use agreement, nor outputs derived from them, into a model or a coding agent unless three conditions all hold. The institution has approved that endpoint for that class of data. The IRB protocol and the consent permit the processing. The data use agreement, or the provider's terms, permits it. If any of the three is unclear, the answer is no.
The third condition is the one most often missed. Institutional approval of a tool does not override the terms under which a data provider released the data. NIH, for example, treats sharing controlled-access data through prompts with public generative AI tools, which it glosses as third-party tools, as a breach of the data use certification. It treats a model trained on such data as a data derivative, to be shared

Draft, November 13, 2026. Content may change.

only with other approved users and not kept after the project closes (National Institutes of Health, 2025a).
Describe the structure instead: the column names and types, or a small synthetic example generated from the data dictionary, never edited from real rows. A coding agent sees whatever files are in its working tree, so run it in a checkout or container where the real data is not mounted.
An agent also acts, so confine it to a branch and keep merges, pushes, and deposits for yourself (Section 2.10.1).

In container work, what an agent can see includes every host directory bind-mounted into a container it runs in, so a mount of the data directory exposes the data as surely as a copy in the working tree would.

Prompt. 'Write me a Dockerfile for my R analysis.'

Watch for. Check whether the answer opens with `FROM rocker/tidyverse:latest` or a bare version tag, installs packages from the live repository, or sets none of the determinism variables. Every one of these is a reproducibility defect: the mutable tag, the undated installer, the absent locale and time zone. Such a recipe runs today and is not L2.

Verification. Confirm that the `FROM` line carries an `@sha256:` digest, not merely a tag. Confirm also that the in-image installer points at a dated snapshot, with a distribution codename that matches the base image's Ubuntu release, and that `renv::restore()` is told to use it. Finally, confirm that `LANG`, `LC_ALL`, and `TZ` are set. Then build the image twice and confirm the two builds carry the same content, comparing the installed operating system and the R and package versions. This can be done, for instance, through `renv::status()` and the apt and R package version lists. Do not compare image digests, which differ across rebuilds even for a correctly pinned recipe because per-layer timestamps vary. If the content differs, the recipe is not pinned, whatever the model asserted. If an agent ran the two builds, read the two version lists yourself; its report that they match is not the comparison (Section 2.10.1).

A better prompt. 'Write a Dockerfile for an R `<version>` analysis restored from the attached `renv.lock`. Pin the base image by digest; point the installer at a dated Package Manager snapshot for `<codename>`; set `LANG`,

Draft, November 13, 2026. Content may change.

LC_ALL, and TZ; install nothing at run time. List the commands I should run to confirm that two builds install identical versions.' The slots are filled from the lockfile and the base image's Ubuntu release, as in 'The cohort, continued' above.

Prompt. 'My container gives different numbers than my laptop. Why?'

Watch for. Check whether the answer reaches first for the code, suggesting a bug in the analysis, and overlooks the environment. The environment is the more probable cause, because the container was supposed to fix it and may have fixed it incompletely. Typical gaps are a missing locale variable, a different thread count, or an unpinned package version inside a pinned environment.

Verification. Compare the environment inside the container with the one outside it: inspect LANG, LC_ALL, TZ, and the thread-count variables in both, and compare the installed package versions against the lockfile. Do not accept a diagnosis of the code until the environment has been ruled out, because a difference the container failed to pin is the more likely explanation.

Prompt. 'Which Rocker base should I use for my project?'

Watch for. Check whether the answer recommends a large, capable base, such as the verse or a data-science image, regardless of need. Over-provisioning inflates the image and obscures the real dependencies; under-provisioning, recommending a minimal base for a project that must render a PDF, produces a build that fails only when the manuscript is rendered.

Verification. Match the base to the archetype, not to the maximum. If the deliverable is a PDF manuscript, confirm the base carries a LaTeX toolchain, for instance by checking that the LaTeX tools are present in the built image. If it is an R package, confirm that R and the check tools suffice and resist any heavier base. The judgment is the first of this chapter's analyst contributions, and it is not one to delegate.

7.15. Exercises

1. Take any Rocker image on your machine or on Docker Hub and find both its tag and its content digest. Explain, in two or three sentences,

Draft, November 13, 2026. Content may change.

what could change about the image named by the tag while the image named by the digest could not, and why a reproducible recipe pins the latter.

2. You are handed a Dockerfile that begins `FROM rocker/tidyverse:latest` and installs packages with a plain `install.packages()` from the live repository. Identify every reason this recipe fails to reach L2, and rewrite the two instructions that most need fixing.

3. For each of the five archetypes, data analysis, manuscript, R package, simulation, and blog, name the profile you would choose and justify the choice in one sentence. For the two archetypes where the choice is genuinely arguable, state the trade-off on both sides.

4. Extend the executable table in this chapter to a date example. Construct two synthetic character vectors representing the same three dates under a month-first and a day-first convention, parse each under a single assumed convention, and show which dates are silently swapped. Relate the result to the role of `TZ` and locale in the image.

5. Explain, to a colleague who has never met the idea, why setting `OMP_NUM_THREADS=1` together with `OPENBLAS_NUM_THREADS=1` can change the last digits of a bootstrap confidence interval or a mixed-model estimate. State clearly what is traded away by fixing it, and name one analysis where you would judge the trade worth making.

6. A collaborator reports that after pulling your image and running `renv::restore()`, she still obtains different package versions from yours. The environment is pinned; the packages are not matching. Diagnose the most likely cause, and explain why a container alone, without the lockfile restored inside it, does not guarantee identical packages.

7. (Cumulative) Run `raw <- load_cohort('analysis/data/raw_data/cohort_raw.csv')` and `cohort <- clean_cohort(raw)` from the compendium's root. Report three things: `nrow(cohort)`; the number of distinct values of `cohort$sex` after cleaning; and whether `file.exists('Dockerfile')` evaluates to `TRUE` or `FALSE` there, as 'The cohort, continued' above checks. Then name the instruction in this chapter's showcase Dockerfile that pins the thread count, and say why it sets two variables rather than one.

Draft, November 13, 2026. Content may change.

8. (Spacing) This exercise returns to the lockfile of Chapter 6, ‘Pinning Packages with renv’ (Chapter 6). Explain, in two or three sentences, what a pinned `renv.lock` guarantees about a result and what it does not, and name the specific gap this chapter’s container closes that the lockfile by itself cannot.

7.16. Solutions

Attempt an exercise before expanding its solution. The first asks you to inspect an image on your own machine, so the values will vary; the reasoning is what the box records.

Exercise 1

The tag and the digest are found together by asking the registry, which needs no pull:

```
docker buildx imagetools inspect rocker/r-ver:4.4.0
```

The tag is `rocker/r-ver:4.4.0`. The `Digest:` line near the top is the index digest, the one a `FROM` line pins; the `Manifests:` list below it gives the digest of each platform’s image, `linux/amd64` among them. The values may differ from any printed in this book, because the tag can since have been repointed. A copy already pulled reports the digest it was pulled under, which is not necessarily what the tag resolves to today:

```
docker pull --platform linux/amd64 rocker/r-ver:4.4.0
docker image inspect --format '{{index .RepoDigests
↪  0}}' \
  rocker/r-ver:4.4.0
```

What can change under the tag. A tag is a mutable label, a name someone chose to point at an image. The maintainer can rebuild `4.4.0` to patch a system library, pick up a security update to the base Ubuntu layer, or correct a packaging mistake, and republish it under

Draft, November 13, 2026. Content may change.

the same tag. Everything about the image can change: its system libraries, its preinstalled packages, its locale data, even its R patch level. What cannot change is the image named by the digest, because the digest is a content hash: an altered image hashes differently and is therefore a different image by construction, not by convention.
A reproducible recipe pins the digest because it wants that guarantee. Pinning the tag records an *intention* about which image to use; pinning the digest records the image. The cost is that a digest-pinned recipe does not receive security updates silently, which is usually the right trade for an analysis and the wrong one for a production service.

Exercise 2

Every reason the recipe fails to reach L2.

1. `:latest` is the most mutable tag there is. The image the recipe resolves to changes without notice, so two builds of the identical Dockerfile can produce different environments.
2. A plain `install.packages()` draws from the live repository, so it installs whatever is current on the day of the build. The package versions are not pinned at all, which means the recipe does not even reach L1.
3. No lockfile is restored, so there is no record of what was installed even after the fact.
4. The determinism variables are absent: no `LANG`, no `TZ`, no thread count. The locale and time zone are inherited from the base and can shift with it.

The first two are the ones to rewrite. The `FROM` line becomes one line, but replacing the live install takes a short block, because a restore needs the lockfile copied into the image and a repository to fetch from:

Draft, November 13, 2026. Content may change.

```
FROM rocker/tidyverse:4.4.0@sha256:3e72037abda2a32d860
 ↪  21bbe5725e727f2c02233e4072557bc70c1e890e3471b

ENV RENV_CONFIG_REPOS_OVERRIDE=https://packagemanager.
 ↪  posit.co/cran/__linux__/jammy/2026-06-27

WORKDIR /project
COPY renv.lock renv.lock
RUN Rscript -e "install.packages('renv', repos =
 ↪  Sys.getenv('RENV_CONFIG_REPOS_OVERRIDE'))" \
 && Rscript -e "renv::restore(library = .libPaths()[1],
 ↪  prompt = FALSE)"
```

The `FROM` line pins the image by its own digest, the one `rocker/tidyverse:4.4.0` resolved to on 2026-10-11. A digest names exactly one piece of content, here an index whose only image is for `linux/amd64`, so it cannot be borrowed from another image such as `rocker/r-ver`. When both a tag and a digest are given, Docker resolves the digest, and the tag is only a label for the human reader. The image is built on Ubuntu 22.04, so the snapshot names `jammy`, and it is published for `linux/amd64` only, so on an Apple Silicon machine it runs under emulation and must be built with `--platform linux/amd64`. The block then copies the committed lockfile into a working directory, installs `renv` from the dated snapshot, and restores the recorded package set, fetching from that snapshot. We built this recipe on 2026-10-11 against a one-package lockfile (`jsonlite` 2.0.0) and the image reported `jsonlite` 2.0.0. A fuller repair would add the locale, time-zone, and thread-count variables and check the `renv` version, as the Dockerfile in the chapter does. Nevertheless, these lines are the ones that carry the recipe from unpinned to pinned.

Exercise 3

Data analysis: tidyverse, because the analysis draws most of its stack from the base rather than the lockfile and the base saves the install. **Manuscript:** publishing, because rendering to PDF needs

Draft, November 13, 2026. Content may change.

the LaTeX toolchain that `rocker/verse` carries and the lighter bases do not. **R package:** minimal, because a package must be checked against a clean environment carrying only its declared dependencies, and a rich base can mask a missing declaration. **Simulation:** minimal, because a simulation typically needs a small numeric stack and image size matters when the image is pulled onto many cluster nodes. **Blog:** tidyverse or publishing, depending on whether posts render to HTML alone.

The two genuinely arguable cases.

Data analysis. Tidyverse buys a fast build and a base whose contents are already tested together. Minimal buys a smaller image and, more importantly, honesty: everything the analysis uses comes through the lockfile, so the lockfile is a complete record rather than a supplement to an opaque base. The trade is build convenience against completeness of the record.

Blog. Publishing carries LaTeX, which a blog rendering only to HTML never uses, and it is a large image to pull and store for that. Tidyverse is lighter but leaves the blog unable to render a PDF version of a post without a rebuild. The trade is image size against format optionality, and it turns on whether any post will ever need PDF.

The general shape: a heavier base is faster to start with and less honest about what the analysis actually depends on.

Exercise 4

```
library(tibble)
dates <- c('01/02/2026', '03/04/2026', '05/06/2026')
tibble(
  written     = dates,
  as_us       = format(as.Date(dates, '%m/%d/%Y'), '%d
   ↪  %B %Y'),
  as_european = format(as.Date(dates, '%d/%m/%Y'), '%d
   ↪  %B %Y')
)
```

Every one of the three parses successfully under both conventions and means something different under each: `01/02/2026` is 2 January

Draft, November 13, 2026. Content may change.

under the day-first reading and 1 February under the month-first reading. No error is raised, because both readings are valid dates. This is the same failure mode as the decimal-mark table earlier in the chapter, and it is worse in one respect. The decimal-mark case produces visibly implausible magnitudes, whereas a swapped date is simply a different plausible date.

The dates that are *silently* swapped are exactly those where both components are twelve or under, which is roughly a third of a calendar year. A date such as `25/12/2026` fails to parse under the month-first convention and so announces itself; `01/02/2026` does not.

The relation to `TZ` and locale. Neither variable changes how an explicit format string parses, so a script that writes `%d/%m/%Y` is safe under any locale. They bite where the format is *inferred*: a locale-dependent parser, a spreadsheet export, or a `read_csv()` guessing a column type will resolve the ambiguity differently under different locale settings. Fixing `LANG` and `TZ` in the image removes that variability, which is why the chapter sets them. The stronger discipline is to state the format explicitly and never rely on inference, and to prefer ISO 8601 in any file the analysis writes, since `2026-02-01` is unambiguous under every convention.

Exercise 5

Why the digits move. Floating-point addition is not associative. Adding a long column of numbers left to right can give a slightly different answer from adding it in pairs and combining, because each intermediate result is rounded. A multithreaded numerical library splits the sum across threads and combines the partial results, and the order in which the partials arrive depends on the thread count and on scheduling. So the same data can produce answers differing in the last few digits between a one-thread and an eight-thread run, and sometimes between two eight-thread runs. A bootstrap interval or a mixed-model estimate is built on many such reductions, so the difference propagates to the reported bounds.

Setting `OMP_NUM_THREADS=1` together with `OPENBLAS_NUM_THREADS=1`, for the OpenBLAS the Rocker image carries, removes the variability by removing the parallelism: there is one summation order and it is

the same every time on that image and CPU. Either variable alone can leave a source of threads in place: `OPENBLAS_NUM_THREADS` alone leaves other OpenMP code threaded, and `OMP_NUM_THREADS` alone yields to any `OPENBLAS_NUM_THREADS` set elsewhere.

What is traded away. Speed, and potentially a great deal of it. A mixed model on a large dataset, or a bootstrap with many replicates, can run several times faster across eight threads. Fixing the thread count to one converts a determinism problem into a wall-clock problem.

Where the trade is worth making. Any analysis whose outputs will be verified against recorded values, which is to say anything reaching for L3, since the verification compares numbers and the numbers must be stable. The concrete case is a regulatory submission. A fixed thread count for the final analysis makes the reported estimates reproducible bit for bit, and the extra hours are irrelevant against the cost of a discrepancy an auditor cannot explain. For exploratory work during development, the trade is usually not worth making.

One refinement: you can develop with threads and fix the count only for the final verified run, which recovers most of the speed and all of the determinism where it matters.

Exercise 6

The most likely cause is that `renv::restore()` did not restore into the library the session is actually using, or did not run against the lockfile the collaborator believes it did. In practice this resolves to one of a few things. The project may have been opened from a directory where the lockfile is absent, so `restore()` had nothing to read. The container may mount a host `renv` cache or library that shadows the project library. `RENV_PATHS_LIBRARY` may point somewhere unexpected inside the image. Or the lockfile itself may not have been committed, so the collaborator restored an older version of it.

The diagnostic sequence is `renv::status()` to see whether the library and the lockfile agree, then `.libPaths()` to see which library is in force, then `renv::project()` to confirm which project `renv` believes it is in. Between them these locate all four causes.

Draft, November 13, 2026. Content may change.

Why the container alone does not guarantee identical packages. A container pins the environment: the operating system, the system libraries, and the R runtime. The packages installed into the image at build time are pinned by the image. However, any package the analysis installs or restores *afterward* is not, because it is resolved at run time against whatever the lockfile or the repository then says. The image and the lockfile do different jobs: in the chapter's framing, the image is the L1-to-L2 step and the lockfile is the L0-to-L1 step. The compact statement: the container guarantees the same *floor*, the lockfile guarantees the same *packages*, and an analysis needs both.

Exercise 7

The solution for this exercise is in the instructor guide, which is distributed separately to course instructors. Self-study readers who want to check their answer can compare the three autogradable values against those reported in the chapter's own 'The cohort, continued' section.

Exercise 8

A pinned `renv.lock` guarantees that every R package the analysis loads is reinstalled at the exact version recorded, from the repository recorded, so two restores of the same lockfile install the same packages. It guarantees nothing about the operating system, the system libraries a package links against, or the version of R itself that runs those packages. None of those three is an R package. The lockfile's `R` block records the R version, but the lockfile can install none of the three. The specific gap this chapter's container closes is exactly those three: the `FROM` line fixes the operating system and the system libraries, and the digest-pinned Rocker base fixes which R runtime runs the restored packages. The lockfile answers 'which packages' and the container answers 'on what'. Consider this chapter's own orientation story: the GDAL-dependent package that failed to install on a collaborator's machine even though the lockfile pinned its version exactly. That is a failure the lockfile could not prevent and the container does.

Draft, November 13, 2026. Content may change.

Answers to the checks

Check 1. *from earlier chapters* (page 255)

The three failures after restoring one lockfile on two laptops sit in three different layers. Case (a) is the operating-system and system-library layer. A lockfile does not pin the system libraries that packages link against (Chapter 6), so this layer is closed only at L2, by a pinned environment. Case (b) is the numerical-library layer, the BLAS and LAPACK. No rung guarantees it (Chapter 3); fixing the thread count narrows it, and a stated tolerance absorbs what remains. Case (c) is session configuration. The `.Rprofile` overrides the default sampler with `Rounding`, the sampler R used before 3.6.0 (Chapter 2). An override is fixed only by an explicit `RNGkind()` call in the analysis code, not by the lockfile.

Check 2. *what the lockfile missed* (page 257)

The collaborator's restore failed because a package linked against GDAL. GDAL is a system library the lockfile cannot name, because it is not an R package. The third machine's different number came from an older R release in which a numerical default differed; the lockfile pins package versions, not the R version they run under. Both failures sit beneath the package layer, on the system-library layer and the R-runtime layer, which is the gap a container closes.

Check 3. *whose decision* (page 259)

The choice of base must match the archetype, not the maximum. A bigger image is not free. It inflates the image, slows every rebuild, and, more importantly, obscures what the analysis actually depends on, because anything the base happens to carry no longer shows up as a declared dependency in the lockfile. An R-package archetype that builds on a publishing base, for instance, could pass its checks while silently depending on a LaTeX toolchain its own declared dependencies never mention. The analyst chooses the smallest base that carries the real requirement.

Check 4. *three layers, one tool* (page 260)

The operating system supplies the C library and the locale and sorting conventions every package was compiled against, and it is not installed by `install.packages()`. The system libraries, such as GDAL or `libxml2`,

Draft, November 13, 2026. Content may change.

are compiled non-R software that packages link against but do not bundle, so `renv` has nothing to record about them. The R interpreter itself is the program the lockfile's `install.packages()` calls run inside, not a package those calls install. Its version is recorded in the lockfile's `R` block, but the lockfile cannot install it. All three need something beneath the package manager, which is the container's job.

Check 5. *image or container* (page 262)

There is no recovery feature. A container is a running instance of a read-only image, and changes a container makes to its filesystem are ordinarily discarded when it stops. Restarting produces a fresh container that begins again from the pristine image, in which the file was never deleted. The image is durable and the container is ephemeral. This is the same property that makes a container reproducible: every run starts from the identical frozen state. It also means that anything one wants to keep, such as edited source or output, must live outside the container, on a host directory made visible inside it.

Check 6. *why not start bare* (page 263)

A hand-rolled install recreates work the Rocker project has already done, tested, and kept in step with R releases, and is slower to build because it shares no cached layers with any other project. More to the point for reproducibility, two analysts who each write their own installation steps are two independent sources of small decisions. Examples are which compiler flags or which locale packages get installed. A shared Rocker base fixes those decisions identically for everyone who builds on it. Building on the same base image turns those into one decision made once, by the Rocker maintainers, rather than many decisions made silently and differently by each analyst.

Check 7. *tag versus digest versus snapshot* (page 271)

Not honest as an L2 claim. The digest pins the operating system, the system libraries, and the R runtime, which is genuine progress on the two layers a lockfile cannot reach. But a plain `install.packages()` against the live repository still resolves 'the current version' to whatever is current on the day of that particular build. The recipe can therefore produce a different set of packages on two builds months apart, even though the `FROM` line is identical both times. Pinning the base and dating the package source

Draft, November 13, 2026. Content may change.

are two separate refinements, and a recipe that does only the first has not finished the second.

Check 8. *emulation and correctness* (page 273)

The worry conflates two separate things. The nine-minute build is QEMU translating `amd64` instructions into `arm64` instructions, because the flag asked this `arm64` host for the `amd64` image. That cost is real, but it is a cost in time, not in correctness, since a translated `amd64` instruction still computes the `amd64` result. What actually decides trustworthiness is whether the `FROM` line names a digest rather than a tag, and whether both builds request the same platform. Here both do: the flag gives the student the `amd64` entry, and the labmate's Intel laptop is an `amd64` host, so both laptops build on the identical base filesystem and restore the same packages, one of them slowly. The build time difference carries no information about whether the two containers agree. Had the student dropped the flag, the build would have been fast, native, and on a different base image. Had the `FROM` line named a tag, the two builds could silently diverge regardless of how long either one took, which is the whole reason this chapter pins by digest.

Check 9. *which variable* (page 276)

The likely culprit is `TZ`, the time zone. If the image does not fix it, the run can supply one: a `-e TZ=...` flag, a mounted `/etc/localtime`, or Apptainer, which binds the host's `/etc/localtime` by default. An event recorded near midnight is then assigned to different calendar days in Berlin and San Diego. The daily buckets then differ at the day boundaries, which is exactly the pattern described. Setting `TZ` in the image, conventionally to `UTC`, makes the day assignment independent of the host, so both collaborators bucket the events identically. Note that identical package versions, that is, a shared L1, did nothing to prevent this: the failure is an environment failure, which is what the container layer exists to fix.

Check 10. *choosing a profile* (page 277)

A package must be checked against a clean environment carrying only its declared dependencies, so that `R CMD check` or its equivalent catches a dependency the `DESCRIPTION` fails to declare. A rich publishing base can mask exactly that failure. The package's code may rely, without declaring it, on a LaTeX tool or a system library the publishing base happens to carry.

Draft, November 13, 2026. Content may change.

The check will then pass in this image and fail the moment the package is installed on a user's machine that lacks that base. The minimal profile is the honest choice because it cannot hide an undeclared dependency behind extra software nobody asked for.

Check 11. *why not rebuild every time* (page 278)

Saved time is real but not the main point. Building a comprehensive image once and pinning it by digest means every project in the lab, and every member working on any of them, starts from the same filesystem, provided all of them pull the same platform. The alternative is independently resolved installs, which could drift even when every project names the same package versions in its own lockfile. Rebuilding from the raw base every time reintroduces the chance that two builds diverge, when they run on different days against a live snapshot or a different system-library patch level. No lockfile would catch such a divergence, because it sits below the package layer. The shared team image is what makes 'works on my machine' inapplicable across the whole lab, not only within one project.

Check 12. *what changed and what did not* (page 280)

Only the `FROM` line changed, from the digest-pinned `rocker/r-ver` to the digest-pinned `rocker/verse`. Every other reproducibility-relevant line stayed fixed: the locale and time-zone and thread-count variables, the dated snapshot wired into the repository options, and the `renv::restore()` that installs the locked packages. This is the point of keeping the profiles cleanly separated from the rest of the recipe: changing what the deliverable is changes one line, not the discipline that makes either build reproducible.

Check 13. *complete the Dockerfile* (page 281)

The four missing lines of the worked example's Dockerfile are these.

```
FROM rocker/r-ver:4.4.0@sha256:73adabf378930f8744f56bac1fe
 ↪  5e99870532eb4ced1a15075991bac909d21f3
ARG PPM_SNAPSHOT=https://packagemanager.posit.co/cran/__li
 ↪  nux__/jammy/2026-06-27
    OMP_NUM_THREADS=1 \
    OPENBLAS_NUM_THREADS=1 \
RUN Rscript -e "renv::restore(library = .libPaths()[1],
 ↪  prompt = FALSE)"
```


Draft, November 13, 2026. Content may change.

Line (1) takes the digest, not the tag alone, because a tag is a mutable pointer that a maintainer rebuild can move. The digest names an image index, so it fixes one filesystem only together with `--platform linux/amd64`. Line (2) takes a dated snapshot naming `jammy`, because an undated URL can resolve differently from one build to the next, and a `noble` URL would install binaries built against other system libraries. Line (3) sets both thread counts, not `OMP_NUM_THREADS` alone, because the `rocker/r-ver` image links R against OpenBLAS, which reads `OPENBLAS_NUM_THREADS` ahead of `OMP_NUM_THREADS`. Line (4) restores the lockfile, because without it the image fixes the environment but not the package versions. The recipe would then not reach L1, and since the rungs are cumulative it would not be L2 either. The `library` argument puts the packages where every session in the container finds them.

7.17. Cheat sheet

Line or rule	What it does
`FROM image@sha256:...`	Pins the base by immutable content digest, not a mutable tag; for a multi-architecture index, it fixes one filesystem only together with a fixed `--platform`.
`ARG PPM_SNAPSHOT=.../jammy/YYYY-MM-DD`	Names a dated package-repository snapshot matching the base's Ubuntu release.
`ENV RENV_CONFIG_REPOS_OVERRIDE=...`	Forces `renv::restore()` to fetch from the dated snapshot (habit 2).
`ENV LANG=... LC_ALL=...`	Fixes locale-dependent sorting and formatted output.
`ENV TZ=UTC`	Fixes day-boundary assignment for date/time arithmetic.
`ENV OMP_NUM_THREADS=1, OPENBLAS_NUM_THREADS=1`	Removes thread count as a source of floating-point variation, trading speed for stability.
`WORKDIR /project`	Keeps `renv` out of the image root, where it refuses to initialize.

Draft, November 13, 2026. Content may change.

Line or rule	What it does
`COPY renv.lock renv.lock` + `renv::restore()`	Restores the exact package set inside the fixed environment.
`docker build --platform linux/amd64`	Selects the `amd64` entry of an index, so a Mac builds the same image as CI and the cluster; required on Apple Silicon for `amd64`-only bases such as `rocker/tidyverse:4.4.0`.
`docker buildx imagetools inspect image:tag`	Asks the registry for the index digest a tag currently resolves to, and each platform's manifest digest.
`docker image inspect --format '{{index .RepoDigests 0}}' image:tag`	Reports the digest a local copy was pulled under, which may be stale.
Rule of thumb	Compare content (`renv::status()`, package lists) across two builds to verify pinning; never compare digests, which differ by default even when content is identical (habit 4).

7.18. Further reading

The foundational case for containers in reproducible research is Boettiger (2015), which introduces Docker to a scientific-computing audience and is the natural first read for anyone new to the tool. The Rocker images themselves are documented by Boettiger & Eddelbuettel (2017), and the wider ecosystem of R containerization tooling is surveyed thoroughly by Nüst, Eddelbuettel, et al. (2020); both are worth consulting before building a base image of one's own.

On making a container reproducible, the essential reference is Nüst, Sochat, et al. (2020), whose ten simple rules for writing Dockerfiles argue for version-specific image tags over `:latest`, explicit version pinning, and much else this chapter has drawn from. The published rules stop short of recommending a digest pin or a dated package snapshot (inspected); this

Draft, November 13, 2026. Content may change.

chapter adds both as a stronger discipline on top of that checklist. Read the ten rules as a baseline against any Dockerfile one is asked to trust.

For the division of labor between the container and the package layer, return to Ushey & Wickham (2026) on `renv`, keeping in mind that its remit stops exactly where this chapter's begins. The compendium structure into which both fit is the subject of Marwick et al. (2018), whose `rrtools` pairs a packaged analysis with a Dockerfile; its template has restored an `renv` lockfile since 2022. The alternatives to Docker as a runtime are taken up in Chapter 8. They range from the high-performance-computing runtime Apptainer (formerly Singularity) (Kurtzer et al., 2017) to the browser-based Binder (Project Jupyter et al., 2018) and the container-free Nix. The team-image model sketched here is developed in full in Chapter 13.

Draft, November 13, 2026. Content may change.

8. Backends and Runtimes: Nix, Podman, and Apptainer

Using Singularity containers, developers can work in reproducible environments of their choosing and design, and these complete environments can easily be copied and executed on other platforms.

Kurtzer, Sochat, and Bauer, *Singularity: Scientific Containers for Mobility of Compute* (2017)

By the end of this chapter you will be able to take any of three situations and name the one alternative to Docker that fits, chosen by the constraint that actually binds. The situations are a job a cluster's scheduler refuses to run under Docker, a laboratory that has declined to license Docker Desktop, and a reviewer who will install nothing. The alternatives are Podman, Apptainer, Nix, the conda family, and Binder, each a route to the pinned-environment rung that Chapter 7 reaches with Docker.

Before you start

Reading time. About 45 minutes.
Prerequisites. Chapter 7, for the Docker recipe, the digest pin, and the dated snapshot that every alternative in this chapter must match to count as reaching the same rung. Also Chapter 3, for the L0-L3 ladder and the capture-versus-validation vocabulary used throughout. This chapter is marked optional in the one-quarter core path (see the Conventions page); a reader on that pace can skip it without losing the ladder itself, which Chapter 7 alone already secures. No new statistics are introduced. R knowledge assumed: calling a function with named arguments and reading a short shell command; no new R idiom appears.
Objectives. By the end of this chapter, can you:

Draft, November 13, 2026. Content may change.

- explain why Docker is one route to the pinned-environment rung and not the only one?
- place a given tool, Podman, Apptainer, Nix via `rix`, conda, Binder, or a devcontainer, on the correct one of the three axes (runtime, backend, remote execution), and say what that axis does and does not change?
- run a container image under Podman or Apptainer, and read a small `default.nix` declaration produced by `rix`, without conflating what each one pins?
- given a concrete constraint, a cluster with no administrative rights, a Windows-only laboratory, a reviewer who installs nothing, name the one tool that fits, and state honestly which rung it reaches?

Outline. We begin with where the chapter sits on the stack: the same two layers Docker captures, reached by a different route. The orientation then introduces three health scientists Docker cannot serve directly, and the analyst's contribution follows, which is to diagnose the binding constraint before picking a tool. The three axes come next in turn. Axis one is a different runtime, Podman or Apptainer, over the same image; axis two is a different environment model, Nix, Guix, or conda, without a container. Axis three is remote execution, Binder or a devcontainer, off the analyst's machine. We then choose among the alternatives with a comparison table keyed to the binding constraint, and ask which route would apply to this book's own compendium today. The chapter closes with a worked example of one pinned image and three destinations and a second, Apptainer under Slurm on a cluster, followed by collaborating with an LLM, exercises, a cheat sheet, and further reading.

Check 1: from earlier chapters

A colleague rebuilds your analysis image from its Dockerfile six months after you first built it, and a fitted coefficient changes in the third decimal. The recipe opens with `FROM rocker/r-ver:4.4.0`, a tag with no digest, and installs packages with a plain `install.packages()` against the live repository. The project has no `renv.lock`. Using

Draft, November 13, 2026. Content may change.

Chapter 3, Chapter 6, and Chapter 7, name each layer that could have moved, the rung the project actually reached, and what would close each gap.
The answer is on page 332.

8.1. Where this sits on the stack

This chapter captures nothing beyond what Chapter 7 already pinned. Its subject is reaching the identical rung by a different road. Figure 8.1 marks the same two layers Docker's chapter captures, the operating system and system libraries, and the R runtime. Here they are reached by Podman, Apptainer, Nix, conda, or a remote service instead of a locally built Docker image.

```
plot_determinant_stack(
  captures = c('Operating system, system libraries', 'R
   ↪  runtime'),
  title = 'Reaching the same pinned-environment rung by a
   ↪  different route')
```


Draft, November 13, 2026. Content may change.

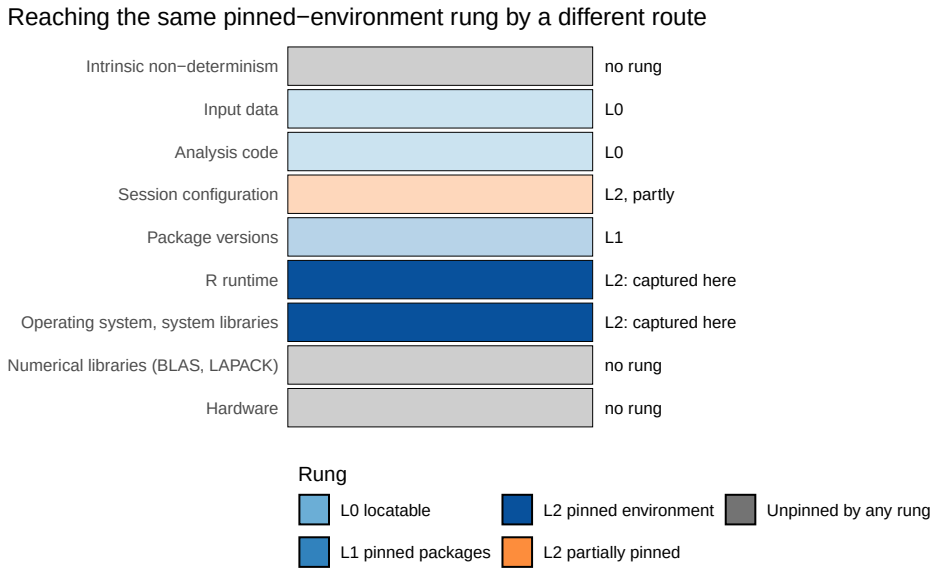


Figure 8.1.: The determinant stack of Chapter 3, with the operating system, system libraries, and R runtime marked as captured, the same two layers Chapter 7 captures with Docker. This chapter reaches the identical pinned-environment rung, L2, by Podman, Apptainer, Nix, conda, or a remote service instead of a Docker build: the same rung, by a different route, not a new capability.

8.2. Orientation

In Chapter 7 we reached the pinned-environment level, L2 on the ladder of Chapter 3, by packaging an analysis inside a Docker image. Such an image fixes the operating system, the system libraries, and the R interpreter that a lockfile alone leaves to chance. It is easy to come away from that chapter believing that Docker is reproducibility, that the two words name the same thing. They do not. Docker is one instrument for reaching L2, and a reader who mistakes the instrument for the goal will be at a loss the first time Docker is unavailable, unwanted, or forbidden.

Consider three health scientists who cannot simply follow the Docker recipe of the previous chapter. The first runs a large simulation study that must

Draft, November 13, 2026. Content may change.

execute on a shared university cluster, where the scheduler will not grant her the administrative privileges that Docker's standard, root-owned daemon requires. The second works in a laboratory of Windows machines and a pipeline that mixes R with Python, a combination that sits awkwardly inside the Rocker images. The third has prepared a teaching blog and wants a reviewer to re-run a single post in a browser, installing nothing at all. Each needs the pinned environment that L2 names; none of them can obtain it in the ordinary Docker way. In this chapter we shall see that the ladder does not depend on any one tool, and how to choose among the alternatives by the constraint at hand.

The framework this book uses organizes those alternatives along three axes. The organization is worth internalizing before any command is typed, because it prevents the common error of treating every unfamiliar tool as a special case. The first axis changes only the runner: a different program runs the very same image, and nothing about its pinning, its packages, or its base is reinstalled. The second axis changes the environment model itself, dispensing with the container and declaring the environment directly, so that the pinned environment is reached by a different mechanism. The third axis keeps the container but moves it off the analyst's machine, into a browser or a hosted service, so that the person reproducing the work provides no environment of their own. A tool belongs to exactly one axis, and knowing which axis a tool occupies tells the analyst almost everything about what it does and does not change. We take up the axes in that order once the analyst's contribution has been stated.

8.3. The analyst's contribution

No tool in this chapter chooses itself. Three judgments remain the analyst's, and they are the substance of the chapter.

1. **Diagnose the constraint before selecting the tool.** The decisive question is usually which constraint is binding. Candidates include an absence of administrative rights on a cluster, an operating system, a mixed-language pipeline, a licensing term, a reviewer's patience, or a demand for maximum rigor. The analyst identifies the binding constraint first, and the constraint usually selects the axis, and often the tool.

Draft, November 13, 2026. Content may change.

2. **Locate each alternative on the correct axis.** A great deal of confusion in practice comes from treating Podman, Nix, and Binder as interchangeable answers to one question. They are answers to three different questions. The analyst who keeps the axes distinct will not, for example, expect Podman to solve a Windows-and-Python problem, nor expect Nix to spare a reviewer an installation.

3. **State the level each option actually reaches.** A Podman run of a digest-pinned image reaches L2 exactly as the Docker run did, and a Nix declaration reaches L2 without a container. A Binder session reaches L2 for the duration of the session but pins nothing on the reviewer's own machine. The analyst reports the level honestly and never lets the novelty of a tool inflate the claim.

ℹ The vocabulary of this chapter

- **Runtime.** The program that runs a container image: Docker, Podman, or Apptainer. The runtime is interchangeable when the image is a standard OCI image.
- **OCI image.** An image built to the Open Container Initiative specification, the common format that Docker, Podman, and Apptainer can all consume. A Docker image is an OCI image.
- **Daemon.** A background system service that runs continuously with elevated privilege. A standard Docker installation relies on one; Podman and Apptainer do not, which is why they run without administrative rights. Docker also has a rootless mode, but its prerequisites are usually installed by an administrator, so on a shared cluster it is rarely available to an ordinary user.
- **Rootless.** Able to run under an ordinary user account, without the `root` superuser privilege. A requirement on shared clusters, where users are not administrators.
- **Backend.** The mechanism that supplies the environment. Docker is the container backend; Nix is a container-free backend that declares the environment directly.
- **Purely functional package manager.** A package manager, such as Nix, in which the installed result is treated as a function of the declared inputs, so that the same declaration, on the same platform, resolves to the same set of packages and dependencies.

Draft, November 13, 2026. Content may change.

- **Input-addressed.** Identified by a hash of the build inputs rather than by a mutable name, so that a reference cannot silently point at different content later.
- **SIF.** The Singularity Image Format, a single-file archive that Apptainer produces from an image, suitable for depositing alongside a compendium.

Check 2: the third judgment

A biostatistician discovers Nix and begins describing her project as 'fully reproducible using Nix,' before running anything twice. Which of the analyst's three judgments has she skipped?
The answer is on page 332.

8.4. Axis one: a different runtime over the same image

We begin with the cheapest of the three changes, because it changes the least. The image built in Chapter 7, pinned to its base by content digest and installing its packages from a dated snapshot, is a standard OCI image. Docker is the program that ran it, but the image does not belong to Docker; any OCI-compatible runtime can run it. Selecting a different runtime changes one parameter. The base image, the digest pin, the dated snapshot, and the lockfile all carry over unchanged, and only the command that launches the container differs.

8.4.1. Podman

Podman is a daemonless, rootless, Docker-command-compatible runtime. Where Docker interposes a background daemon running with elevated privilege, Podman runs the container directly under the invoking user's account. Its command-line interface is close enough to Docker's that `podman run` stands in for `docker run` in most cases:

Draft, November 13, 2026. Content may change.

```
$ podman build -t myproject .
$ podman run --rm -v "$(pwd)":/project myproject \
    Rscript -e 'source("analysis/run.R")'
```

Two properties recommend Podman to the health scientist. The first is that, being daemonless and rootless, it runs on a multi-user Linux machine where an analyst has no administrative rights, a situation in which Docker's daemon is often simply unavailable. The second concerns licensing, and it catches many university users unawares. Docker Desktop, the packaged Docker distribution for macOS and Windows, is under its current terms not free for use by larger commercial and institutional organizations, and many universities fall within those terms. The terms do carve out an exception for education, but institutions differ on whether a given use qualifies. The point is contested, which is why many institutions either license Docker Desktop or avoid it. Podman carries no such restriction. For an analyst whose institution has declined to license Docker Desktop, Podman reaches the same L2 environment, from the same image, at no licensing cost.

8.4.2. Apptainer

Apptainer, known until its move to the Linux Foundation as Singularity, is the runtime built for high-performance computing (Kurtzer et al., 2017). Kurtzer and colleagues designed it for the setting that defeats Docker on a cluster. It runs unprivileged, integrating with the batch schedulers, message-passing libraries, and graphics processors that scientific clusters provide. A user with no administrative rights can nonetheless run a containerized analysis under the cluster's job system.

Two rules govern its use, and observing them keeps the image pinning of Chapter 7 intact. The first is that an Apptainer image should be built from the digest-pinned Docker image, never authored independently, so that the build recipe stays pinned end to end. The SIF file Apptainer produces gets its own content hash, computed over the converted single-file format, not the same bytes as the Docker digest. The two checksums therefore differ. (Apptainer does not claim to preserve the OCI digest identity into the SIF hash; what is pinned is the input to the conversion, not the output's checksum.) The build and a subsequent run look as follows:

Draft, November 13, 2026. Content may change.

```
$ apptainer build env.sif \
    docker://ghcr.io/lab/env@sha256:9f2c...

$ apptainer exec env.sif \
    Rscript -e 'source("analysis/run.R")'
```

The second is that the single file Apptainer produces, the SIF, is itself an archival artifact. Unlike a Docker image, which lives in a layered local store, an Apptainer image is one file that can be checksummed, deposited in a data repository, and cited alongside the compendium. It is thus both the thing that runs the analysis on the cluster and the thing that is archived to let others run it later. That is a convenient coincidence for the verification of Chapter 12.

The expected pattern, and the one we recommend, is to develop on the laptop with Docker or Podman and to run at scale on the cluster with Apptainer, all from a single image. Nothing is reinstalled for the cluster. Apptainer repackages the pinned image into SIF, carrying the content over from the same digest and converting only the format, so the same pinned environment runs under a different program.

This is the point at which the archetypes of Chapter 5 begin to matter. A **simulation study**, which may need to run thousands of replicates across the nodes of a cluster, is the archetype that most naturally reaches for Apptainer. The SIF file travels to the cluster, the scheduler runs it unprivileged across many cores, and the same file is later deposited as the archival record of the environment. A plain **data-analysis** project that runs comfortably on a laptop has no such need and is better served by Docker or Podman.

Check 3: what changes with the runtime

An analyst has a Docker image pinned as `FROM rocker/verse@sha256:...` with packages installed from a dated snapshot. She switches from Docker to Podman to avoid a licensing constraint. Which of the following change: the base image, the pinned digest, the installed package versions, the reproducibility level reached?
The answer is on page 332.

8.5. Axis two: a different environment model without a container

The second axis is a deeper change. Here we abandon the container altogether and declare the environment directly, reaching the pinned environment by a different mechanism.

8.5.1. Nix and the rix front-end

Nix is a purely functional, input-addressed package manager (Dolstra et al., 2004). 'Purely functional' means that the installed environment is a function of a declaration: given the same declaration, Nix reconstructs the same package set and dependency graph on the same platform, at a later time. 'Input-addressed' means that every built component is identified by a hash of its inputs. A reference therefore cannot silently come to mean something else, which is the mutable-tag failure we guarded against in Chapter 7 by pinning digests. Input addressing fixes what goes into each build, not that every build produces bit-identical output, and the platform still matters. Dolstra, in the work that introduced Nix, made this purely functional model the organizing principle of the design, and that property is what recommends Nix here.

The decisive fact for our purposes is that a single Nix declaration fixes the R version, the R packages, and the system libraries together. It therefore reaches the pinned-environment level, L2, without a container at all. This is why the ladder of Chapter 3 speaks of 'a Docker or Nix image'. Nix is not a supplement to the container, in the way that `renv` is a supplement, but a peer of it, an alternative road to the same rung.

The obstacle is that the Nix language is unfamiliar to almost every biostatistician, and this is the dominant reason it is not the default. The `rix` package removes much of that obstacle by generating the Nix declaration from R, in R idiom that a health scientist already knows (Rodrigues & Baumann, 2024). Rodrigues and Baumann designed `rix` so that the analyst writes a familiar R call and `rix` writes the Nix:

Draft, November 13, 2026. Content may change.

```
library(rix)

rix(
  r_ver = "4.4.1",
  r_pkgs = c("dplyr", "ggplot2", "targets"),
  system_pkgs = NULL,
  git_pkgs = NULL,
  ide = "rstudio",
  project_path = "."
)
```

That call produces a `default.nix` file, the declaration that pins the environment. A reader need not master the Nix language to recognize its shape; a minimal `default.nix` reads, in outline, as follows:

```
let
  pkgs = import (fetchTarball
    "https://github.com/rstats-on-nix/nixpkgs/archive/<dat
     ↪  e>.tar.gz")
    {};
  rpkgs = with pkgs.rPackages; [ dplyr ggplot2 targets ];
  system_packages = with pkgs; [ R glibcLocales ];
in
pkgs.mkShell {
  buildInputs = [ rpkgs system_packages ];
}
```

The pinned `nixpkgs` snapshot, written as `<date>` above, is the analogue of the base-image digest: it fixes the set of package definitions from which the environment is built. Current versions of `rix` point at a dated branch of `rstats-on-nix/nixpkgs`, a fork of `nixpkgs` its maintainers keep for R, rather than at a commit of the main `NixOS/nixpkgs` repository. Since version 0.18.3 the generated `fetchTarball` also carries the archive's `sha256`, so a moved branch fails the build rather than changing the environment; older versions omit it. A full commit hash in place of the date is stricter still. Entering the environment is then a single command, and no image is built:

```
$ nix-shell default.nix
```

Newer Nix workflows prefer a flake, a `flake.nix` paired with a generated `flake.lock` that records the exact pin, the direct analogue of `renv.lock`. Because flakes remain behind an experimental flag in current Nix, the framework tolerates both the classic `default.nix` and the flake form, and the reader will encounter both in the wild.

Nix has trade-offs on both sides. On the plus side, it delivers the entire environment pillar in one mechanism and reaches L2 without the container machinery. Its input-addressed builds pin the whole dependency graph from one declaration, where Docker reaches a comparable pin only through careful digest and snapshot discipline. On the minus side, the Nix model is alien to the audience. Although `rix` hides much of the language, diagnosing a build that has failed still calls for some Nix knowledge. The first build is slow and consumes substantial disk. Nix is a host prerequisite comparable to Docker, requiring Linux, macOS, or the Windows Subsystem for Linux. And `rix` is young enough that its own version must be pinned. These caveats are the reason Nix is carried as an opt-in backend for the analyst who wants maximum rigor, while `renv` with a container remains the accessible default.

8.5.2. Guix and the conda family

Two neighbors of Nix deserve mention, because a reader will meet them and should know where they sit. GNU Guix is Nix's close sibling, sharing the purely functional, input-addressed model, with a smaller R ecosystem. For the health scientist it is best understood simply as Nix's relation, reaching the same rung by the same idea.

The conda family occupies a different middle ground, and it answers two constraints the container route handles awkwardly. Conda (Anaconda Inc., 2023), with the faster resolver `mamba` and the lockfile-oriented `pixi` (prefix.dev, 2024), installs and runs directly on Windows, without the WSL2 or Hyper-V virtual machine a Docker (or Podman) image requires there. This is a practical advantage, but not a guarantee that any given package is available. Conda-forge's Windows (`win-64`) build coverage for R packages

Draft, November 13, 2026. Content may change.

is narrower than its Linux coverage. A pipeline that needs a package without a `win-64` build therefore still needs a fallback, and 'reproduces on Windows' should be read as 'for the packages that conda-forge actually ships there'. Conda also spans R and Python in a single environment, so the mixed-language analyst need not bridge two toolchains. For a Windows-only laboratory running a pipeline that mixes the two languages and whose packages are available on `win-64`, this is a decisive practical advantage. The cost is rigor: conda draws its packages from conda-forge rather than from CRAN or the Posit Package Manager, and it pins less strictly than Nix. It is a middle-ground option, more reproducible than an unpinned install and less exacting than Nix. It is the right choice when its two native strengths, running directly on Windows and spanning two languages, are the binding constraint and the required packages are available there.

The honest level

This chapter describes as many as five routes to L2, the pinned environment: Podman, Apptainer, Nix through `rix`, a Binder session, or a devcontainer. Describing a route is not taking it, and 'The cohort, continued' below computes which, if any, the book's own compendium has taken. Even a route taken is not verified until a clean rebuild is compared against a recorded baseline within a stated tolerance, which is what L3 requires. A `default.nix` or a SIF file in a repository is a claim of L2, no more, until Chapter 12's baseline test has actually been run (habit 7).

Check 4: peer, not supplement

A `default.nix` generated by `rix` pins the R version, the R packages, and the system libraries in one declaration, with no Docker image anywhere. What rung does this reach, and why does the chapter call Nix a peer of Docker rather than a supplement to it?
The answer is on page 332.

8.6. Axis three: remote execution

The third axis keeps the container but moves it off the analyst's machine. The pinning is unchanged; what changes is who runs it. This axis answers one limitation, the learning curve and installation burden that a container imposes on the person reproducing the work; the container format itself is kept.

Binder runs a compendium's container in the browser, so that a reviewer or reader installs nothing whatever (Project Jupyter et al., 2018). Jupyter's Binder service reads the configuration files already present in a well-formed compendium, builds the environment on a hosted server, and serves it as an interactive session reached by a single link. For the health scientist the appeal is that a reviewer clicks a link and is inside the running environment, with no Docker, no Nix, and no local installation of any kind.

Two qualifications follow from how Binder works. First, the service, through its `repo2docker` tool, builds its own image from the repository's configuration files; it does not pull a registry digest. A `.binder/` directory of `install.R` and `runtime.txt` files yields an environment resolved on Binder's server, not the digest-pinned image of Chapter 7. Binder runs that pinned image only when the configuration is a `.binder/Dockerfile` whose `FROM` line names the digest, and the image must also serve Jupyter, which a `rocker/binder` base provides, with `jupyter-rsession-proxy` for RStudio. Second, the public service limits each session's memory and lifetime. The honest level is then L2 for the life of the session, and only when the image is the pinned one. Nothing is pinned on the reviewer's own machine, and the session does not persist. Binder suits demonstration and review; it is not an archival guarantee.

Development containers, and the hosted Codespaces service built on them, bring the same pinned environment into an editor rather than a browser (Development Containers Project, 2022). A `.devcontainer/` configuration in the compendium describes the environment, and a compatible editor or the Codespaces service builds and opens it. A collaborator then edits and runs inside the pinned environment without assembling it locally. The principle is the same presence-driven one we have met throughout: the configuration file's existence drives the automated build.

Draft, November 13, 2026. Content may change.

Here again the archetype guides the choice. A **blog** or a **manuscript** archetype, whose whole purpose is to be read and its results inspected by others, benefits most from browser execution. A reader of a teaching post, or a reviewer of a manuscript, re-runs the analysis in Binder with nothing to install, which is the low-friction path such archetypes want. A blog that renders many posts gains the most, since each post becomes independently re-runnable by a reader at a link. A **simulation** archetype, by contrast, gains little from browser execution and much from the cluster, which is the Apptainer story of the first axis. The reproducibility machinery is the same across all archetypes; the archetype decides only how it is delivered.

> Check 5: what a Binder session pins
>
> A `.binder/` configuration lets a reviewer run a compendium's main figure in a browser with nothing installed. Does the session pin anything on the reviewer's own machine? What rung does the session itself reach, and for how long?
> *The answer is on page 332.*

8.7. Choosing among the alternatives

With the three axes laid out, the governing question, as the analyst's first contribution insisted, is which constraint binds. We answer it by setting the guidance beside a compact comparison, generated at build time from the same tibble that a reader could inspect directly.

```
library(tibble)
library(knitr)

comparison <- tibble(
  name = c(
    "Docker", "Podman", "Apptainer",
    "Nix (via rix)", "Guix", "conda / pixi",
    "Binder", "Devcontainer / Codespaces"
  ),
  axis = c(
```

```
    "runtime", "runtime", "runtime",
    "backend", "backend", "backend",
    "remote", "remote"
  ),
  pins = c(
    "the image (OS, R, libraries)",
    "the image (OS, R, libraries)",
    "the image, as one SIF file",
    "R, packages, system libraries",
    "R, packages, system libraries",
    "R and Python packages, less strictly",
    "the image (server-side, per session)",
    "the image (in the editor)"
  ),
  container = c(
    "yes", "yes", "yes",
    "no", "no", "no",
    "yes (hosted)", "yes (hosted)"
  ),
  best_fit = c(
    "the default local runtime",
    "licensing constraint; rootless Linux",
    "an HPC cluster; archival SIF",
    "maximum rigor; input-addressed builds",
    "a Nix-aligned site",
    "Windows-only or mixed R and Python",
    "a reviewer who installs nothing",
    "a collaborator editing in the environment"
  )
)

kable(
  comparison,
  col.names = c(
    "Name", "Axis", "What it pins",
    "Container?", "Best-fit setting"
  )
)
```


Draft, November 13, 2026. Content may change.

Table 8.1.: Backends and runtimes for reaching the pinned environment, by axis

Name	Axis	What it pins	Container?	Best-fit setting
Docker	runtime	the image (OS, R, libraries)	yes	the default local runtime
Podman	runtime	the image (OS, R, libraries)	yes	licensing constraint; rootless Linux
Apptainer	runtime	the image, as one SIF file	yes	an HPC cluster; archival SIF
Nix (via rix)	backend	R, packages, system libraries	no	maximum rigor; input-addressed builds
Guix	backend	R, packages, system libraries	no	a Nix-aligned site
conda / pixi	backend	R and Python packages, less strictly	no	Windows-only or mixed R and Python
Binder	remote	the image (server-side, per session)	yes (hosted)	a reviewer who installs nothing
Devcontainer / Codespaces	remote	the image (in the editor)	yes (hosted)	a collaborator editing in the environment

Read as guidance, the table resolves into a short set of recommendations, each keyed to a binding constraint. When the destination is an **HPC cluster** and the analyst has no administrative rights, the choice is Apptainer, built from the same digest-pinned image and run under the scheduler. The SIF file then doubles as the archival deposit. When the setting is a **Windows-only laboratory**, or the pipeline **mixes R and Python**, the choice is the conda family, `pixi` in particular. It reproduces natively on Windows and spans both languages at the cost of strict pinning. When a **reviewer must install nothing**, the choice is Binder, which serves the running environment in a browser for the life of a session. When the project demands **maximum rigor** and the team can absorb the learning curve, the choice is Nix through `rix`, whose input-addressed builds reach

Draft, November 13, 2026. Content may change.

L2 without a container. And when the obstacle is a **licensing constraint**, an institution that has not licensed Docker Desktop, the choice is Podman. Podman reaches the identical L2 environment from the identical image at no licensing cost. The default, for the analyst under none of these constraints, remains Docker with `renv`, the composition of Chapter 7.

> Check 6: match the constraint to the axis
>
> For each situation, name the axis and the tool. (a) A genomics simulation must run ten thousand replicates on a university cluster where you are not an administrator. (b) A reviewer of your manuscript should re-run the main figure without installing anything. (c) Your laboratory runs Windows and your pipeline calls both R and Python. *The answer is on page 333.*

8.8. The cohort, continued

We now step back from the hypotheticals and apply the same three axes to this book's own running cohort compendium, as this repository holds it now. Two of this chapter's three axes presuppose an image already exists to act on; the third does not, and the check below turns on that difference.

```
repo_root <- '../..'
backend_artifacts <- c(
  renv.lock = file.exists(file.path(repo_root,
   ↪  'renv.lock')),
  Dockerfile = file.exists(file.path(repo_root,
   ↪  'Dockerfile')),
  default.nix = file.exists(file.path(repo_root,
   ↪  'default.nix')),
  devcontainer = file.exists(
    file.path(repo_root, '.devcontainer',
     ↪  'devcontainer.json')))
backend_artifacts
#>    renv.lock   Dockerfile  default.nix devcontainer
#>         TRUE        FALSE        FALSE         TRUE
```


Draft, November 13, 2026. Content may change.

Of the four artifacts, the check finds `renv.lock` and `devcontainer` and misses `Dockerfile` and `default.nix`. By the ladder of Chapter 3 the compendium is therefore at L1. The devcontainer file is an instance of axis three, a development container, but its own comments say plainly that it is a stand-in. It starts from a generic `rocker/verse:4.4.0` image, not one built from this compendium's own pinned recipe, because that recipe does not exist yet. It lets a collaborator open an editor already inside a working R environment, at the cost of pinning nothing specific to the cohort analysis itself. Axis one has nothing to act on, because Podman and Apptainer both swap the runtime over an *existing* image, and there is none yet to swap. Axis two is the one route open without waiting on any other chapter. A `default.nix` generated by `rix` could pin the cohort compendium's R version, packages, and system libraries directly, reaching L2 without any Dockerfile. No such declaration has been built yet, so this chapter leaves the compendium at L1; it only widens which door could be opened first.

> Check 7: the open door
>
> Given the artifact check above, which of this chapter's three axes could lift the cohort compendium to L2 today, without waiting for any other chapter's work to land first?
> *The answer is on page 333.*

8.9. Worked example

To make the axes concrete, consider a single research group with one image and three destinations, which is a common situation once a project matures. The group has built a pinned image for a cohort-analysis compendium, `ghcr.io/lab/cohort-env`, resolved to an immutable digest.

On the analyst's own laptop, at an institution that has declined to license Docker Desktop, the image runs under Podman with no change to the image and no licensing cost:

```
$ podman run --rm -v "$(pwd)":/project \
    ghcr.io/lab/cohort-env@sha256:9f2c... \
    Rscript -e 'source("analysis/run.R")'
```


Draft, November 13, 2026. Content may change.

When a sensitivity analysis must be repeated across many simulated cohorts, the work moves to the university cluster. The analyst builds a SIF from the same digest and submits it to the scheduler. Nothing is rebuilt, and the image contents are the same ones pinned on the laptop, because the SIF was built from that one digest and not reauthored. The SIF pins the recipe; it does not control what the cluster passes into the run.

> ⚠ Pitfall
>
> By default, Apptainer binds the user's `$HOME` into the container and passes host environment variables through into the container (Apptainer Project, 2024). A stray `R_LIBS_USER` or locale variable set on the cluster login node can therefore still leak into execution, even though the image itself did not change. The `--cleanenv` flag drops the inherited environment variables, and only those. The host `$HOME` stays mounted, and with it any `~/.Rprofile`, `~/.Renviron`, and user package library under `~/R`, which R inside the container still reads. A verified run therefore adds `--no-home`, binds only the project directory with `--bind "$PWD"`, and starts R with `--vanilla` so that no startup file is read. The combination `--containall --bind "$PWD" --pwd "$PWD"` isolates the same things; without `--pwd` it starts the job in the container's home directory, where relative paths such as `analysis/simulate.R` no longer resolve.

```
$ apptainer build cohort-env.sif \
    docker://ghcr.io/lab/cohort-env@sha256:9f2c...

$ apptainer exec --cleanenv --no-home --bind "$PWD" \
    cohort-env.sif \
    Rscript --vanilla -e 'source("analysis/simulate.R")'
```

Finally, when the paper is submitted, a reviewer wishes to re-run the principal figure without installing anything. The compendium carries a short `.binder/Dockerfile` that starts from the same digest and copies the repository into the session's home directory, as Binder requires. This works only because the group built `cohort-env` on a `rocker/binder` base, so the image already serves Jupyter and RStudio and has the required user:

Draft, November 13, 2026. Content may change.

Binder builds from that file on a hosted server and serves the environment in the browser, reached by a single link in the cover letter. A `.binder/` of `install.R` and `runtime.txt` would instead make Binder resolve an environment of its own, which is not the pin the laptop and the cluster share.

The three destinations used three tools, drawn from two of the three axes, and one image throughout. The pinning was done once, in Chapter 7; this chapter merely ran it in three places. Had the group instead chosen Nix from the outset, the image would have been replaced by a `default.nix`, and the laptop and cluster runs would have entered the environment through `nix-shell` rather than a runtime. The discipline, one pinned declaration exercised in several places, would have been identical.

Check 8: what made three destinations cheap

The research group never touches its Dockerfile again after the original build, across the laptop, the cluster, and the Binder deposit. What single fact about the three axes makes that possible?
The answer is on page 333.

8.10. An HPC worked example: Apptainer under Slurm

This section is supplementary to an already-optional chapter, and the reader may skip it freely if the cluster destination above was enough. It extends that story with the submission mechanics a real cluster imposes, because knowing that Apptainer is the right tool does not by itself tell an analyst how a job reaches the scheduler.

Three realities of a shared cluster shape the script below. First, the analyst has no administrative rights on any node. As explained above, Apptainer runs unprivileged, so a job submitted under the analyst's own account can still execute a containerized analysis, and that is why it is the standard choice on shared clusters. Second, the SIF file typically sits on a shared, read-only filesystem mounted on every compute node. The image is built once, usually on a login node or a separate build host with outbound

Draft, November 13, 2026. Content may change.

network access, since compute nodes are often firewalled from the registry the image is pulled from. Third, and most consequentially, the analyst cannot install anything at the operating-system level once that image is built and placed on the shared filesystem. There is no `apt install` to fall back on from inside a running job. On a cluster, then, the pinning discipline of Chapter 7 is a requirement: the SIF built from the digest-pinned image is the only environment the job will ever see, for the life of the allocation.

A minimal submission script for the cohort-analysis image introduced in the worked example above looks like this:

```
#!/bin/bash
#SBATCH --job-name=cohort-analysis
#SBATCH --partition=general
#SBATCH --time=02:00:00
#SBATCH --mem=8G
#SBATCH --cpus-per-task=4
#SBATCH --array=1-100
#SBATCH --output=cohort-analysis-%A_%a.log

apptainer exec --cleanenv --no-home --bind "$PWD" \
    --env SLURM_ARRAY_TASK_ID="$SLURM_ARRAY_TASK_ID" \
    --env SLURM_CPUS_PER_TASK="$SLURM_CPUS_PER_TASK" \
    /shared/images/cohort-env.sif \
    Rscript --vanilla -e 'source("analysis/simulate.R")'
```

The `#SBATCH` lines are directives Slurm's `sbatch` command reads out of the script before any shell code runs. They specify the partition to queue on, a wall-clock time limit, a memory reservation, a core count, an array of 100 tasks (one per simulated replicate set), and a destination for each task's combined output. In the output name, `%A` is substituted for the array's job ID and `%a` for the task's index. An analyst with access to a compliant cluster would submit the script with `sbatch run-cohort.sh`.

The `apptainer exec` line carries the same flags met in the Pitfall above, applied here to a batch job. Slurm's own job environment carries login-node variables, module loads, and whatever a cluster's site-wide profile sets, into the job by default. This is the batch-system version of the leakage the Pitfall described for an interactive shell. `--cleanenv` drops those

Draft, November 13, 2026. Content may change.

variables, `--no-home` keeps the host home directory and its R startup files and user library out of the container, `--bind "$PWD"` exposes the project directory the job was submitted from, and `--vanilla` stops R reading any startup file. The job then runs against the image's own pinned packages and nothing the cluster happened to load on the analyst's behalf. Because `--cleanenv` also strips the `SLURM_*` variables, the script passes back the two the analysis needs with `--env`. `SLURM_ARRAY_TASK_ID` tells `simulate.R` which replicate set to run, and should also choose its seed. `SLURM_CPUS_PER_TASK` tells it how many cores it was granted. The image of Chapter 7 pins `OMP_NUM_THREADS` and `OPENBLAS_NUM_THREADS` to 1, so the four requested cores are used only if the script itself runs R-level parallel work across that count; a script that does not should request one core.

This script's `#SBATCH` directive syntax is inspected against standard Slurm usage, not verified end to end. No Slurm cluster was available to actually submit it. The script should be treated as an illustrative template to adapt to a given site's partition names and resource limits, not as a script copied and run unexamined.

Check 9: what the script does and does not pin

A colleague asks why the submission script above needs `--cleanenv` when the SIF file was already built from a digest-pinned image. What would leak into the job without it, and what does the flag not protect against?
The answer is on page 333.

8.11. Collaborating with an LLM

A language model is a capable assistant for the mechanics of this chapter and a poor judge of its trade-offs. The commands and recipes it helps with need no data, but they can name it, and the prompts should keep them from doing so.

Draft, November 13, 2026. Content may change.

> **!** Restricted data
>
> One rule comes first (habit 3). Never paste or upload protected health information, identifiable records, or data held under a data use agreement, nor outputs derived from them, into a model or a coding agent unless three conditions all hold. The institution has approved that endpoint for that class of data. The IRB protocol and the consent permit the processing. The data use agreement, or the provider's terms, permits it. If any of the three is unclear, the answer is no.
> The third condition is the one most often missed. Institutional approval of a tool does not override the terms under which a data provider released the data. NIH, for example, treats sharing controlled-access data through prompts with public generative AI tools, which it glosses as third-party tools, as a breach of the data use certification. It treats a model trained on such data as a data derivative, to be shared only with other approved users and not kept after the project closes (National Institutes of Health, 2025a).
> Describe the structure instead: the column names and types, or a small synthetic example generated from the data dictionary, never edited from real rows. A coding agent sees whatever files are in its working tree, so run it in a checkout or container where the real data is not mounted.
> An agent also acts, so confine it to a branch and keep merges, pushes, and deposits for yourself (Section 2.10.1).

Any host path that names a site, study, or participant should be replaced with a placeholder before a command is pasted. For the commands of this chapter, what a coding agent can see runs through the bind mount: whatever `-v` or `--bind` names is visible to an agent that runs them. The following triples show where each stands.

Prompt. 'Convert this `docker run` command to the equivalent Podman and Apptainer commands, preserving the bind mount and the digest-pinned image.'

Watch for. Check that the `--platform` flag survives, that the Apptainer bind syntax is the real one and not an invented `--volume` form, and that the digest, not a mutable tag, was translated. A tag in place of the digest silently undoes the pin.

Draft, November 13, 2026. Content may change.

Verification. Run the translated command and confirm it executes the analysis; confirm the digest, not a tag, appears in the Apptainer `docker://` reference; and confirm the bind mount exposes the working tree as expected.

A better prompt. 'Convert the attached `docker run` command, which uses `--platform linux/amd64`, a digest-pinned image, and one bind mount of the working tree, to Podman and to Apptainer. Keep the digest, the platform, and the mount target unchanged, and add no other mounts. For each command, give the check that shows which image digest ran and which host directory the container could see.'

Prompt. 'Write a `rix::rix()` call that pins R 4.4.1 with dplyr, ggplot2, and targets, producing a `default.nix`.'

Watch for. Check that the answer is the `rix` call requested rather than hand-written raw Nix, that the `nixpkgs` revision, the whole point of the pin, is present, and that every package exists under `pkgs.rPackages`.

Verification. Run the `rix` call, inspect the generated `default.nix` for a concrete pinned revision, and enter the environment with `nix-shell` to confirm the packages load. A declaration that builds is the only evidence that counts.

Prompt. 'My analysis mixes R and Python and my lab uses Windows. Should I use Nix, Podman, or conda?'

Watch for. Check whether the recommendation follows the constraint or a tool's general reputation, and whether it overstates a tool's rigor, for example by presenting conda as reaching the same level as Nix.

Verification. Check the recommendation against the binding constraint: mixed languages on Windows point to the conda family, whatever a tool's general reputation. The analyst, not the model, owns the honest statement of the level reached.

8.12. Exercises

1. Take a Docker image you have built in a previous chapter and run it under Podman without rebuilding it. Confirm that the container

Draft, November 13, 2026. Content may change.

starts and that the analysis runs. State, in one sentence, why the reproducibility level is unchanged.

2. Build an Apptainer SIF file from a digest-pinned `docker://` reference. Compute a checksum of the SIF and explain how that single file could serve both as the cluster runtime and as an archival deposit.

3. Using `rix::rix()`, generate a `default.nix` that pins a specific R version and three packages. Locate the pinned `nixpkgs` revision in the generated file and explain what would change in the built environment if that revision were altered.

4. For each of the following, name the axis and the single most appropriate tool, and justify the choice in one sentence: (a) a laboratory forbidden from licensing Docker Desktop; (b) a manuscript reviewer who must run a figure with no local installation; (c) a simulation study destined for a shared cluster; (d) a Windows-only pipeline mixing R and Python.

5. A colleague claims that adopting Nix makes an analysis 'fully reproducible'. Write a two-sentence reply that corrects the level claim without dismissing Nix, naming the rung Nix reaches and the rung it does not.

6. Consider a simulation archetype and a blog archetype built from the same compendium machinery. Describe which runtime or backend each most naturally uses and why the underlying pinning is nonetheless identical.

7. Run the artifact check from 'The cohort, continued' against this repository (or reconstruct it with `file.exists()`), and report the four `TRUE`/`FALSE` values in the order `renv.lock`, `Dockerfile`, `default.nix`, `devcontainer`. Name the one axis that could lift the cohort compendium to L2 without requiring any other missing artifact to be built first.

8. This exercise returns to the digest pinning of Chapter 7 (Chapter 7). Take any digest-pinned base line from that chapter, such as `FROM rocker/r-ver@sha256:...`, and write (a) the `apptainer build` command that would convert the image it names into a SIF, and (b) one sentence stating why the SIF's own checksum differs from the

Draft, November 13, 2026. Content may change.

Docker digest even though both were produced from the identical pinned recipe.

8.13. Solutions

Attempt an exercise before expanding its solution. The first three are hands-on and their outputs will vary by machine; the boxes record the reasoning and the expected shape of the result.

Exercise 1

```
podman run --rm -v "$PWD":/project -w /project \
  <image>@sha256:... Rscript analysis/run.R
```

The container starts and the analysis runs, with no rebuild, because Podman consumes the same OCI image format Docker produces. The command differs only in its first word.

Why the level is unchanged. The reproducibility level is a property of what is *captured*, and the image is the capture. The base image, its digest, the installed package versions, and the determinism variables are all properties of the image, not of the program that launches it. Swapping the launcher therefore changes nothing that the ladder grades. The analysis sits at L2 under either runtime.

This is the sense in which the runtime is the cheapest of the three axes: it changes the runner and nothing else. One caveat: rootless Podman handles file ownership on bind mounts differently from Docker, so the *mechanics* of the mount may need adjusting even though the environment inside does not change.

Exercise 2

```
apptainer build analysis.sif
 ↪  docker://<image>@sha256:...
sha256sum analysis.sif
```


Draft, November 13, 2026. Content may change.

The build converts the OCI image into a single SIF file. The checksum is a hash of that file.

How one file serves both purposes. As a cluster runtime, the SIF needs no daemon and no root, which is what makes it acceptable on a shared HPC system where the user is not an administrator. The scheduler runs `apptainer exec analysis.sif ...` directly. As an archival deposit, the SIF is a single immutable file with a checksum, which is the shape an archive wants. It can be uploaded to a repository, assigned a DOI, and verified on download by rehashing.

The property that unifies the two roles is that a SIF is a file rather than a service. A Docker image lives in a daemon's storage and is addressed by a name that a registry must resolve. A SIF, by contrast, is bytes on disk that can be copied, hashed, and deposited like any other artifact. That is why the chapter treats it as the archival form of a container.

Two qualifications apply. The SIF is opaque: it records the environment without recording how it was made, so the Dockerfile remains the source of truth and the SIF is the built product. And SIF files are large, often several gigabytes, which some archives will not accept.

Exercise 3

```
rix::rix(
  r_ver = '4.4.0',
  r_pkgs = c('dplyr', 'ggplot2', 'targets'),
  project_path = '.'
)
```

The pinned `nixpkgs` snapshot appears near the top of the generated `default.nix`, in the `fetchTarball` expression that imports `nixpkgs`. In current versions of `rix` it is a dated branch of the `rstats-on-nix/nixpkgs` fork, a URL ending in a date (`2024-04-29.tar.gz` for R 4.4.0), and from version 0.18.3 on it carries the archive's `sha256`. Older versions of `rix` omit the `sha256`, and older versions or hand-written declarations may instead name a commit of `NixOS/nixpkgs`. A full commit hash with a `sha256` is the strictest pin.

Draft, November 13, 2026. Content may change.

What would change if the revision were altered. Essentially everything below the R packages, and some of the packages too. The `nixpkgs` revision determines the entire dependency graph. That graph includes the R version itself, the C and Fortran compilers used to build it, the BLAS and LAPACK implementations, every system library, and the versions of the R packages available at that revision. Moving the revision moves all of them together.
This property is what makes Nix interesting for reproducibility. The revision is a single value that pins the whole stack from the system libraries upward. By contrast, a lockfile pins only the R packages, and a container pins a built artifact rather than a recipe for one. It is also the property that makes Nix unforgiving, since there is no way to move one component without moving the revision, and moving the revision moves everything.

Exercise 4

(a) A laboratory forbidden from licensing Docker Desktop. Axis one, a different runtime over the same image: Podman. The images, the digests, and the recipes are unchanged, so the constraint is met at the cost of changing one word in the launch command.
(b) A reviewer who must run a figure with no local installation. Axis three, remote execution: Binder, which builds the environment from the repository and serves it in a browser for the session. The reviewer installs nothing.
(c) A simulation study destined for a shared cluster. Axis one again, but a different tool: Apptainer, because a shared cluster will not grant the root daemon a standard Docker installation requires. The SIF also doubles as the archival record.
(d) A Windows-only pipeline mixing R and Python. Axis two, a different environment model: the conda family, and `pixi` in particular, which reproduces natively on Windows and spans both languages without a container.
No single tool answers all four, and each constraint selects an *axis* before it selects a tool.

Draft, November 13, 2026. Content may change.

Exercise 5

A serviceable reply:

> Nix pins the whole stack from the system libraries upward, and does it from a recipe rather than a built artifact, so it reaches L2 as cleanly as a container does and arguably more transparently. What it does not reach is L3, because pinning an environment is capture and L3 is verification: it needs evidence that the analysis was rebuilt from the pinned definition and regenerated its recorded outputs within a stated tolerance, which a baseline test establishes and Nix does not.

The correction to make is the level claim, not the tool choice. 'Fully reproducible' is the phrase that overstates, and it overstates in the direction this book cares about: it asserts verification that has not been performed. The colleague is right that Nix is a strong capture mechanism and right to prefer it if they value a declarative recipe; they are wrong that capture is the whole ladder.
Worth adding if the conversation continues: Nix pins more of the stack than a lockfile and does not pin the hardware or the intrinsic non-determinism, so even the capture claim has a ceiling.

Exercise 6

The simulation archetype most naturally uses Apptainer on a scheduler. Its work is many independent replicates, which suits a cluster, and a cluster will not run a root daemon. The SIF is built once from the digest-pinned image and dispatched to every node.
The blog archetype most naturally uses remote execution: Binder, if readers are to run the posts themselves. Its audience is on the web, and each post is small enough for a browser session. The render on every push is a separate matter, the continuous integration of Chapter 12, which is not one of this chapter's three axes.
Why the pinning is nonetheless identical. Both are built from the same compendium machinery: the same digest-pinned base image, the same dated snapshot, the same restored lockfile, the same determinism

Draft, November 13, 2026. Content may change.

variables. The SIF is a conversion of that image, not a different environment, and a Binder session reaches the same pin only when its `.binder/Dockerfile` starts from that digest. Neither archetype changes what is captured; they differ only in where the captured environment is executed and by what program.
The backends and runtimes are an axis of *delivery*, and the reproducibility level is set on an axis of *capture*. A simulation on a cluster and a blog in Binder can sit on exactly the same rung, because they are running the same pinned thing in different places.

Exercise 7

The solution for this exercise is in the instructor guide, which is distributed separately to course instructors. Self-study readers who want to check their answer can compare the file-existence results against those reported in the chapter's own 'The cohort, continued' section.

Exercise 8

(a)

```
apptainer build cohort-env.sif \
  docker://rocker/r-ver@sha256:<the digest from Chapter
↪  7>
```

(b) The SIF's checksum is computed over the bytes of the converted single-file archive, not over the Docker image's layers. Converting a pinned input into a different format therefore produces a different set of bytes, and so a different hash, even though the input digest was pinned. The SIF also embeds build metadata, so two conversions of the same digest need not hash alike either. What is pinned is the *input* to the conversion; the SIF's own checksum is evidence about the *output*, and the two need not, and do not, match.

Draft, November 13, 2026. Content may change.

Answers to the checks

Check 1. *from earlier chapters* (page 303)

The rebuilt image with the changed coefficient leaves two captured layers free to move. Package versions could move, because `install.packages()` against the live repository resolves to whatever is current on the day of each build, and no lockfile records the versions (Chapter 6). The operating system, the system libraries, and even the R patch level could move too, because a tag is a mutable label that a maintainer can rebuild and republish (Chapter 7). The rungs are cumulative (Chapter 3), so a Dockerfile without a lockfile is not L1, hence not L2, and the project stays at L0. The remedy is a base pinned by digest, a dated snapshot, and a lockfile restored inside the image. Even then, no rung guarantees the BLAS, so a stated tolerance is still needed.

Check 2. *the third judgment* (page 307)

The third: stating the level each option actually reaches. Diagnosing the constraint and locating Nix on the correct axis do not excuse the analyst from reporting the rung honestly. At best Nix reaches L2, the pinned environment, and only once the declaration has actually been run. 'Fully reproducible' overstates it because no rebuild has been verified, which is this chapter's own honest-level warning in miniature.

Check 3. *what changes with the runtime* (page 309)

None of them change. The runtime is a parameter over the same OCI image. The base image, its digest, and the package versions are all properties of the image, not of the program that runs it, and the level reached is L2 in either case. Only the launching command differs (`podman run` for `docker run`). This is why a runtime swap is the cheapest of the three axes: it changes the runner and nothing else.

Check 4. *peer, not supplement* (page 313)

L2, the pinned-environment rung, exactly as Docker reaches it. Nix is a peer rather than a supplement because the pin is achieved directly, by declaration, with no container format underneath it. There is no image for Nix to supplement, in the way `renv` supplements an otherwise unpinned project sitting inside a container.

Draft, November 13, 2026. Content may change.

Check 5. *what a Binder session pins* (page 315)

No; the environment lives entirely on Binder's hosted server, and nothing is pinned on the reviewer's machine. The session reaches L2 for as long as it runs only if the configuration is a `.binder/Dockerfile` built `FROM` the compendium's digest-pinned image. A configuration of `install.R` and `runtime.txt` files makes Binder resolve its own environment, which is not the compendium's pin. Even then the pin does not persist once the session ends, which is why Binder serves demonstration and review rather than archiving.

Check 6. *match the constraint to the axis* (page 318)

(a) Axis one, a different runtime over the same image: Apptainer, built from the digest-pinned image and run under the scheduler, with the SIF as the archival record. (b) Axis three, remote execution: Binder, which serves the container in a browser per session. (c) Axis two, a different environment model: the conda family, and `pixi` in particular, the one option that reproduces natively on Windows and spans both languages. Note that no single tool answers all three; each constraint selects a different axis.

Check 7. *the open door* (page 319)

Axis two, Nix through `rix`: it pins the whole environment directly, R, packages, and system libraries, from a declaration alone, with no Docker image to build first. Axis one is unavailable as long as the check shows no `Dockerfile`, because there is then no image for Podman or Apptainer to run. A devcontainer, axis three, reaches L2 for the cohort analysis only if it builds from this compendium's own pinned recipe; one that starts from a generic image, as the paragraph above reports when it does, does not.

Check 8. *what made three destinations cheap* (page 321)

All three destinations operate on, or around, the same pinned recipe, and none reinstalls it. Axis one swaps only the program that runs the image (Podman, Apptainer), and axis three moves where that same image runs (a hosted server). Neither axis asks what packages or base layer the image contains. Only switching to axis two, replacing the image with a Nix declaration, would have required redoing the pin itself, because then there is no image left to carry forward.

Draft, November 13, 2026. Content may change.

Check 9. *what the script does and does not pin* (page 323)

Without `--cleanenv`, Slurm's job environment would pass through into the container alongside the pinned image, exactly as it would in an interactive Apptainer session. That environment includes variables a cluster's login-node profile or a loaded module sets, such as a stray `R_LIBS_USER` or a locale setting. The flag clears those variables and nothing else. It does not isolate the host `$HOME`, which Apptainer still mounts, so a `~/.Rprofile`, a `~/.Renviron`, or a user library under `~/R` would still reach R; that takes `--no-home` (or `--containall`) and `Rscript --vanilla`, as the script uses. It also strips the `SLURM_*` variables the job may need, which is why the script passes them back with `--env`. None of these flags touches the image's own pinned contents, which were already fixed when the SIF was built from the digest. A job that fails only without `--cleanenv`, and succeeds with it, is failing because of something outside the image, not inside it.

8.14. Cheat sheet

Command or rule	What it does
`podman run ...`	Daemonless, rootless substitute for `docker run`; same OCI image, same digest, same L2.
`apptainer build x.sif docker://img@sha256:...`	Converts a digest-pinned OCI image into one archival SIF file.
`apptainer exec --cleanenv --no-home --bind "$PWD" x.sif Rscript --vanilla ...`	Runs the SIF with host environment variables cleared, the host `$HOME` unmounted, the project directory bound (Apptainer's system binds such as `/tmp` remain), and no R startup file read; `--cleanenv` alone clears variables only (see the Pitfall above).
`rix::rix(r_ver=, r_pkgs=, project_path=)`	Generates a `default.nix` pinning R, packages, and system libraries together.

Draft, November 13, 2026. Content may change.

Command or rule	What it does
`nix-shell default.nix`	Enters the Nix-declared environment; no image is built.
`.binder/Dockerfile`	Lets Binder serve the compendium's pinned image in a browser, per session, when its `FROM` line names the digest; `install.R` and `runtime.txt` make Binder resolve its own environment instead.
`.devcontainer/ devcontainer.json`	Lets an editor or Codespaces open the pinned environment directly.
Rule of thumb	Diagnose the binding constraint first; it selects the axis, and usually the tool.
Rule of thumb	A runtime swap (axis one) changes the runner only; the image, digest, and packages carry over unchanged.
Habit 2	Pin what you ran: the `nixpkgs` revision and the image digest are the same idea on two axes.
Habit 7	A `default.nix` or a SIF is a capture claim only, until Chapter 12's baseline test has run.

8.15. Further reading

The high-performance-computing runtime is described by its authors in Kurtzer et al. (2017), which sets out the unprivileged, scheduler-integrated design and the single-file image format that make Apptainer, formerly Singularity, the natural cluster companion to a Docker workflow.

The foundational account of the purely functional, input-addressed model is Dolstra et al. (2004). The R front-end that makes it approachable for the health scientist is documented in Rodrigues & Baumann (2024), whose `rix` package generates the Nix declaration from ordinary R code.

Draft, November 13, 2026. Content may change.

For the conda family as a middle-ground backend, we have two references: the reference documentation of the conda ecosystem (Anaconda Inc., 2023) and that of the lockfile-oriented `pixi` (prefix.dev, 2024). Together they describe the native-Windows and mixed-language reproduction that distinguishes it from the container route.

The remote-execution axis is introduced by Project Jupyter et al. (2018), whose Binder service runs a compendium's container in the browser, and by the development-containers specification (Development Containers Project, 2022), which brings the same environment into an editor. Both rest on the presence-driven principle, a configuration file's existence driving an automated build, that recurs throughout this book and is developed in Chapter 12.

The container backend against which all of these are alternatives is developed in Chapter 7, and the ladder of levels that lets us state honestly what each of them reaches is Chapter 3.

Draft, November 13, 2026. Content may change.

Part III.

Validation

Draft, November 13, 2026. Content may change.

9. Literate Programming and the Rendered Report

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

Donald E. Knuth, *Literate Programming* (1984)

This chapter has three aims. The first is to write an analysis whose every reported number and figure is produced by the code sitting beside it. The second is to recognize the handful of ways that guarantee silently breaks. The third is to say precisely what a successful render does and does not establish about an analysis's place on the reproducibility ladder.

Before you start

Reading time. About 50 minutes.
Prerequisites. Chapter 6 and Chapter 7, for the lockfile and the container. With these in hand, the seal this chapter develops is not mistaken for either kind of pin. Chapter 3, for the L0-L3 ladder and the capture-versus-validation vocabulary this chapter applies to the literate document itself. Statistically, nothing beyond reading a scatter plot and a reported percentage is required. R knowledge assumed: calling `library()`, writing a short function, and reading a basic `ggplot2` call; no prior exposure to `knitr` or Quarto chunk syntax is assumed.
Objectives. By the end of this chapter, can you:

- Explain Buckheit and Donoho's reproducibility argument, that a paper is advertising and the code and instructions are the scholarship, and say how a literate document makes the two the same artifact?

Draft, November 13, 2026. Content may change.

- Build a sealed render in Quarto or `knitr` and recognize each of the five landmines that silently break the seal?
- Choose between HTML and PDF output, name the toolchain each requires, and explain why a LaTeX distribution belongs in the image rather than on the host?
- Explain why two renders of an unchanged document do not hash identically, and say what verification compares instead?
- Place a sealed render on the L0-L3 ladder, and state precisely what the seal does and does not establish about an analysis's reproducibility?

Outline. We begin by locating a sealed render on the stack, and then trace the oldest idea in the book, from Knuth to Buckheit and Donoho. We next follow the R lineage from Sweave to Quarto. With that history in hand, we develop the sealed render, apply the same seal to inline numbers, and catalog the landmines that break the seal. Quarto's freeze and knitr's cache, which let a render skip the code, come next. We then consider HTML or PDF and the toolchain each needs, and explain why a render will not hash the same twice. The later sections treat literate programming across the archetypes, the write-as-you-go doctrine, a worked example, the cohort continued, and collaborating with an LLM. Lastly, the chapter closes with exercises, a cheat sheet, and further reading.

Check 1: from earlier chapters

Your bootstrap confidence interval is computed inside a digest-pinned image with a restored lockfile. A colleague runs the same image as a SIF under Apptainer on her cluster, with `--cleanenv`, the same seed, and the same data, and her interval differs. Her home directory holds an `.Rprofile` that calls `RNGkind(sample.kind = 'Rounding')`, and the project has none. Drawing on Chapter 3, Chapter 7, and Chapter 8, which determinant layer failed, why did the image not prevent it, and what closes it?
The answer is on page 373.

Draft, November 13, 2026. Content may change.

9.1. Where this sits on the stack

A sealed render is an unusual case among this book's chapters, in that it pins nothing new. Figure 9.1 marks analysis code as validated here rather than captured, because version control already secures the code at L0. What a sealed render adds is the check that the document's prose and figures are actually produced by that code, not pasted in and left to drift.

```
plot_determinant_stack(
  validates = c('Analysis code'),
  title = 'A sealed render validates that prose and code
   ↪  agree; it captures nothing new')
```

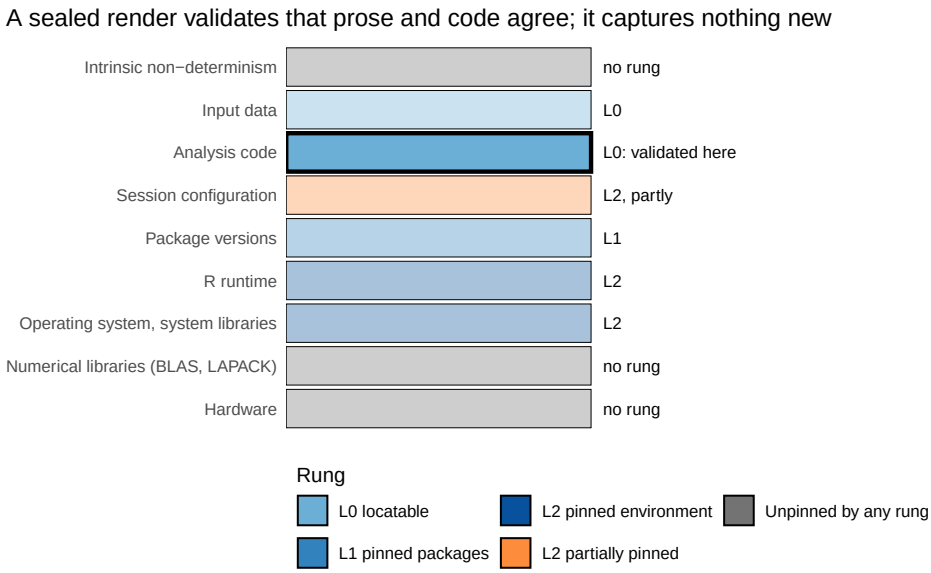


Figure 9.1.: The determinant stack of Chapter 3, with analysis code marked as validated rather than captured. A sealed render pins no new layer of the stack; it confirms that the numbers and figures a reader sees are the ones the already-committed code actually produces, which is the capture-versus-validation distinction this chapter applies to the rendered document itself.

Draft, November 13, 2026. Content may change.

Check 2: capture or validation, for a render

A sealed render regenerates every figure and number in a document from its own code on every build. Does sealing the render raise a project from L0 to L1, and if not, what does it do instead?
The answer is on page 373.

9.2. Orientation

Consider a familiar scene from a clinical-trial analysis. The statistician has a results table in the manuscript, produced months ago. She also has a script that, re-run today, produces a subtly different table: a mean that has moved in the third decimal, a confidence interval that no longer matches the sentence that interprets it. Neither the prose nor the code is obviously wrong. What has failed is the correspondence between them. The number in the paragraph was copied, by hand, from an analysis that has since been edited, and the paragraph was never updated. The document and its analysis have drifted apart, silently, and no one noticed until a reviewer asked how the interval was computed.

This chapter is about closing that particular gap, the gap between the prose a reader sees and the code that produced the numbers in it. The instrument is literate programming, by which we mean the practice of writing the narrative and the computation in a single source document. The figures and numbers a reader sees are then not pasted in but regenerated every time the document is built. We shall call a document with this property a sealed render: prose bound to the code that produces every number and figure it reports, so that the document cannot silently disagree with its own analysis.

The reader will recall the ladder of reproducibility levels that organizes this book: L0 locatable, L1 pinned packages, L2 pinned environment, L3 verified. But where on that ladder does a sealed render sit? The answer tends to surprise newcomers. A sealed render is, on its own, a property almost orthogonal to the ladder. It guarantees that the document agrees with the code, but it says nothing about whether the packages are pinned, whether the environment is fixed, or whether the outputs have been verified.

Draft, November 13, 2026. Content may change.

One can have a beautifully sealed document that runs only on its author's laptop, at L0. One can equally have a fully pinned L2 environment whose report pastes its numbers in by hand. The seal and the pins are different virtues, and much of the discipline of this chapter lies in not confusing them.

That said, the sealed render is the natural home for the machinery of the earlier chapters. It is where the pinned packages of Chapter 6 and the pinned environment of Chapter 7 are actually exercised, because it is the thing that runs the code. It is also the artifact that continuous integration renders on every change, as we shall see in Chapter 12. The sealed render is therefore the hinge between capturing an environment and validating that the capture suffices.

9.3. The analyst's contribution

Three judgments in this chapter are the analyst's own, and no tool makes them.

1. **Deciding what the document computes at build time and what it merely loads.** A figure regenerated on every render cannot drift from the text, which argues for computing everything in the document. A model that takes an hour to fit cannot be re-fitted on every render, which argues for computing it once in a script and caching the result. Where to draw this line is a design decision the analyst makes for each project. The line falls between the code that lives in the sealed document and the code that lives in `analysis/scripts/` with its output saved to `derived_data/`. The choice trades the strength of the seal against the cost of the render.

2. **Keeping the render sealed by refusing to reach outside it.** A sealed render must draw only on what is captured and committed. The temptations to reach outside are many and quiet, and we catalog them below as landmines. They are installing a package at render time, loading with `pacman::p_load`, reading a file by its absolute path on the author's machine, depending on state left in the session by a previous chunk, reading a file that was never committed. Each produces a document that renders perfectly for its author and fails

Draft, November 13, 2026. Content may change.

for everyone else. No linter catches all of them; the discipline is the analyst's.

3. **Matching the claim to what the render actually seals.** A sealed render seals prose to code. It does not, by itself, seal code to its dependencies, nor dependencies to an environment, nor establish that the outputs are correct. The analyst who says 'the analysis is reproducible' because the report renders has claimed more than the seal delivers. The honest claim names the level actually reached, and we return to this below.

Check 3: the analyst's third judgment

A document renders without error, and every figure in it is produced by the code beside it. Why is 'the analysis is reproducible' still not a claim the analyst is entitled to make on that basis alone?
The answer is on page 373.

The vocabulary of this chapter

- **Literate programming.** Writing narrative prose and executable code together in one source document, so the document is at once the program and its explanation. The term is Knuth's.
- **Sealed render.** A document in which every reported number and figure is regenerated from the code that produces it on each build, so the prose cannot drift from the analysis.
- **Code chunk.** A block of executable code embedded in a literate document, delimited from the prose, whose output is inserted into the rendered document at the chunk's position.
- **Render (verb and noun).** To execute a literate source document and produce a finished output file (HTML, PDF, Word); and the resulting file itself.
- **Rendering toolchain.** The programs a render invokes behind the scenes: `knitr` or the Quarto engine to run the code, `pandoc` to convert the result, and, for PDF, a LaTeX distribution to typeset it.
- **Determinism (of a render).** The property that rendering the same source under the same environment produces a byte-

Draft, November 13, 2026. Content may change.

for-byte identical output file. A rendered PDF is usually not deterministic in this strict sense, for reasons developed below.
- **`SOURCE_DATE_EPOCH`.** A conventional environment variable, honored by many document tools, that fixes the timestamp a build embeds, so that two builds do not differ merely because they ran at different moments.
- **Landmine.** A pattern in a document that makes the render succeed for its author and fail for a later reader, by reaching outside the captured and committed material.

9.4. The oldest idea in the book

The idea is Knuth's, and it is nearly as old as the practice of scientific computing itself. In 1984 Knuth proposed that a program should be written as a work of literature, a connected narrative addressed to a human reader, from which the machine-executable code is extracted mechanically (Knuth, 1984). He called the practice literate programming, and inverted the usual priority: the explanation is primary and the code is embedded within it, rather than the code being primary and the comments an afterthought. Knuth's own motive was the comprehensibility of programs. The application to reproducible research came later, and from a different quarter.

That application, and the sentence this book takes as its keystone, is due to Buckheit and Donoho, writing in 1995 about their WaveLab software and paraphrasing their colleague Jon Claerbout (Buckheit & Donoho, 1995). Their argument, in its clearest form, is this. An article about computational science is not the scholarship itself; it is merely advertising of the scholarship. The actual scholarship is the complete software environment and the complete set of instructions that generated the figures. On this view the printed figures and tables are not the result but a picture of the result, and a picture can be a faithful likeness or a flattering one. The result itself, the thing that can be checked and built upon, is the code and the instructions that produce the figures. A paper that advertises a result it cannot regenerate has, in Claerbout's sense, advertised scholarship it does not possess.

Draft, November 13, 2026. Content may change.

Literate programming is the technique that closes the distance between the advertisement and the scholarship. If the figure in the paper is generated, at build time, by code that lives in the same document as the prose around it, then the advertisement and the scholarship are the same artifact. The figure cannot be a flattering likeness of a result the code does not produce, because the figure is the output of the code. This is the whole of the reproducibility argument for literate programming, and every practical detail in the rest of this chapter is in service of it.

9.5. From Sweave to Quarto

The R ecosystem has realized Knuth's idea four times over, each realization improving on the last. The reader will meet all four in existing code, so we review the lineage briefly.

The first was Sweave, introduced by Friedrich Leisch around 2002 and shipped with R itself. Sweave let an author interleave R code with LaTeX prose in a single `.Rnw` file, ran the code, and wove the results into a LaTeX document. It proved the concept for R and served for a decade, but it was tied to LaTeX, awkward to extend, and offered little control over how code and output were formatted.

The second, and the decisive advance, was `knitr`, written by Yihui Xie and released in 2012 (Xie, 2015). `knitr` generalized Sweave in two directions at once. It decoupled the computation from the output format, so that the same code chunks could be woven into LaTeX, into HTML, or into Markdown. And it gave the author fine control over each chunk through chunk options, the `echo`, `eval`, `warning`, and `fig.cap` settings that recur throughout this book. `knitr` is still the engine underneath most R literate documents, including the executable chunk later in this chapter.

The third was R Markdown, which paired `knitr` with `pandoc`, the universal document converter (Xie et al., 2018). The author now wrote in Markdown, a light and readable prose syntax, rather than in LaTeX. `pandoc` then converted the woven Markdown into whatever final format was wanted: HTML for the web, PDF by way of LaTeX, Word for a collaborator who redlines in track changes. This is the `report.Rmd` that the zzcollab

Draft, November 13, 2026. Content may change.

templates scaffold, and the format in which a great deal of current health-sciences analysis is written.

The fourth and most recent is Quarto, released by Posit in 2022 and documented by Allaire and colleagues (Allaire et al., 2024). Quarto is best understood not as a replacement for R Markdown but as its generalization beyond R. It keeps the `knitr` engine as one option and adds a Jupyter engine for Python and Julia. It also unifies the chunk-option syntax across languages into the `#|` comment form used in this book, and it standardizes the production of books, websites, and journal articles from the same source. This very book is a Quarto book. For an R analyst the migration from R Markdown to Quarto is small, and the two will coexist for years. All four tools implement the same idea, and any of them can secure the seal.

Check 4: four tools, one idea

Sweave, `knitr`, R Markdown, and Quarto differ in the languages and output formats they support. What single property do all four share that makes each of them, and not merely the latest one, a literate-programming tool in Knuth's sense?
The answer is on page 374.

Check 5: advertisement and scholarship

Buckheit and Donoho call the published article 'advertising'. In one or two sentences, say what they identify as the scholarship itself, and explain how a literate document makes the advertisement and the scholarship the same object.
The answer is on page 374.

9.6. The sealed render

We begin with a figure. The following chunk generates a small synthetic dataset in the spirit of the Palmer penguins, three species measured on two bill dimensions. It fits a within-species and a pooled relationship, and draws the result. The point of the exercise is not the statistics, which are a textbook instance of Simpson's paradox, but the mechanism: this figure

Draft, November 13, 2026. Content may change.

is produced when the book is built, from the code the reader sees, and it cannot therefore disagree with the sentences around it.

```
library(dplyr)
library(tibble)
library(ggplot2)

set.seed(2024)

make_species <- function(name, mean_length, base_depth,
                         n = 60) {
  bill_length <- rnorm(n, mean = mean_length, sd = 2.2)
  bill_depth <- base_depth +
    0.15 * (bill_length - mean_length) +
    rnorm(n, sd = 0.6)
  tibble(species = name,
         bill_length = bill_length,
         bill_depth = bill_depth)
}

penguin_like <- bind_rows(
  make_species('Species A', mean_length = 39,
               base_depth = 18.5),
  make_species('Species B', mean_length = 49,
               base_depth = 18.5),
  make_species('Species C', mean_length = 47,
               base_depth = 15.0)
)

ggplot(penguin_like,
       aes(bill_length, bill_depth)) +
  geom_point(aes(colour = species), alpha = 0.7) +
  geom_smooth(aes(colour = species), method = 'lm',
              se = FALSE) +
  geom_smooth(method = 'lm', se = FALSE,
              colour = 'black', linetype = 'dashed') +
  labs(x = 'Bill length (mm)',
       y = 'Bill depth (mm)',
       colour = 'Species') +
```


Draft, November 13, 2026. Content may change.

```
  theme_minimal()
```

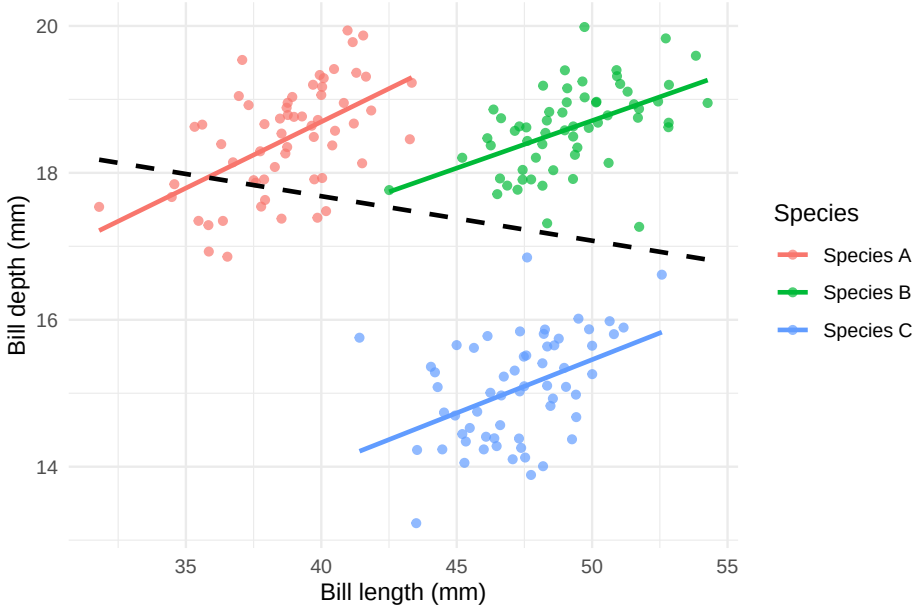


Figure 9.2.: The reversal of an association upon pooling. Within each synthetic species bill depth rises with bill length (solid lines); across the pooled sample the fitted line falls (dashed). Because the figure is computed when the book is built, the described reversal cannot drift from the code that produces it.

Within each of the three synthetic species, bill depth rises with bill length: the three solid lines slope upward. Pooled across the species, the relationship reverses sign. The dashed line comes out nearly flat with a slight downward tilt rather than a steep descent, because across species longer bills are associated on average with shallower ones. The reversal here is real but modest: it is the sign of the pooled slope that flips, not its magnitude that grows. This is Simpson's paradox, and it is a useful thing for a figure to show. What matters more for this chapter is that the preceding paragraph is safe to write. If a later edit changed the synthetic means so that the pooled line no longer reversed, the figure would change on the next render.

Draft, November 13, 2026. Content may change.

The paragraph would then become visibly false, prompting its correction. In a document that pasted the figure in as a static image, the paragraph could go on asserting a reversal the data no longer showed, indefinitely. The seal is precisely this coupling: the prose and the picture rise and fall together because they are the same computation.

Two properties of the chunk keep it sealed. First, it draws its data from `set.seed()` and `rnorm()`, not from a file on the author's disk, so it carries its inputs with it. Second, it loads only packages that the build environment is known to provide, and installs nothing. A chunk that violated either property might still render on the author's machine and would fail on anyone else's. The next section is a catalog of exactly these violations.

9.6.1. The seal applies to numbers in sentences, not only to figures

A figure is the conspicuous case, and the inconspicuous one does more damage. Numbers reported in prose are where drift is hardest to see, because a wrong number in a sentence looks exactly like a right one.

```
d <- survival::diabetic
laser <- d[d$trt == 1, ]
n_eyes <- nrow(laser)
n_events <- sum(laser$status)
pct <- round(100 * n_events / n_eyes, 1)
```

Having computed those, the sentence that reports them can be written with the values substituted rather than typed. In the laser-treated arm of the retinopathy trial there are 197 eyes, of which 54 progressed to visual loss, a rate of 27.4 percent.

If we read the source of that sentence, we find no numerals in it. Each is an inline expression evaluated at build time, so if the trial data were corrected, or the arm redefined, the sentence would change with it. This is the same seal the figure has, applied to the unit of text where a reader is least able to detect a lie. It is also the cheapest reproducibility discipline in this book: it costs a few characters per number, and it removes an entire class of error that no test, no lockfile, and no container can reach.

Draft, November 13, 2026. Content may change.

The habit generalizes. Any number a reader might check against the analysis, a sample size, a count of exclusions, a rate, a p-value, belongs in an inline expression. Numbers that are genuinely constants of the study design, an enrollment target or a protocol threshold, are properly typed, because they are inputs rather than results.

Verification

To see the seal hold on an inline number, change one line upstream of `n_eyes`, `n_events`, or `pct` in the `seal-inline` chunk above, say by subsetting a different treatment arm, and rerender. The sentence that reports them updates to match, with no edit to the prose itself. To make it fail on purpose, replace one of the inline expressions with the literal number it currently evaluates to, change the upstream data, and rerender. The hardcoded sentence will now silently disagree with the chunk that computed it, which is the drift this chapter exists to prevent.

Check 6: which numbers are sealed

In the retinopathy example above, `n_eyes`, `n_events`, and `pct` are each written as inline expressions. Suppose the study's protocol had stated an enrollment target of 400 eyes; the figure is hypothetical, not taken from the trial. Why would it be wrong to write that target as an inline expression computed from the data?
The answer is on page 374.

9.7. The landmines that break the seal

A sealed render is fragile in a specific way: it holds only if everything the render touches is captured and committed. Several common patterns reach outside that boundary, and each produces the same failure signature, a document that renders flawlessly for its author and errors, or worse renders differently, for a later reader. The zzcollab publishing playbook calls these landmines, and the name is apt, because they lie undetonated in a document that works until the moment someone else steps on it. We take the principal ones in turn.

The first is **installing packages at render time**. A chunk that calls `install.packages('somepkg')` or `pak::pak('somepkg')` during the render reaches out to a package repository as it builds. This makes the render depend on network access, on the repository's state on the day of the render, and on whatever version is current then. The remedy is that provisioning belongs in the environment, the Dockerfile and the lockfile of the earlier chapters, never on the render path. A sealed render installs nothing; it assumes its packages are already present, pinned, and captured.

The second, a specific and common case of the first, is **loading with `pacman::p_load`**. The `pacman` package offers `p_load()`, which installs a package if it is missing and then loads it, in one call. It is convenient in interactive exploration and a liability in a sealed render. On any machine where a named package is absent, it installs whatever version the repository serves that day, with all the fragility just described, and the render then succeeds against a package set no one chose. It also makes `pacman` itself a silent dependency of the render. On a machine without `pacman`, the call stops at once with 'there is no package called 'pacman' '. A common belief is that dependency scanners cannot see the packages `p_load()` names, but `renv` can. In a test with `renv` 1.2.3, `renv::dependencies()` reported the packages passed to `p_load()` as well as `pacman` itself, and the solution to Exercise 8 repeats the scan at every build. A scanner built on a narrower search may not, so whether a given scanner sees them is a question to test, not to assume. The remedy is to replace `pacman::p_load(a, b, c)` with explicit `library(a)`, `library(b)`, `library(c)` calls. These name the dependencies in the form every scanner recognizes, and they load without installing.

The third is **absolute host paths**. A chunk that reads `/Users/jsmith/data/cohort.csv`, or sources a LaTeX preamble from `~/shr/preamble.tex`, embeds a fact about one particular machine into the document. The path exists on the author's laptop and nowhere else, so the render fails the moment the document moves into a container, onto a colleague's machine, or into continuous integration. The remedy is that every file a render touches must live inside the compendium and be referenced by a relative path, so that the document carries its inputs with it.

Draft, November 13, 2026. Content may change.

The fourth is **hidden state**. A document renders its chunks in order, in a single R session. In that setting it is easy to write a chunk that depends on an object created in an earlier chunk, or on an option set earlier, or on a package attached earlier, without saying so. The document renders correctly from top to bottom and fails when a reader runs the chunk in isolation, or when a chunk is reordered, or when caching skips the chunk that created the state. A related anti-pattern is a chunk that calls `rm(list = ls())` to force a clean slate. It does not actually deliver one, since attached packages and changed options survive it, and it can delete objects the rendering harness itself relies on. The remedy is a discipline of self-contained chunks and, at the level of the toolchain, rendering each document in a fresh session so that no state leaks in from the author's console.

The fifth is **reading uncommitted files**. A chunk may read a file that exists in the author's working directory but was never committed to version control, a scratch CSV, a manually downloaded spreadsheet, an intermediate the author produced by hand. The render succeeds for the author, whose working tree contains the file, and fails for anyone who clones the repository, whose working tree does not. This is the subtlest landmine, because version control hides it: the document and the file both sit in the author's directory, and only a clean checkout reveals that one of them was never tracked. The remedy is that a sealed render reads only committed inputs, and the surest test is to render from a fresh clone.

> ⚠ Pitfall
>
> Every landmine in this section shares one signature: the render succeeds on the author's machine. Local success is therefore no evidence that a render is sealed. The only convincing test is a render in a clean environment that carries none of the author's accumulated state, which is exactly what a container, or continuous integration, provides. We develop that test in Chapter 12; for now, the reader should treat 'it renders for me' as the beginning of the question, not the end.

Check 7: diagnosing a landmine

A colleague's `report.Rmd` renders on her laptop but fails in continuous integration with the message 'cannot open file: data/interim.csv'.

Draft, November 13, 2026. Content may change.

> Name two distinct landmines that could produce this symptom, and give the one-line check that would distinguish them.
> *The answer is on page 374.*

9.8. Freeze and cache: renders that skip the code

Two features let a render skip work, and both weaken the seal without saying so on the page. Quarto's `freeze` option acts on whole documents during a project render. With `freeze: true` a document is never re-executed by a project render. With `freeze: auto` it is re-executed only when its source file changes. Otherwise Quarto reuses the results stored in the project's `_freeze/` directory, and its documentation advises committing that directory, so that others can render the project without the environment that computed it (Posit Software, PBC, n.d.). Freeze applies only to a render of the whole project; rendering a single document always executes its code. The `knitr` chunk cache acts on single chunks. It saves the objects a chunk creates, its printed results, and the random seed, and reloads them on the next render. A change to the chunk's code or options invalidates it (Posit Software, PBC, n.d.; Xie, 2011).

Neither feature looks at the rest of the stack. Under `freeze: auto` the comparison is with the document's own source. A corrected data file, an updated package, a new R release, or an edited function under `R/` therefore leaves the frozen results in place. The chunk cache behaves the same way unless the analyst ties it to those inputs, for example by setting the `cache.extra` option to the R version string or to a data file's modification time (Xie, 2011). A cached chunk also skips its side effects, which is how caching springs the hidden-state landmine above. In each case the page carries a fresh build date over numbers computed earlier, perhaps on another machine.

This book is a case in point. Its `_quarto.yml` sets `freeze: auto`. When this page last executed, the repository committed frozen results for 15 documents (52 files under `_freeze/`). Its continuous-integration workflow runs a project render, so freeze applies there too. A chapter whose source has not changed since its last local execution is published with the results computed on the machine that executed it and committed `_freeze/`, not

Draft, November 13, 2026. Content may change.

with results computed in the CI environment. The counts in this paragraph are subject to the same rule: they are as of this page's last execution. For a book whose chunks illustrate rather than report, that trade may be acceptable. For a deposited analysis it is not. The render that produces a submitted or deposited document should run in the pinned environment with freeze off, or after deleting `_freeze/` and the cache directories. Freeze and cache are conveniences for drafting, not evidence that the numbers are current.

9.9. HTML or PDF, and the toolchain each needs

A literate source document is format-agnostic; the same `report.Rmd` can render to HTML, to PDF, or to Word. The choice of format is not merely cosmetic, however, because each format invokes a different toolchain behind the render, and the environment must carry whatever the chosen format needs.

HTML is the lighter of the two. Rendering to HTML runs the code with `knitr` and converts the result with `pandoc`, and both are modest, self-contained dependencies. HTML is the natural default for a working analysis, for a blog post, and for anything read on a screen; it supports interactive figures, it degrades gracefully, and it imposes the smallest environmental burden. For most of a project's life, and for every archetype whose output is read rather than submitted, HTML is the sensible target.

PDF is heavier, because a PDF is typeset, and typesetting in this ecosystem means LaTeX. Rendering to PDF runs the same `knitr` and `pandoc` steps and then hands the result to a LaTeX distribution. The distribution lays out the pages, resolves the cross-references, sets the mathematics, and embeds the fonts. A LaTeX distribution is large: a full TeX Live installation occupies several gigabytes. Its absence is the single most common cause of a PDF render failing on a machine where the HTML render succeeds. This is precisely the environmental problem the earlier chapters solved, and it is why the framework's publishing profile exists.

The publishing profile builds on the `rocker/verse` base image, which carries a base LaTeX distribution sufficient for a straightforward render: plain prose, ordinary tables, standard mathematics. It does not carry a full

Draft, November 13, 2026. Content may change.

LaTeX distribution. A document that reaches for a package the base omits (dense tables, `moderncv`-style layouts, and similar) still needs that package installed with `tlmgr` before it will render. zzcollab's own publishing profile pre-installs roughly two dozen such packages at image build time precisely because the base image does not ship them. A project that renders to PDF chooses the publishing profile at initialization. A project that renders only to HTML does not need it, and pays neither the build time nor the image size that the LaTeX toolchain costs. The playbook that migrates a compendium to the publishing profile turns on exactly this question. It scans every report's YAML for a `pdf_document` or `latex_engine` output and, finding one, concludes that the verse base is required.

As elsewhere in the book, the literate document declares what it wants to be, and the environment must be provisioned to supply it. HTML asks little; PDF asks for LaTeX. The honest way to supply LaTeX is to pin it in the image, not to install it, fragilely and at render time, on whatever machine happens to build the document.

> Check 8: choosing the profile
>
> A project renders only to HTML today, but a collaborator wants to add a `pdf_document2` output to one report next month. What, if anything, should change about the project's Docker image now, and why might waiting be the wrong call?
> *The answer is on page 375.*

9.10. Why a rendered document will not hash the same twice

The next point surprises many analysts, and it shapes how verification is done in Chapter 12. It is natural to suppose that the way to check a reproducible report is to render it twice and compare the two output files: if the analysis is reproducible, surely the PDFs should be identical? They will not be, and the reason is instructive.

A rendered PDF, and to a lesser degree a rendered HTML file, embeds metadata that has nothing to do with the analysis and everything to do

Draft, November 13, 2026. Content may change.

with the moment of the render. A PDF carries a creation timestamp and a modification timestamp in its metadata. It also carries an internal document identifier, which engines derive from the time, the file name, and in some cases the build path. Some engines also give the font subsets they embed names that can vary between builds. Two renders of byte-for-byte identical source, in the same environment, minutes apart, therefore produce two PDF files that differ in their bytes. They differ even though every number and figure in them is identical.

Part of this non-determinism can be normalized away. Many document tools, including LaTeX and `pandoc`, honor a conventional environment variable, `SOURCE_DATE_EPOCH`, which fixes the timestamp the build embeds to a specified value instead of the wall-clock time. In a test with TeX Live 2026, setting it alone made two pdfLaTeX, XeLaTeX, or LuaLaTeX builds of a short document in one directory identical. Builds in different directories could still differ in their identifier, and a full report adds figures, paths, and tool versions, each another source of difference. Normalizing them all is worthwhile work in some settings, notably in the reproducible-builds movement for operating systems. For a research compendium it is usually misplaced effort, because it fights a symptom.

The resolution the framework adopts is to compare the right thing. Verification, developed in Chapter 12, does not hash the rendered document. It recomputes the computational outputs, the derived datasets, the model estimates, the figure data, the numbers themselves, and compares them with a recorded baseline within a stated tolerance (habit 5). The tolerance matters because even these numbers can move in their last digits under a different numerical library without the analysis changing. A hash detects any change at all, so it is the right instrument only for a file whose bytes should not change, such as a raw data extract; Chapter 12 develops both. The rendered report is a human-facing artifact, a faithful presentation of those outputs. It is checked by regenerating it and confirming that the outputs it presents match the recorded baseline within that tolerance, not by demanding that its bytes be identical. For provenance the framework goes further in the opposite direction. Its render hook deliberately stamps each rendered PDF with the render time and the source path, in a footer, and stages a dated copy whose name and manifest row record the git version of the source. A reader can therefore tell two renders apart and trace each to its source commit. The rendered document is thus a dated, signed

Draft, November 13, 2026. Content may change.

presentation of a result; the object of record is the computation.

> **!** The honest level
>
> This chapter's own executed chunks, `fig-simpson`, the inline-sealed retinopathy sentence, and the cohort passage below, are sealed. Every number in them regenerates from the code beside it. The seal adds no rung, however. These chunks sit on whatever rung the compendium's lockfile and environment reach, which Chapter 6 and Chapter 7 compute from the repository; nothing in this chapter raises it.

> Check 9: two PDFs, one analysis
>
> Two renders of the same, unchanged `report.Rmd` produce PDF files with different checksums. Does this mean the analysis is not reproducible?
> *The answer is on page 375.*

9.11. Literate programming across the archetypes

The weight the sealed render carries varies sharply across the five archetypes; Table 5.2 in Chapter 5 is the reference. It is the center of gravity for the manuscript, multiplied across many posts for the blog, and lighter for an internal data-analysis summary. The two cases most worth stating here are the ones a reader fixed on the manuscript will misjudge. The **R-package** archetype has no sealed report at all. Its 'document' is its help pages and vignettes, generated from roxygen comments and demonstrating the package rather than reporting an analysis. Its correctness rests on the tests of Chapter 11, not on a rendered result. The **simulation-study** archetype inverts the usual weight for a different reason. The heavy computation, hours or days of Monte Carlo replicates, cannot live in the sealed document. It therefore runs once in `analysis/scripts/`, seeded with the care Chapter 3 stressed, and saves its results to `derived_data/`. The literate report then becomes a thin presentation layer that loads and presents those results. In every case the underlying machinery (`knitr` or Quarto, chunks, committed inputs, a pinned environment) is the same; only how heavily an archetype

Draft, November 13, 2026. Content may change.

leans on it changes.

> Check 10: the package that renders nothing
>
> An R package maintainer is told, by a colleague who has read only the manuscript archetype's section above, that the package needs a `report.Rmd` to be reproducible. Is the colleague right?
> *The answer is on page 375.*

9.12. The write-as-you-go doctrine

One matter of practice remains, and experienced compendium authors are most insistent about it. The traditional order of work runs: write the analysis scripts, generate the results, then start the manuscript, and finally copy the results into the prose. This order builds the drift we opened the chapter with directly into the workflow, because the copying is manual and the results and the prose are separate artifacts from the start.

The write-as-you-go doctrine inverts the order. We create the `report.Rmd` on the first day of the project, before there is any analysis to report. Into it we write the research question, the background, and the planned methods as prose. The analysis code is then added inline, in chunks, as the work proceeds, so that from the first day the numbers in the document are computed rather than copied. As patterns emerge, code that is used more than once is extracted from the document into a function in `R/`, and as those functions stabilize they acquire tests in `inst/tinytest/`, the subject of Chapter 11. When a computation grows too slow to run on every render, it moves to `analysis/scripts/`, is run once, and saves its result for the document to load. This is the build-time-versus-load-time boundary named among the analyst's contributions above. The compendium's directory structure, in this view, emerges from the writing rather than preceding it. That is, the manuscript drives the analysis, and the `R/`, `tests/`, and `scripts/` directories fill in as the code in the document matures.

The doctrine's reproducibility payoff is that the document is sealed from the first day and stays sealed. There is never a phase in which results live in the prose by having been copied there. The number in the paragraph was computed by the chunk above it on the very first render, and on every

Draft, November 13, 2026. Content may change.

render since. The drift that opened this chapter has no moment at which to enter.

> Check 11: why day one, not day thirty
>
> The write-as-you-go doctrine creates `report.Rmd` before any analysis exists. What specifically would be lost by instead writing the analysis scripts first for a month and only then starting the document, even if every number in the eventual document were still computed rather than typed?
> *The answer is on page 375.*

9.13. Worked example

We now assemble the pieces into a concrete `report.Rmd`, as a health scientist beginning a cohort analysis would. On the first day, before any data are analyzed, the author creates the report and writes its header. A representative YAML header, shown here as a static, unrendered block, declares the title, the authorship, the output formats, and the bibliography:

```
---
title: 'Bill morphology across three penguin species'
author:
  - name: A. Analyst
    affiliation: Department of Epidemiology
date: 'November 08, 2026'
output:
  bookdown::pdf_document2:
    number_sections: true
    keep_tex: true
  bookdown::html_document2:
    toc: true
bibliography: references.bib
---
```

Two features of this header repay attention. The `output` field names two formats, a PDF built through LaTeX for submission and an HTML version

Draft, November 13, 2026. Content may change.

for reading on a screen. The project that renders the former needs the publishing profile and its baked-in LaTeX distribution, while the latter needs neither. The `date` field is itself a small literate device: it holds an inline R expression that inserts the render date, so the header cannot claim a date the render did not have. The provenance footer discussed below is not declared in this header at all; it is applied by the render tooling (`make docker-render`, by way of `tools/stamp-render.R`).

Below the header the author writes a setup chunk that fixes the chunk defaults and loads the packages the document uses. Each package is named with an explicit `library()` call so that `renv` can discover them and no package is installed at render time:

```
knitr::opts_chunk$set(
  echo = FALSE,
  message = FALSE,
  warning = FALSE,
  fig.path = '../figures/'
)
library(dplyr)
library(ggplot2)
```

From here the write-as-you-go doctrine takes over. The author writes the introduction as prose and adds a data-summary chunk. Then come the modeling chunk that produces the analysis and the figure. The figure chunk is exactly the sealed `fig-simpson` chunk demonstrated earlier in this chapter, which computes its result at build time and carries its synthetic inputs with it. Each number the introduction and discussion cite is inserted by an inline expression that reads the computed object, never typed by hand. Rendering the document inside the container, with `make docker-render` or its equivalent, executes every chunk in a fresh session, regenerates every figure, and produces the PDF and HTML, each carrying the render's provenance footer.

The result is a document that is sealed by construction. To convince ourselves that the seal holds, and does not merely appear to hold on the author's machine, we render it from a fresh clone of the repository inside the pinned environment (habit 6). If it renders there, with no file missing and no package installed on the fly, the seal is real. If it fails, one of the

Draft, November 13, 2026. Content may change.

landmines cataloged above has been stepped on, and the failure names which.

> Check 12: what the header's `date` field buys
>
> The YAML header above sets the `date` field to an inline expression that evaluates `Sys.Date()` at render time, rather than a fixed string typed on the day the project began. What would be lost by typing the date literally instead, and is this an instance of the same seal as the figure and the inline numbers?
> *The answer is on page 376.*

9.14. The cohort, continued

The same discipline applies directly to this book's own running cohort. The chunk below loads and cleans the extract and computes two numbers the way every sealed document should, as objects a sentence can cite rather than as values copied from a console.

```
cohort <- clean_cohort(load_cohort(
  '../data/raw_data/cohort_raw.csv'))
n_cohort <- nrow(cohort)
pct_missing_outcome <- round(
  100 * mean(is.na(cohort$incident_dm)), 1)
```

The cleaned cohort holds 337 records, of which 9.5 percent carry no recorded outcome. Both numbers are inline expressions reading `cohort`, not literals typed after a console check. A later edit to `clean_cohort()`, or to the raw extract, therefore changes this paragraph on the next render instead of leaving it to drift the way the orientation's opening story describes. Nothing about this passage pins a package or an environment; it is the seal, and only the seal, demonstrated on the book's own data.

Draft, November 13, 2026. Content may change.

Check 13: what the cohort passage does and does not show

The chunk above reports that 9.5 percent of the cleaned cohort is missing an outcome. Does rerunning this chunk on a colleague's machine, and getting the identical number, tell that colleague anything about whether the number is correct?
The answer is on page 376.

9.15. Collaborating with an LLM

A language model is a capable assistant for literate documents, and a treacherous one, because its fluency in producing chunks that run on the author's machine is exactly the fluency that plants landmines. Three exchanges illustrate the pattern of use and the verification the analyst must supply.

! Restricted data

One rule comes first (habit 3). Never paste or upload protected health information, identifiable records, or data held under a data use agreement, nor outputs derived from them, into a model or a coding agent unless three conditions all hold. The institution has approved that endpoint for that class of data. The IRB protocol and the consent permit the processing. The data use agreement, or the provider's terms, permits it. If any of the three is unclear, the answer is no.
The third condition is the one most often missed. Institutional approval of a tool does not override the terms under which a data provider released the data. NIH, for example, treats sharing controlled-access data through prompts with public generative AI tools, which it glosses as third-party tools, as a breach of the data use certification. It treats a model trained on such data as a data derivative, to be shared only with other approved users and not kept after the project closes (National Institutes of Health, 2025a).
Describe the structure instead: the column names and types, or a small synthetic example generated from the data dictionary, never edited from real rows. A coding agent sees whatever files are in its

Draft, November 13, 2026. Content may change.

working tree, so run it in a checkout or container where the real data is not mounted.
An agent also acts, so confine it to a branch and keep merges, pushes, and deposits for yourself (Section 2.10.1).

A literate document mixes code with data and results, so it is easy to paste too much. The derived outputs the rule covers include a rendered table of small cells, and a coding agent asked to render the document reads the data it renders.

Prompt. 'Write a `knitr` chunk that loads `dplyr` and `survival` and reads the CSV at `analysis/data/raw_data/cohort_raw.csv`, which has columns `mrn` (character), `enroll_date` (character, in two date formats), `hba1c` and `bmi` (numeric), and `incident_dm` (integer, 0 or 1, with missing values).'

Watch for. Check for convenience chosen over the seal: a `pacman::p_load(...)` call, an `install.packages()` call, or a `library(pacman)` line. Check also that the data are read from the path given, not from an invented absolute path such as `/Users/you/data/cohort.csv` or from a path outside the project.

Verification. Insist on explicit `library()` calls, one per package, with no installation on the render path, and a relative path to a committed file under the compendium. Then apply the decisive test: render the chunk from a fresh clone inside the container. A chunk that survives that test is sealed; a chunk that only ran in the author's console is not, whatever the model asserted.

Prompt. 'My PDF render fails with a LaTeX error but the HTML render works. Fix it.'

Watch for. A model may propose to install a LaTeX package at render time, or to add a `tinytex::tlmgr_install()` call inside a chunk. Either converts an environment problem into a render-time landmine and makes the document depend on network access and repository state.

Verification. The correct locus of the fix is the environment, not the document. Confirm that the failure is a missing LaTeX toolchain, and provision it by building on the publishing profile so that LaTeX is baked

Draft, November 13, 2026. Content may change.

into the image, not installed as the document renders. Verify that the PDF then renders in the container with nothing installed on the fly.

A better prompt. 'My Quarto document renders to HTML, but the PDF render fails; the log is attached, with paths replaced by placeholders. The build runs in a container from the attached Dockerfile, and nothing may be installed during the render. Identify the missing LaTeX component from the log, propose the Dockerfile change that provides it, and name the command and the log line that will confirm the fix.'

Prompt. 'Find the hidden state in this document.'

Watch for. Check whether the answer rests on reading the source alone. Hidden state belongs to the session in which the chunks run. A reading can miss an object created in the console, an option set in `.Rprofile`, or a chunk that works only because a cached predecessor was skipped, and it can flag chunks that are in fact self-contained.

Verification. Test each item by execution. Delete `_freeze/` and any cache directory, so that the skipped chunks of Section 9.8 run too, and render the document in a fresh session inside the container. Then run each flagged chunk alone in a new R session. A chunk that fails alone depends on state that something earlier supplied; a chunk that passes does not, whatever the model reported.

A literate document carries the same disclosure obligation as the rest of the paper. A render's seal attests that the code ran and the prose agrees with it, not who or what drafted either. Record any model use in drafting a chunk, a caption, or a paragraph with the template and the `LLM-USE.md` record of Section 2.10.2. A model is also not a versioned dependency the way a package is: a provider can retire or update an identifier, and one unchanged version need not return the same text twice. The reproducibility conclusion follows. Text a model drafted is reproducible only as the committed file, never by asking again, and a model whose output feeds the analysis is a dependency in its own right (Section 10.12).

Draft, November 13, 2026. Content may change.

9.16. Exercises

1. Take an existing `.Rmd` or `.qmd` document of your own in which at least one reported number was typed into the prose by hand. Replace that number with an inline R expression that computes it from the data, render the document, and confirm the value is unchanged. Then edit the data slightly, re-render, and observe the number update in the prose. Describe what would have happened to the hand-typed number under the same edit.

2. Write a short `knitr` or Quarto chunk that generates synthetic data with `set.seed()` and produces a figure, as the `fig-simpson` chunk in this chapter does. Render it twice and confirm the figure is identical. Then remove the `set.seed()` call, render twice more, and explain in terms of this chapter why the figure now differs between renders and what that implies for the seal.

3. Deliberately plant each of the five landmines from this chapter, one at a time, in a copy of a working document. The five are an `install.packages()` call, a `pacman::p_load()` call, an absolute host path, a chunk that depends on hidden state from an earlier chunk, and a read of an uncommitted file. For each, predict whether the document will still render on your own machine, then test the prediction by rendering from a fresh clone in a clean environment.

4. Locate a document in your work that renders to PDF, and determine whether the LaTeX toolchain it requires is pinned in an environment or assumed present on the host. If it is assumed present, describe the steps by which you would move it into the publishing profile, and state which reproducibility level the move would raise the project to.

5. Render a PDF from a literate document twice, minutes apart, and compare the two files byte for byte (for instance with a checksum). Confirm that they differ, inspect the metadata to identify what differs, and explain why hashing the rendered document would be the wrong basis for verification. Then describe what should be compared instead, and how.

6. For each of the five archetypes named in this chapter, the manuscript, the blog, the data analysis, the R package, and the simulation study,

Draft, November 13, 2026. Content may change.

write one or two sentences stating what the literate document renders in that archetype, if anything, and how heavily the reproducibility weight rests on the render as opposed to elsewhere.

7. On the running cohort compendium, run `clean_cohort(load_cohort('analysis/data/raw_data/cohort_raw.csv'))` and write a one-sentence, inline-sealed report of the cleaned row count and the percentage of rows missing an outcome, in the style of 'The cohort, continued' above. Report the two specific numbers a grading script could check without rendering anything: `nrow(cohort)` and the percentage missing `incident_dm`, rounded to one decimal place.

8. This exercise returns to the dependency-pinning discipline of Chapter 6 (Chapter 6). The `fig-simpson` chunk in this chapter loads `dplyr`, `tibble`, and `ggplot2` with explicit `library()` calls. Run `renv::dependencies()` on this chapter's own `.qmd` file, as Chapter 6's own cohort exercise runs it on `R/cohort.R`, and report which of those three packages the scan finds referenced. Then write a scratch `.R` file whose only line is `pacman::p_load(dplyr, tibble, ggplot2)`, scan it the same way, and report what the scan finds. Explain what the result implies for the `pacman` landmine, and for any claim about a scanner's coverage that has not been tested.

9.17. Solutions

Attempt an exercise before expanding its solution. Most of this chapter's exercises are performed on a document of the reader's own, so their outputs vary; the boxes record what should happen and why.

Exercise 1

The substitution replaces a literal with an expression, so that a sentence reading 'The mean age was 61.4 years' becomes, in the source, `r round(mean(cohort$age_at_enroll, na.rm = TRUE), 1)` written inline where the number used to be.
On the first render the printed value is unchanged, which is the confirmation that the expression computes what the hand-typed number

Draft, November 13, 2026. Content may change.

claimed. After a small edit to the data, the prose updates on re-render.
What would have happened to the hand-typed number. Nothing at all. It would have sat in the paragraph, unchanged and now wrong, with no error raised anywhere and nothing in the document to indicate a problem. That silence is the whole difficulty: the failure produces no symptom, so it is found only when someone independently recomputes the number, which for most published analyses is never.
This is the drift the chapter opens with, in its smallest form. The inline expression does not make the number correct; it makes the number *bound*, so that it cannot disagree with the analysis without the disagreement being visible on the next render.

Exercise 2

With `set.seed()` present, the two renders produce identical figures, because the synthetic data is a deterministic function of the seed and the code.
With the call removed, the generator is initialized from varying session state, so each render draws a different sample and the figure differs between renders.
What the seal does and does not cover. The seal binds prose to code: it guarantees that the figure a reader sees was produced by the code shown beside it. That guarantee still holds without the seed. What fails is something adjacent and easily confused with it, which is that the document is no longer a deterministic function of its source. Two people rendering the same committed document get different figures. Any sentence in the prose describing the figure ('the association reverses within each group', 'three points lie outside the interval') can therefore become false without the code changing.
A seal makes the document agree with its code, and a seed makes the code agree with itself across runs. An unseeded literate document is sealed and not reproducible, which is a good illustration of the chapter's warning that the seal and the pins are different virtues.

Draft, November 13, 2026. Content may change.

Exercise 3

The predictions worth making before testing, and what actually happens.

`install.packages()`. Renders on your machine, and probably on the fresh clone too, since it installs what it needs. That is what makes it insidious: it does not fail, it silently installs whatever version is current, so the document renders against a different package set than the one it was written for. The landmine here is not a broken render but an unpinned one.

`pacman::p_load()`. Renders on your machine. On the fresh clone the outcome depends on whether `pacman` is installed there. If it is not, the render stops at once with 'there is no package called 'pacman' '. If it is, `p_load()` silently installs whatever versions are current, exactly as `install.packages()` does.

An absolute host path. Renders on your machine and fails immediately on the fresh clone, with a 'cannot open file' error. This is the friendliest landmine, because it announces itself.

A chunk depending on hidden state from an earlier chunk. Usually renders in both places, since a full render executes chunks in order in one session. It fails when someone runs the chunk in isolation, or when the chunk order changes, or when caching or `freeze` skips the chunk that established the state. Intermittent and hard to diagnose.

A read of an uncommitted file. Renders on your machine and fails on the fresh clone, the same symptom as the absolute path. This is exactly why the chapter pairs them and asks for a distinguishing check: inspect version control to see whether the file is tracked.

Two of the five fail loudly, two render anyway, and `pacman::p_load()` does either, depending on the machine. The ones that render anyway are the dangerous ones. Rendering from a fresh clone catches the loud two reliably and the quiet ones only sometimes, which is the argument for continuous integration in Chapter 12.

Exercise 4

No single correct answer, since it depends on the document. The determination is usually quick: if the project contains no Dockerfile

Draft, November 13, 2026. Content may change.

and no environment specification naming a TeX distribution, the toolchain is assumed present on the host, which is the common case.
The steps to move it in. Select the publishing profile, so the base image becomes `rocker/verse`, which carries the LaTeX toolchain. Pin that base by digest. Confirm that the document's specific LaTeX dependencies are present, since `verse` carries a substantial but not unlimited TeX installation. Add any missing packages explicitly to the Dockerfile rather than letting a run-time installer fetch them. Then render inside the container and confirm the PDF is produced.
Which level the move raises the project to. L2, from wherever it was before, provided a lockfile is also in place. The exercise invites an overclaim: moving the toolchain into the image is the L1-to-L2 step for the *rendering* dependency specifically. If the packages were never pinned, the project moves to L2 only once the lockfile is restored inside the image too, since L2 presupposes L1. And it does not reach L3, because no baseline test has yet shown that the render regenerates its recorded outputs within a stated tolerance.

Exercise 5

The two PDFs differ, and their checksums differ.
What differs. Inspecting the metadata, with `pdfinfo` or by examining the raw file, shows a `CreationDate` and a `ModDate` recording the moment of the render. Depending on the engine, the file may also embed a document identifier derived from the time, font subset tags that vary with the subsetting order, and absolute paths from the build directory. None of this is the analysis; all of it changes between renders.
Why hashing the rendered document is the wrong basis for verification. The check would fail on every rebuild, including rebuilds in which nothing whatever changed. A check that always fails carries no information, and its practical effect is worse than no check at all, because it trains whoever sees it to ignore the failure. A verification gate is useful in proportion to how reliably its failures mean something.
What should be compared instead, and how. The computational outputs that feed the report: the derived datasets, the fitted model's estimates, the results tables. A baseline test (Chapter 11) re-

Draft, November 13, 2026. Content may change.

computes them after a clean rebuild and compares them with recorded values within a stated tolerance (habit 5), so a failure means that a number moved by more than floating-point noise. A hash is the wrong instrument here, because an estimate's last binary digit can move under a different numerical library without the analysis changing, and a hash would report that as a failure. Keep hashes for files whose bytes should not change, such as a raw data extract. The render gate then sits alongside as a separate check confirming that the document still builds from those outputs.

Byte-identical PDFs are attainable, by setting `SOURCE_DATE_EPOCH` and holding the build path and every tool version fixed, but this is a fragile route to a result the baseline comparison gives directly. The chapter's position is to compare the computation, within a tolerance, rather than work to make the non-deterministic document deterministic.

Exercise 6

Manuscript. The literate document renders the paper itself, and the reproducibility weight rests almost entirely on the render: the deliverable *is* the rendered artifact, and every number in it must be computed rather than typed. This is the archetype the machinery was designed for.

Blog. Renders each post. The weight is substantial but lighter than the manuscript's, because the stakes of a drifted number are lower and posts are usually self-contained. The distinctive pressure is longevity: posts are written once and read for years, so the pinned packages matter more than the render does.

Data analysis. Renders a report, if there is one, and the weight is moderate. Much of the analysis lives in scripts producing derived data, and the report summarizes rather than performs. The seal matters for the summary; the reproducibility of the analysis rests mainly on the scripts and the pinned environment.

R package. Renders nothing, or at most a vignette. The weight rests almost entirely elsewhere, on `R CMD check` and the unit tests, which bind behavior to claims more strongly than prose can. A vignette that renders is a convenience and a demonstration, not the deliverable.

Draft, November 13, 2026. Content may change.

Simulation study. Renders a report on the results, if one exists. The weight rests overwhelmingly on the seed and the generator rather than on the render: a sealed document reporting an unseeded simulation is bound to its code and still not reproducible, as exercise 2 shows.
The pattern across the five is that the render carries the most weight exactly where the document is the deliverable. In the two archetypes where it carries least, the package and the simulation, a different mechanism (tests and seeds respectively) does the binding instead.

Exercise 7

The solution for this exercise is in the instructor guide, which is distributed separately to course instructors. Self-study readers who want to check their answer can compare the two autogradable values against those reported in the chapter's own 'The cohort, continued' section.

Exercise 8

The chunk below runs both scans when the book is built, from this chapter's own directory.

```
scan_pkgs <- function(path) {
  found <- renv::dependencies(path, progress = FALSE)
  sort(unique(found$Package))
}
chapter_pkgs <-
 ↪  scan_pkgs('07-literate-programming.qmd')
pacman_file <- tempfile(fileext = '.R')
writeLines('pacman::p_load(dplyr, tibble, ggplot2)',
 ↪  pacman_file)
pacman_pkgs <- scan_pkgs(pacman_file)
simpson_pkgs <- c('dplyr', 'tibble', 'ggplot2')
```

On this build the scan of the chapter returns `dplyr`, `ggplot2`, `knitr`, `pkgload`, `renv`, `rmarkdown`, `survival`, and `tibble`. Of the `fig-simpson` chunk's three packages it finds `dplyr`, `tibble`, and `ggplot2`.

Draft, November 13, 2026. Content may change.

The scan of the one-line `pacman` file returns `dplyr`, `ggplot2`, `pacman`, and `tibble`. The scan sees the packages passed to `p_load()`, and the claim that they go unrecorded is false for this version of `renv`.
What the result implies. The `pacman` landmine does not rest on invisibility to `renv`. It rests on the render-time install: `p_load()` fetches whatever version is current on any machine that lacks a package, and stops the render on any machine that lacks `pacman`. The wider lesson concerns coverage claims. A belief about what a scanner sees, including the gate of Chapter 6, is an empirical claim about one tool at one version. Run the scan on a small file whose answer you know before relying on the tool, in the spirit of habit 4.

Answers to the checks

Check 1. *from earlier chapters* (page 340)

The colleague's different bootstrap interval comes from the session-configuration layer, not the R runtime. The R release sets the default random-number kind, so the image fixes that default at L2 (Chapter 3, Chapter 7). Her `~/.Rprofile` overrides the default, and an override is session configuration, fixed only by an explicit `RNGkind()` call in the analysis code. The image did not prevent it because Apptainer mounts the host `$HOME` by default, and `--cleanenv` drops only environment variables, so R inside the container still reads her startup file (Chapter 8). Two changes close it: an explicit `RNGkind()` call in the analysis, and a run with `--no-home` and `Rscript --vanilla`, so that no startup file is read.

Check 2. *capture or validation, for a render* (page 342)

No. Sealing the render pins no new layer of the determinant stack: the analysis code it runs was already locatable, at L0, because it sits in a version-controlled document. The seal captures nothing new; it validates that the document's prose agrees with that already-committed code, by construction, every time the document is built. A sealed, unpinned-package document and an unsealed, fully-pinned one are both possible, which is exactly why the seal and the ladder are different virtues.

Check 3. *the analyst's third judgment* (page 344)

Draft, November 13, 2026. Content may change.

Because a sealed render delivers only one thing, that the document agrees with its own code. It says nothing about whether the packages are pinned, the environment is fixed, or the outputs have been shown to regenerate from a clean rebuild. The analyst's third judgment named above, matching the claim to what the render seals, is exactly the discipline that stops a successful render from being overclaimed.

Check 4. *four tools, one idea* (page 347)

Each interleaves narrative prose with executable code in one source document. Each also weaves the code's output into the rendered result at build time, so the figure or number a reader sees is produced by the code beside it. That property, not the particular markup language or output format, is what secures the seal; the four tools are successive engineerings of the same idea Knuth proposed in 1984.

Check 5. *advertisement and scholarship* (page 347)

The scholarship is the complete software environment and the complete set of instructions that generated the figures, not the printed figures themselves. A literate document collapses the two because the figure a reader sees is produced, at build time, by the code embedded in the same document. The advertisement is therefore the output of the scholarship rather than a hand-copied picture of it, and cannot misrepresent a result the code does not produce.

Check 6. *which numbers are sealed* (page 351)

The enrollment target is a constant of the study design, not a result of the analysis. It was fixed by the protocol before any data were collected, and no quantity in the dataset determines it. Writing it as an inline expression would make a design input masquerade as a computed result, and would break, or silently misreport, if the observed sample size happened to differ from the target. Design constants are properly typed; only quantities the analysis itself produces belong in an inline expression.

Check 7. *diagnosing a landmine* (page 354)

The two candidates are an absolute or author-specific path, and an uncommitted file. They are distinguished by inspecting version control. If `data/interim.csv` is tracked and committed, the failure is more likely a path problem or a working-directory assumption. If `git status` or `git`

Draft, November 13, 2026. Content may change.

`ls-files` shows the file is untracked, it is the uncommitted-file landmine. In that case the file must be committed (or the chunk that generates it must be included in the render) before the render can seal. Rendering from a fresh clone of the repository reproduces the CI failure locally and settles the matter.

Check 8. *choosing the profile* (page 356)

Nothing needs to change now; an HTML-only project does not need the publishing profile's LaTeX distribution and should not pay its build time or image size before it is needed. What matters is recognizing, once the PDF output is added, that the project must move to the publishing profile (built on `rocker/verse`). It must also confirm that any package-specific LaTeX dependencies are present, rather than installing `tlmgr` packages ad hoc on whichever machine first attempts the PDF render. Waiting to decide is fine; waiting to act once the YAML declares `pdf_document2` reintroduces exactly the render-time, machine-specific fragility this chapter warns against.

Check 9. *two PDFs, one analysis* (page 358)

No. The checksums differ because of metadata that has nothing to do with the analysis, a creation timestamp, a document identifier, font-subset names, not because the figures or numbers changed. The right question is whether the computational outputs, the numbers and figures the document presents, regenerate within a stated tolerance. That question is answered by a baseline test that recomputes the outputs and compares them with recorded values within that tolerance, not by hashing the typeset file. A rendered document's bytes are not the object of record; the computation is.

Check 10. *the package that renders nothing* (page 359)

No. The R-package archetype has no single analysis to report, so it has no `report.Rmd` to seal. Its literate surface is the vignette and the roxygen-generated help page, and its correctness is established by `R CMD check` and the test suite rather than by a rendered result. Demanding a manuscript-style sealed report of a package applies this chapter's heaviest-weight case to the one archetype where the machinery is largely absent.

Check 11. *why day one, not day thirty* (page 360)

Draft, November 13, 2026. Content may change.

Nothing would be lost about any individual number's seal, if the author were careful. What is lost is a guarantee: starting on day one leaves no period in which numbers are copied, so the copying habit never forms. A month of scripts-first work creates thirty days during which a results-in-progress tempts an author to paste a number into a slide or an email. The habit of pasting is exactly what later infects the manuscript once it starts. Starting the document on day one removes the opportunity for that habit to form, rather than merely disciplining the author once it already has.

Check 12. *what the header's* `date` *field buys* (page 362)

A literal date would freeze at whatever day it was typed, so a document rendered months later would still carry its first day's date, silently misrepresenting when the reader's copy was actually produced. It is the same seal, applied to metadata rather than to a result: the date is computed from the render itself, so it cannot drift from the moment it claims to describe.

Check 13. *what the cohort passage does and does not show* (page 363)

No, for the same reason Chapter 11 presses at length. An identical rerun shows that the document's prose agrees with its own code. It also shows that the code is at least stable across the two machines on the one layer this chapter touches. It does not show that `clean_cohort()`'s missingness calculation is the one the analyst intended. That correctness question belongs to a unit or data-validation test with an independent oracle, not to the seal, which confirms agreement between prose and code and nothing about whether the code is right.

9.18. Cheat sheet

Command or rule	What it does
`library(pkg)`, one per package	Names a dependency in the form every scanner recognizes and loads without installing; prefer it to `pacman::p_load()`, which installs at render time.
An inline R expression in prose	Seals a number in prose to the code that computes it.

Draft, November 13, 2026. Content may change.

Command or rule	What it does
`knitr::opts_chunk$set(...)`	Sets chunk defaults, `echo`, `message`, `warning`, `fig.path`, once near the top.
`SOURCE_DATE_EPOCH`	Fixes the embedded build timestamp, so two renders agree on that one field.
`make docker-render`	Renders inside the pinned container and stamps the provenance footer (habit 6).
Five landmines	Installing at render time; `pacman::p_load()`; an absolute host path; hidden cross-chunk state; reading an uncommitted file.
Rule of thumb	A sealed render binds prose to code; it pins no package, no environment, and proves nothing correct.
Rule of thumb	Compare the computational outputs with a recorded baseline within a stated tolerance (habit 5), not the rendered PDF or HTML; the render's bytes are not the object of record.
Habit 6	Rebuild from clean, on a machine that is not yours, to test whether the seal actually holds.
Habit 7	Write the honest level down; 'it renders' is a claim about the seal, not about L1, L2, or L3.
`freeze: auto`	Re-executes a document only when its source changes; data, package, and R changes do not trigger it (Section 9.8)

Draft, November 13, 2026. Content may change.

9.19. Further reading

The foundational statement of literate programming is Knuth (1984), which repays reading in the original for its inversion of the usual priority of code over explanation, even though its concrete apparatus is now of historical interest. The reproducibility argument this chapter takes as its keystone is Buckheit & Donoho (1995). Their paraphrase of Claerbout, that the article is advertising and the code and instructions are the scholarship, is the sentence this book takes as its keystone, and the one worth committing to memory. The earlier formulation in Claerbout & Karrenbach (1992) and the signal-processing lineage surveyed by Vandewalle et al. (2009) give the argument its original setting.

On the R realizations, Xie (2015) is the definitive treatment of `knitr` by its author, and Xie et al. (2018) documents R Markdown and its `pandoc`-based toolchain. Allaire et al. (2024) introduces Quarto and is the natural starting point for a reader beginning today. The compendium framing that binds the literate document to the rest of the project is Gentleman & Temple Lang (2007), who first proposed the compendium as an executable document, and Marwick et al. (2018), whose `rrtools` operationalizes it around a `report.Rmd`. For the practical habits, Sandve et al. (2013) and Wilson et al. (2017) both stress recording exactly how each figure was produced, which is the sealed render stated as a rule of conduct. The determinism questions touched on here connect to the verification machinery of Chapter 12. In the wider setting of reproducible builds, they also connect to a substantial body of systems literature, beyond the scope of this book, that a reader pursuing byte-level reproduction of documents will want to consult.

Draft, November 13, 2026. Content may change.

10. Automating the Analysis: Make and targets

Avoid Manual Data Manipulation Steps.

Sandve, Nekrutenko, Taylor, and Hovig, *Ten Simple Rules for Reproducible Computational Research*, Rule 2 (2013)

This chapter concerns the multi-step analysis that someone runs by hand, script after script. By its end you will be able to replace the running of such an analysis with a single command whose completeness does not depend on anyone's memory. That command rests on a dependency graph that a tool, not an analyst, walks to decide what has gone stale and what has not.

Before you start

Reading time. About 55 minutes.
Prerequisites. The determinant stack and the capture-versus-validation distinction of Chapter 3 (Chapter 3). This chapter's central claim, that a pipeline validates re-execution without capturing any new layer, is stated entirely in those terms. The sealed, reproducible render of Chapter 9 (Chapter 9). A pipeline is what a growing analysis needs once some of its computation has moved out of that single document and into scripts. Comfort writing an R function with explicit arguments and a return value, reading a short shell command, and no prior exposure to Make, `targets`, or any other build tool. In the one-quarter core path (see the Conventions page), only the Make section below is core. The `targets`, Snakemake, Nextflow, and `workflowr` sections that follow it are marked optional, and they can be skipped on that schedule without losing the chapter's central claim.
By the end of this chapter you will be able to answer:

- Can you diagnose the failure mode of a manually run, multi-step analysis: steps run out of order, an upstream change leaves a downstream output stale, and no one can be sure the whole thing still runs end to end from raw data to report?
- Can you express an analysis as a directed acyclic graph of steps, and explain why a single command can then rebuild exactly what is out of date and nothing more?
- Can you read and write a GNU Make rule, its target, prerequisites, and recipe, explain the timestamp-based staleness check, and say why Make's reach is language-agnostic?
- Can you build an R-native, function-oriented pipeline with `targets`, including `_targets.R`, `tar_target()`, `tar_make()`, and `tar_visnetwork()`, and explain how `targets` invalidates exactly the targets that depend on a changed function, object, or package?
- Can you state, in the book's capture-versus-validation terms, precisely what a pipeline does and does not add to the reproducibility ladder, and judge how heavily the pipeline machinery bears on each research archetype?

Outline.

- Where this sits on the stack: a pipeline validates, it does not capture.
- Orientation: a report stale behind a corrected dataset.
- The analyst's contribution: granularity, honest dependency declarations, and the level a pipeline does not establish.
- The analysis as a dependency graph: the DAG, and its cost measured rather than asserted.
- GNU Make: targets, prerequisites, rules, and the timestamp check.
- `targets`: a function-oriented pipeline that hashes code and objects, not just file times.
- What a pipeline does, and does not, add to the ladder.
- Pipelines across the archetypes.
- Beyond R: Snakemake, Nextflow, and `workflowr`.
- Worked example: refactoring four scripts into a `targets` pipeline.

Draft, November 13, 2026. Content may change.

- The cohort, continued: the compendium's own two-step graph, timed.
- A language model inside the pipeline: a dependency no rung pins, and its responses kept as data.

Check 1: from earlier chapters

A collaborator updates `survival` to a newer release and commits the new `renv.lock`. You pull the commit, run `renv::restore()`, and render the whole Quarto project, whose `_quarto.yml` sets `freeze: auto`. The hazard ratio on the page is unchanged, under a fresh build date. Drawing on Chapter 6 and Chapter 9, which determinant layer changed, can the page tell you whether the new release moved the estimate, and what render would?
The answer is on page 419.

10.1. Where this sits on the stack

We begin by locating the chapter on the determinant stack. Its claim is the sharpest negative result in the book so far: the pipeline tools it teaches pin no new determinant at all. They do not fix a package version, and they do not fix an operating system or an R runtime; that is the work of Chapter 6 and Chapter 7. What a pipeline does is confirm, mechanically and on every run, that the recorded analysis code and the input data currently in hand are the ones a given output reflects. We therefore mark, in Figure 10.1, the analysis-code and input-data layers as validated here, not captured. This is the same reading Chapter 11 gives its own test suite for the same two layers, though by a different mechanism: a test suite checks that the code is *correct* on some input, where a pipeline checks only that an output is *current* with respect to whatever input it was last given, correct or not.

```
plot_determinant_stack(
  validates = c('Analysis code', 'Input data'),
  title = 'A pipeline validates currency; it captures no
   ↪  layer')
```

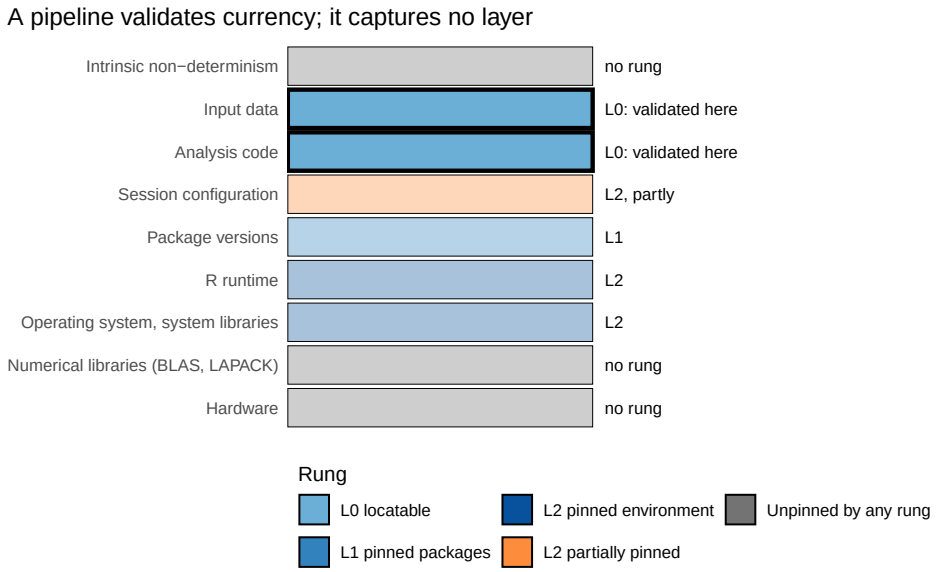


Figure 10.1.: The determinant stack of Chapter 3, with the analysis-code and input-data layers marked as validated, not captured, by a pipeline tool. Make and `targets` confirm that a recorded output is current with respect to the code and data that produced it; they do not pin a package version, an operating system, or an R runtime, and they say nothing about whether the code or the data are correct, only whether the output in hand still reflects them.

Check 2: capture or validation, again

A colleague argues that adding a `_targets.R` pipeline to a project must raise its reproducibility level, since the project can now be rebuilt with one command instead of several remembered ones. Using this chapter's stack figure, is the colleague correct?

The answer is on page 419.

10.2. Orientation

Consider an analysis that has grown, as most do, one step at a time. There is a script that reads the raw export and writes a cleaned dataset. A second fits a model to the cleaned data and saves the fitted object. A third draws the figures from the model, and a report assembles the figures and the tables into a manuscript. Each script runs, and the analyst runs them in order, by hand, at the console. For a while this works.

The trouble begins when the raw data are corrected. The analyst re-runs the cleaning script, and now must remember three further things. The model depends on the cleaned data and must be re-run; the figures depend on the model and must be re-run; and the report depends on the figures and must be re-rendered. Each of these is a separate act of memory. Miss one, and the report now shows a figure drawn from the old model, fitted to the old data, and nothing in the document announces the discrepancy. The output is stale: it is downstream of a change that never propagated to it. Worse, the staleness is invisible, because a stale figure is a perfectly valid PNG file that opens and displays without complaint.

This is the failure this chapter exists to prevent, and it has three faces. The first is **order**. The steps have a required sequence, cleaning before modeling before plotting, and running them out of order produces either an error or, more dangerously, a result computed from the wrong inputs. The second is **staleness**. A change to an upstream file leaves every downstream output obsolete, and re-propagating the change by hand is exactly the kind of bookkeeping a human does poorly. The third is the **end-to-end doubt**. After a session of editing scripts and re-running some of them, the analyst cannot honestly say whether the whole analysis still runs from raw data to finished report. It has been weeks since anyone ran it all, in order, in one act. The project works, in the sense that its outputs exist, but no one can be sure it would work if rebuilt from nothing.

How, then, are we to remove memory from the process? The remedy is to stop treating the analysis as a remembered sequence of commands and to express it instead as a structure. That structure is a graph of steps, each declaring what it consumes and what it produces. From it a machine can work out the order, detect the staleness, and re-run exactly what a change requires. Such a structure is a pipeline, and this chapter is about two tools

Draft, November 13, 2026. Content may change.

that build one, the venerable and language-agnostic GNU Make and the R-native `targets`. The manual-steps critique that motivates them is stated crisply in the practices literature. Sandve et al. (2013), in their second rule, urge that one 'avoid manual data manipulation steps' and automate the path from raw data to result. Their reason is that a manual path cannot be trusted to have been followed the same way twice.

Check 3: naming the failure

An analyst corrects a typo in the raw data, re-runs the modeling script because that is the step they are thinking about, and ships the resulting report. The figures were drawn from the model fitted before the correction. Which of the orientation's three faces, order, staleness, or end-to-end doubt, does this failure illustrate, and why does the stale figure give no warning?
The answer is on page 420.

10.3. The analyst's contribution

Three judgments in this chapter are the analyst's own, and no pipeline tool makes them.

1. **Deciding where the steps divide.** A pipeline is a graph of steps, and the analyst chooses what a step is. That is, the analyst decides how finely to cut the analysis into targets, which computations to group and which to separate. Cut too coarsely, into one enormous step, and the pipeline caches nothing useful, because any change re-runs everything. Cut too finely, into hundreds of trivial targets, and the graph becomes hard to read and the bookkeeping overhead dominates. The right granularity puts the expensive, stable computations, a model fit, a simulation run, in targets of their own, so that a change elsewhere does not needlessly re-run them. The choice trades caching benefit against graph legibility.

2. **Keeping the dependency declarations honest.** A pipeline can only track dependencies it is told about, or can infer. Make infers

Draft, November 13, 2026. Content may change.

nothing. Every prerequisite an analyst omits from a rule is a dependency the tool cannot see, and an omitted prerequisite is exactly how a stale output slips through. `targets` infers a great deal automatically, but it too can be defeated, by a step that reads a file without declaring it as a file target, or that reaches for global state the tool cannot trace. Getting the declared dependencies to match the real ones is the analyst's responsibility, and a pipeline's correctness rests on it.

3. **Matching the claim to what a pipeline delivers.** A pipeline mechanizes re-execution. It does not pin a package, fix an environment, or establish that a result is correct. An analyst who says 'the analysis is reproducible' because `tar_make()` runs to completion has claimed more than the pipeline delivers. This is the same overreach, for the same reasons, as treating the sealed render of Chapter 9 as a reproducibility guarantee. The pipeline secures the act of re-running. The level reached is a separate question, settled by the lockfile, the container, and the verification of the later chapters.

Check 4: whose judgment, again

A `targets` pipeline runs green, end to end, on the analyst's machine. Which of the three judgments above, if any, has the green run settled, and which remain open?
The answer is on page 420.

The vocabulary of this chapter

- **Pipeline.** An analysis expressed as a set of steps, each declaring its inputs and outputs, so that the whole can be re-executed by a single command that runs the steps in dependency order.
- **Dependency graph.** The graph whose nodes are the steps and whose edges record that one step consumes what another produces. A valid pipeline's graph is a *directed acyclic graph* (DAG): directed because inputs flow to outputs, and acyclic because a step cannot, directly or transitively, depend on its own output.
- **Target.** A single step in the pipeline, together with the output

Draft, November 13, 2026. Content may change.

it produces. The word is shared by Make and `targets`, with a shift of emphasis: for Make a target is a file, for `targets` it is more often an R object.
- **Prerequisite.** In Make, an input a target depends on. If any prerequisite is newer than the target, the target is out of date.
- **Recipe (or command).** The shell command Make runs to bring a target up to date from its prerequisites.
- **Staleness (out-of-date).** The condition of a target whose inputs have changed since it was last built, so that its stored output no longer reflects its inputs.
- **Up-to-date (skippable).** The condition of a target whose inputs have not changed since it was last built, so that rebuilding it would reproduce the stored output and can be skipped.
- **Invalidation.** The act of marking a target out of date because something it depends on, a file, an upstream target, a function, or a package, has changed.

10.4. The analysis as a dependency graph

Having named the failure, we now turn to the structure that prevents it. The conceptual move that a pipeline asks of an analyst is to stop thinking of the analysis as a list of commands to run. Instead, the analyst thinks of it as a graph of dependencies to satisfy. The two descriptions contain the same steps, but they license different machinery. A list of commands can only be replayed, from the top, every time. A graph of dependencies can be interrogated, so that a tool can answer the question a list cannot: which steps actually need to run now, given what has changed?

Figure 10.2 draws the graph for the small analysis of the orientation. Each node is a step that produces an output, and each arrow records that the step at its head consumes what the step at its tail produces. The graph is directed, because the flow from raw data to report has a direction, and it is acyclic, because no output feeds, however remotely, back into its own inputs. This is the directed acyclic graph, the DAG, that every pipeline tool builds internally, whether or not it draws it for the reader.

Draft, November 13, 2026. Content may change.

```
library(tibble)
library(ggplot2)

nodes <- tibble(
  step = c('raw\ndata', 'cleaned\ndata', 'model',
           'figures', 'report'),
  x = 1:5,
  y = 0
)

edges <- tibble(
  x = 1:4,
  xend = 2:5,
  y = 0,
  yend = 0
)

ggplot() +
  geom_segment(
    data = edges,
    aes(x = x + 0.30, xend = xend - 0.30,
        y = y, yend = yend),
    arrow = grid::arrow(length = grid::unit(0.16, 'cm'),
                        type = 'closed'),
    linewidth = 0.5, colour = 'grey45'
  ) +
  geom_label(
    data = nodes,
    aes(x = x, y = y, label = step),
    size = 3, lineheight = 0.9,
    label.padding = grid::unit(0.35, 'lines'),
    label.r = grid::unit(0.12, 'lines'),
    fill = 'grey96', colour = 'grey15'
  ) +
  coord_cartesian(xlim = c(0.5, 5.5),
                  ylim = c(-1, 1), expand = FALSE) +
  theme_void()
```


Draft, November 13, 2026. Content may change.

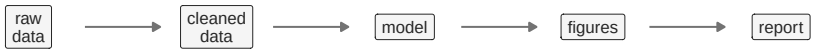


Figure 10.2.: The analysis of the orientation as a directed acyclic graph. Each node is a step that produces an output; each arrow records that the step it points to consumes what the previous step produced. A change to the raw data invalidates every node downstream of it (all four to its right); a change to the plotting code invalidates only the figures and the report. A pipeline tool reads this graph to rebuild exactly the stale nodes and skip the rest.

Two operations on this graph are the whole point of a pipeline. The first is the **topological sort**. Reading the arrows, the tool can order the steps so that no step runs before the steps it depends on. This is the order the analyst was keeping in their head and now need not. The second is **selective rebuild**. Given a record of what each step last consumed, the tool can mark a step stale when any of its inputs has changed and up to date otherwise. It then rebuilds only the stale steps and those downstream of them. If the raw data change, all four downstream nodes are stale and all four rebuild. If only the plotting code changes, the figures and the report are stale but the cleaned data and the model are not. The model, which may have taken an hour to fit, is not re-fitted. The single command that triggers this, `make` or `tar_make()`, replaces the remembered sequence with one reliable act. The end-to-end doubt of the orientation is thereby dissolved, because the one act, by construction, runs everything that is out of date and leaves nothing behind.

Draft, November 13, 2026. Content may change.

10.4.1. The graph of an actual analysis

Figure 10.2 draws the shape in the abstract. But what does the shape cost over an analysis that actually runs? There the same shape is less tidy than the diagram, and the argument below depends on that untidiness.

```
library(tibble)
library(knitr)

time_it <- function(expr) {
  t <- system.time(force(expr))[['elapsed']]
  round(1000 * t, 1)
}

d <- survival::diabetic
t_load <- time_it(d <- survival::diabetic)
t_clean <- time_it({
  d$arm <- factor(d$trt, levels = c(0, 1),
                  labels = c('Untreated', 'Laser'))
  d <- d[!is.na(d$time), ]
})
t_fit <- time_it(
  fit <- survival::coxph(survival::Surv(time, status) ~ arm
 ↪  + age,
                         data = d)
)
t_boot <- time_it({
  set.seed(1)
  replicate(200, {
    i <- sample(nrow(d), replace = TRUE)
    unname(coef(survival::coxph(
      survival::Surv(time, status) ~ arm + age, data = d[i,
 ↪  ]))[1])
  })
})

steps <- tibble(
  Step = c('read raw data', 'derive arm labels',
```


Draft, November 13, 2026. Content may change.

```
            'fit proportional-hazards model',
            'bootstrap the treatment effect'),
  `Consumes` = c('--', 'raw data', 'cleaned data',
                 'cleaned data'),
  `Elapsed (ms)` = c(t_load, t_clean, t_fit, t_boot)
)

kable(steps, align = 'llr')
```

Table 10.1.: The retinopathy analysis expressed as a dependency graph, with the cost of each step measured rather than asserted. The timings are from one run on one machine and will differ on another, and repeated runs on the same machine have varied by several-fold; even the ratios between steps should be read as illustrative, not as a stable property of the analysis. Note that the expensive step sits at the end, so an edit to the plotting code that forced a full rebuild would waste almost all of the total.

Step	Consumes	Elapsed (ms)
read raw data	–	0
derive arm labels	raw data	0
fit proportional-hazards model	cleaned data	5
bootstrap the treatment effect	cleaned data	270

On this build the bootstrap took 270 milliseconds, at least 98 percent of the four steps' total, and none of the other three took more than 5 milliseconds. An analyst who re-runs everything after changing a label pays for the bootstrap to learn nothing. Every figure in these two paragraphs is an inline expression, for the reason Chapter 9 gives: had they been typed, the next machine to build the book would have made them false.

The table also carries a caveat about itself. These timings are computed when the book is built, so they describe the build machine and not the reader's. A slower machine would show different absolute values, and so can a second run on the same machine. Even the ratio of the bootstrap to the

Draft, November 13, 2026. Content may change.

model fit is unstable. The fit took 5 milliseconds on this build, a quantity so small that a millisecond or two of timer noise in that denominator moves the ratio several-fold from one run to the next. What held in every repeated run is the ordering: the bootstrap dominates the total, and the steps before it are negligible. That ordering, not any single number, is what the table supports. This is the same distinction the verification chapter draws between a result worth recording and a measurement worth repeating.

Check 5: reading the measured costs

Table 10.1 shows the bootstrap taking almost all of the total time. An analyst who changes only the color of a figure, then reruns the whole four-step chain by hand, without a pipeline, pays for which step unnecessarily? And which feature of the table should be expected to survive a change of machine?
The answer is on page 420.

10.5. GNU Make: targets, prerequisites, and rules

We turn now from the idea to the tools. The oldest and most widely used pipeline tool is GNU Make, a reimplementation of the `make` utility first written in 1976 to compile programs. It has been pressed into service for data analysis ever since. Make is worth learning first, before any R-specific tool, for two reasons. First, its model is the simplest possible expression of the pipeline idea, so that understanding it is understanding pipelines in general. Second, its reach is language-agnostic. A single Makefile can orchestrate an analysis whose steps are written in R, Python, the shell, and a typesetting engine, without knowing or caring what language each step uses.

A Makefile is a list of **rules**, and a rule has three parts. The first is a **target**, the file the rule produces. The second is its **prerequisites**, the files the target depends on, written after a colon. The third is a **recipe**, the shell command that builds the target from its prerequisites. The recipe is written on the following line and, by an unforgiving convention, indented with a literal tab.

Draft, November 13, 2026. Content may change.

> **Pitfall**
>
> A text editor set to insert spaces for indentation will silently turn that leading tab into spaces, and what Make says then depends on the line. To see it, take the four-rule Makefile below and replace the tab before `Rscript make_figures.R` with eight spaces. GNU Make stops with `Makefile:5: *** missing separator (did you mean TAB instead of 8 spaces?). Stop.` With four spaces the hint disappears, and the bare 'missing separator' names a line but not the cause. A spaced recipe that contains a colon, such as the `rmarkdown::render()` line of the first rule, raises no error at all. Make reads it as one more rule, and `report.html` is left with no recipe. For a permanent fix, configure the editor to preserve a literal tab inside files named `Makefile`.

The analysis of Figure 10.2 becomes four rules:

```
report.html: report.Rmd figures.rds
    Rscript -e 'rmarkdown::render("report.Rmd")'

figures.rds: make_figures.R model.rds
    Rscript make_figures.R

model.rds: fit_model.R cleaned.rds
    Rscript fit_model.R

cleaned.rds: clean_data.R raw_data.csv
    Rscript clean_data.R
```

If we read the rules from the bottom up, the dependency graph is plain. `cleaned.rds` depends on the cleaning script and the raw data; `model.rds` depends on the fitting script and the cleaned data; and so on up to the report. Make never has to be told the order, because the order is implied by the prerequisites. It reads the whole Makefile, builds the graph, sorts it, and knows that `cleaned.rds` must be built before `model.rds` because the latter names the former as a prerequisite. To rebuild the analysis the analyst types a single command:

Draft, November 13, 2026. Content may change.

```
make report.html
```

or, since the first target in the file is the default, simply `make`. Make walks the graph and rebuilds what is needed, in order, and reports each recipe as it runs.

But how does Make decide what is needed? The mechanism is the one subtlety worth dwelling on, because it is both the source of Make's power and the source of its most common surprise. Make is **timestamp-based**. It compares the modification time of each target against the modification times of its prerequisites. It considers a target out of date if any prerequisite is newer than the target. Suppose we edit `fit_model.R`; its modification time advances past that of `model.rds`. Make sees that `model.rds` is now older than a file it depends on and rebuilds it. That rebuild advances the timestamp of `model.rds` past that of `figures.rds`, and so the rebuild propagates downstream exactly as far as the change reaches and no further. A target all of whose prerequisites are older than it is already up to date. Make skips it, printing a line to the effect that there is nothing to be done. This is the selective rebuild of the previous section, implemented with nothing more than the file system's clock.

The timestamp rule is also Make's principal limitation. Make knows only whether a file is newer, not whether its contents have changed. Touching a file without editing it will trigger a needless rebuild. More subtly, a change that does not alter a file's modification time will be missed. Make also tracks only what the analyst declares. It has no way to know that `fit_model.R` reads a configuration file unless that file is listed as a prerequisite. An undeclared input is invisible, and its change will not trigger the rebuild it should. Finally, Make tracks files and only files. It cannot cache an in-memory R object, so every step must serialize its result to disk, and the graph is a graph of files. These limitations are the opening that the R-native tools were built to close.

Check 6: what Make rebuilds

A project uses the four-rule Makefile above. The analyst edits `make_figures.R` to change a color palette and runs `make`. Which of the four targets does Make rebuild, and which does it skip? Explain

Draft, November 13, 2026. Content may change.

your answer in terms of the timestamp rule.
The answer is on page 420.

10.6. targets: a function-oriented pipeline for R

Make treats an analysis as a graph of files built by shell commands. For an R analysis this is a slightly awkward fit. Every step must be a script that reads files and writes files, and the intermediate objects must all be serialized to disk by hand. Moreover, Make's timestamp check is blind to the thing an R analyst most wants tracked, the R functions that do the work. The `targets` package, written by William Landau as the successor to his earlier `drake` (Landau, 2018, 2021), reworks the pipeline idea natively for R. It closes exactly these gaps.

The reframing is captured in the phrase 'function-oriented'. In a `targets` pipeline the analyst writes the analysis as a set of R functions, one for each meaningful step. The analyst then declares a list of targets that connect those functions into a graph, in a special script named `_targets.R` at the project root. Each target is created by `tar_target()`, which names the target and gives the R expression that produces it. The package reads those expressions and notices that one target's expression mentions another target's name. From this it infers the dependency graph automatically, with no colons and no prerequisites to maintain by hand. A `_targets.R` for our running analysis reads:

```
library(targets)
library(tarchetypes)
tar_option_set(packages = c('dplyr', 'ggplot2'))

# Analysis functions live in R/ and are sourced here.
tar_source('R')

list(
  tar_target(
    raw_file,
    'data/raw_data.csv',
```


Draft, November 13, 2026. Content may change.

```
    format = 'file'
  ),
  tar_target(raw_data, read_raw(raw_file)),
  tar_target(cleaned, clean_data(raw_data)),
  tar_target(model, fit_model(cleaned)),
  tar_target(figures, make_figures(model, cleaned)),
  tar_render(report, 'report.Rmd')
)
```

The last line needs a word, because it is where a hand-written pipeline most often goes wrong. A report is a file produced from another file, the `report.Rmd` source. The target that renders it must therefore track that source as well as the results it displays. `tar_render()`, from the companion package `tarchetypes`, does both. It tracks `report.Rmd` as a file, and it reads the document for calls such as `tar_read(model)` and `tar_load(figures)`, making each target so named a dependency of the report. Inside `report.Rmd`, results are therefore fetched with `targets::tar_read()` rather than recomputed. For a Quarto document, `tar_quarto()` plays the same role. The tempting alternative is a plain `tar_target(report, render_report(model, figures), format = 'file')`. It tracks the model, the figures, and the rendered HTML, but not the source. In a scratch run with `targets` 1.12.0 and `tarchetypes` 0.14.0, editing the report's prose left `tar_outdated()` empty under that pattern, while under `tar_render()` it listed `report`. A report that does not rebuild when its own text changes is a stale report that the pipeline calls current.

Several features distinguish this from the Makefile. The intermediate results, `raw_data`, `cleaned`, `model`, `figures`, are **R objects, not files**. `targets` stores each in a hidden `_targets/` store, keyed and versioned, and hands it to the next target as a plain R value. The analyst never writes a `saveRDS` or a `readRDS`, because the storage is the tool's job. Where a real file must be tracked, such as the raw CSV at the top, the target is declared with `format = 'file'`. `targets` then watches the file's contents, by hashing them, rather than the R object. `tar_render()` does the same for the report's source and its rendered output.

That hashing is the second and deeper difference. Where Make compares

Draft, November 13, 2026. Content may change.

timestamps, `targets` compares **content hashes**, so that a file touched but not changed does not trigger a rebuild. The hashing is not unconditional, however. When the `trust_timestamps` option is left unset, `targets` chooses by file system. On a file system with precise time stamps it trusts them: it compares a tracked file's modification time with the one it recorded and recomputes the hash only if the two differ. A file whose content changed while its time stamp did not is therefore *not* caught. Such changes are rare in ordinary editing, but copying tools that preserve time stamps, and file systems with coarse time stamps, can produce them. In a scratch run with `targets` 1.12.0, a one-digit edit to the raw CSV with its time stamp restored left `tar_outdated()` empty under the default. Under `tar_option_set(trust_timestamps = FALSE)`, by contrast, it listed every target downstream of the file, at the cost of hashing every tracked file on every check.

The hashing also extends past the data to the code. `targets` records a hash of the body of every function a target calls, and of the R expression in the target itself. Editing `fit_model()` therefore invalidates the `model` target, and everything downstream of it, even though no file has changed at all. Make, seeing only files, cannot detect such a change. A changed function invalidates exactly the targets that use it, directly or transitively, and no others. A change to `make_figures()` invalidates `figures` and `report` but leaves `model` untouched, and the expensive fit is preserved. Sensitivity to the *versions* of the packages a target loads is a separate matter, and one that `targets` does not provide. `tar_option_set(packages = ...)` merely attaches those packages to each target's evaluation environment, as a call to `library()` would. By default `targets` does not track a package's version or invalidate a target when that version changes. Version-sensitivity comes from the lockfile of Chapter 6, which is why Landau recommends using `renv` alongside `targets`.

Two commands drive the pipeline. To see the graph, `tar_visnetwork()` draws it as an interactive network, coloring each target by whether it is up to date or out of date. This is Figure 10.2 made live and queryable:

```
tar_visnetwork()
```

To run the pipeline, `tar_make()` walks the graph, in a fresh R session, and

Draft, November 13, 2026. Content may change.

builds every target that is out of date while **skipping every target that is up to date**:

```
tar_make()
```

On the first run every target builds and `targets` reports each in turn. If we edit `make_figures()` and run `tar_make()` again, the output is eloquent. `targets` names `figures` and `report` as built, and reports `raw_data`, `cleaned`, and `model` as already current. In `targets` 1.12 and later the up-to-date targets are summarized as a count in the progress display rather than listed by name. The distinction between what rebuilt and what did not is unchanged. The analyst did not have to remember which steps the edit affected; the tool derived it from the graph and the hashes. A companion command, `tar_outdated()`, answers the end-to-end doubt directly. It lists the targets that a `tar_make()` would rebuild, without running anything. One can thus ask, at any moment, 'what is stale?' and get an exact answer.

Verification

To see this hold: in a project with `_targets.R`, run `tar_make()` once, then edit the body of any function a target calls, change nothing else, and run `tar_outdated()` before rebuilding. It should name that target and every target downstream of it, and no others. To make it fail on purpose, add a line to a function that instead reads a file directly with `read.csv()` rather than through a declared `format = 'file'` target. Change that file's contents, and run `tar_outdated()` again. The target that silently reads the file will not appear, because `targets` cannot hash a dependency it was never told about. This is the same discipline failure Make suffers from an undeclared prerequisite.

Check 7: what `targets` tracks that Make does not

In the Makefile, editing the body of a function used by `fit_model.R` without changing any file's dependency declaration might not trigger a rebuild of the downstream targets. Explain why `targets` does trigger it, and name the two things `targets` hashes that Make does not.
The answer is on page 421.

Draft, November 13, 2026. Content may change.

10.7. What a pipeline does, and does not, add

With both tools in hand, we may now ask what they have bought us. What a pipeline contributes to reproducibility is easily overstated in one direction and undersold in the other. The book's capture-versus-validation framework allows a precise statement.

Recall the framework. A computational result depends on a stack of determinants: the operating system, the system libraries, the R version, the package versions, the session configuration, the data, the source. Reproducibility is secured by *capturing* each determinant, which is what a lockfile and a container do. It is secured also by *validating* that the capture suffices, which is what continuous integration and verification do. The ladder, L0 locatable, L1 pinned packages, L2 pinned environment, L3 verified, measures how far down the stack an effort has reached.

Against this framework the pipeline occupies a precise and initially surprising position: **a pipeline captures no new determinant and adds no rung to the ladder.** It does not pin a package version; that is the lockfile's work, unchanged by whether a pipeline exists. It does not fix an environment; that is the container's. It does not verify that an output is correct; that is verification's. An analysis run through `targets` and an identical analysis run by hand, on the same machine, with the same packages, produce the same numbers and sit at the same level of the ladder.

Yet it would be wrong to conclude that the pipeline is reproducibility-neutral. The reason returns us to the distinction that organizes the book. What a pipeline does is **mechanize the re-execution** that validation depends on. Verification, in Chapter 12, checks that rebuilding the analysis from its captured inputs reproduces the recorded outputs; continuous integration runs that rebuild on every change, in a clean machine. Both presuppose that 'rebuild the analysis' is a single, reliable, complete act. A manually run analysis cannot guarantee precisely that act, because the human running it may re-run some steps and forget others. The pipeline turns 'rebuild the analysis' from a remembered sequence into one command whose completeness is structural. In doing so it removes the 'did I re-run everything?' doubt that undermines every validation step. Like continuous integration, a pipeline captures no determinant and adds no

Draft, November 13, 2026. Content may change.

rung; it makes the re-execution that verification (Chapter 12) confirms reliable and repeatable.

The relation to continuous integration is especially close, and worth making explicit, because the two are sometimes confused. Continuous integration is the machine that is not ours, running the rebuild in a clean environment on every change. The pipeline is the definition of what 'the rebuild' means. They compose naturally. For a pipelined project, a CI workflow's core step is to restore the environment and then run `tar_make()` or `make`. The pipeline supplies the re-execution, and CI supplies the clean, automatic, external trigger for it. Neither replaces the other. The pipeline without CI still requires a human to remember to run it; CI without a pipeline must hard-code the sequence of steps that the pipeline would have derived from the graph. Together they close the loop. A change is pushed, a clean machine rebuilds the analysis, and the result is checked, with no act of memory anywhere in the chain. On a clean runner the `_targets/` store is absent, because it is not in the checkout, so the rebuild runs the whole graph, which is itself the clean-rebuild test. A runner rebuilds only what a change made stale when the store is restored between runs, from a cache or a cloud store.

Check 8: the composed claim

A team reports that 'every push rebuilds and tests the analysis, so it is reproducible.' Using this section's account of what CI and the pipeline each contribute, what exactly has been shown, and what rung, if any, has been demonstrated?

❗ The honest level. This chapter's own practice runs `tar_make()` and `make` to completion

10.8. Pipelines across the archetypes

We turn next to how much of this machinery each kind of project needs. The weight a pipeline tool carries varies sharply across the research archetypes; Table 5.2 in Chapter 5 is the reference. It is most useful for the **data-analysis** archetype's chain of transforms, run and re-run as the cleaning and the model are adjusted. It is stronger still for a **simulation study**. There `targets`' *branching*, creating one target per parameter setting from a single declaration, maps a whole parameter grid onto the dependency graph, so adding a setting rebuilds only the new cell. A **manuscript**'s pipeline is present but modest and often implicit, since the sealed `report.Rmd` of Chapter 9 is itself a small, one-target build. The sharpest contrast is the **R-package** archetype, which has no analysis pipeline at all. A package is a library of functions, not a chain of data transforms producing a report, and its 're-execution' is `R CMD check` (Chapter 11), not `tar_make()`. The **blog** archetype returns to a pipeline, but over documents rather than an analysis. Its natural rebuild touches only the posts whose sources changed, the role `workflowr` fills in the next section. Reach for the tool where re-execution is frequent or costly, and do not impose its overhead where a single render suffices.

> Check 9: an archetype with no pipeline
>
> A reader building an R package asks why this chapter never shows a `_targets.R` for it. What is the answer, and what plays the corresponding role?
> *The answer is on page 421.*

10.9. Beyond R: Snakemake, Nextflow, and workflowr

Make and `targets` are the right tools for an R-centric analysis. However, a reader whose work reaches beyond R, or beyond a single machine, should know the tools that dominate computational science more widely. In pipelines, the R ecosystem is a small part of a much larger field.

The two dominant language-agnostic pipeline managers come from bioinformatics. There, analyses routinely chain dozens of command-line tools across

Draft, November 13, 2026. Content may change.

terabytes of sequencing data and must run at the scale of a computing cluster or a cloud. **Snakemake**, by Köster and Rahmann, extends the Make model, rules with inputs and outputs, into a Python-based language. That language has pattern rules, wildcards, and native awareness of cluster schedulers, so that a single workflow definition can fan a computation out across thousands of jobs (Köster & Rahmann, 2012). **Nextflow**, by Di Tommaso and colleagues, takes a different, dataflow approach, connecting processes through channels. It is built from the ground up for portability across high-performance-computing clusters and cloud platforms, with containerization of each step as a first-class feature (Di Tommaso et al., 2017). Both far exceed Make and `targets` in their handling of scale and heterogeneity. Both are the correct choice when an analysis spans many languages and many machines. For a first-year health-sciences analysis run on a laptop or a single server in R, they are heavier than the problem requires. A reader moving into genomics or large-scale imaging, however, will meet them quickly and should recognize them as the same pipeline idea scaled up.

Closer to home, **`workflowr`**, by Blischak and colleagues, serves the specific case of a reproducible research website (Blischak et al., 2019). This covers the blog archetype above and the versioned analysis site more generally. Its contribution is less the pipeline than the binding of each rendered result to version control. `workflowr` rebuilds the documents that have changed and stamps each rendered page with the git commit of the source that produced it. Every published figure thus carries, visibly, the version of the code behind it. It is the natural tool where the deliverable is a website of analyses that must each be traceable to a commit. It also composes with the version-control discipline that runs through this book.

The pipeline idea, express the analysis as a dependency graph and rebuild only what is stale, is common to all five tools. The choice among them is a choice of scale and setting. Make is the language-agnostic minimum, and `targets` is the R-native tool that tracks objects and functions. Snakemake and Nextflow are the same idea at cluster and cloud scale, and `workflowr` is the idea applied to a versioned website. We now put the R-native tool to work on a concrete case.

Draft, November 13, 2026. Content may change.

Check 10: choosing among the five tools

A genomics collaborator's analysis chains forty command-line tools across a cluster, written in a mix of Python and shell. Which of this section's tools fits, and why would `targets` be the wrong reach here? *The answer is on page 422.*

10.10. Worked example

To illustrate, we convert the running analysis from a set of hand-run scripts into a `targets` pipeline, as a health scientist tidying a working project would. The analysis begins as four scripts in `analysis/scripts/`, run in order at the console. One cleans `data/raw_data.csv` and saves `cleaned.rds`, one fits a model and saves `model.rds`, one draws figures, and one renders the report. The project works, but every correction to the raw data sets off the manual re-running the orientation warned against.

The first step is to refactor the scripts into functions, before any `_targets.R` is written, because a function-oriented pipeline needs functions. Each script's body becomes a function in `R/`: `read_raw()`, `clean_data()`, `fit_model()`, and `make_figures()`. Each takes its inputs as arguments and returns its result rather than reading and writing files by side effect. The fourth script, the report, becomes `report.Rmd`. It fetches the model and the figures with `targets::tar_read()` instead of recomputing them, so that `tar_render()` can see what it depends on. This refactoring is worthwhile on its own. Functions with explicit inputs and outputs are testable in the sense of Chapter 11, where a script that reads and writes global files is not.

The second step is the `_targets.R` shown earlier in the chapter. It declares the six targets: the raw file, the read data, the cleaned data, the model, the figures, and the report. It then lets `targets` infer the graph from the way each target's expression names the targets before it. With the file in place, `tar_visnetwork()` draws the graph, and the analyst confirms it carries the intended structure of Figure 10.2. Every node is initially colored 'outdated' because nothing has been built yet. Figure 10.2 is a simplified linear schematic of the chain from raw data to report. The

Draft, November 13, 2026. Content may change.

actual dependency graph is more connected. The `_targets.R` declares that `figures` depends on both `model` and `cleaned` and that `report` depends on both `model` and `figures`. Hence `tar_visnetwork()` draws those extra cross-edges that the schematic omits.

The third step is to build the pipeline once, with `tar_make()`. This runs all six targets in dependency order in a fresh session and populates the `_targets/` store. A correction to the raw data, followed by `tar_make()`, rebuilds all five downstream targets and skips nothing, because everything depends on the data. A refinement to `make_figures()`, followed by `tar_make()`, rebuilds only `figures` and `report`. `raw_data`, `cleaned`, and `model` are left untouched, the model fit, the expensive step, among them. In current versions of `targets` the console reports the untouched targets only as a skip count. A change to the wording of the discussion in `report.Rmd` rebuilds `report` alone. At any moment `tar_outdated()` answers the question the manual workflow could never answer with confidence. It lists precisely the targets a rebuild would touch, so that the analyst always knows whether the project is current.

The final step ties the pipeline to the validation machinery of the surrounding chapters. The pipeline runs inside the pinned environment of Chapter 7, restored from the lockfile of Chapter 6. Thus `tar_make()` re-executes against fixed packages and a fixed R. The continuous-integration workflow of Chapter 12 is given one core instruction, to restore the environment and run `tar_make()`. Every push then rebuilds the whole pipeline on a clean machine, since the runner starts without a `_targets/` store. The ‘did I re-run everything?’ doubt is answered, permanently, by a machine that is not the author’s.

Check 11: the two-target split

The worked example’s `_targets.R` declares `raw_file` with `format = 'file'` and a separate target `raw_data` that reads it, rather than one target that both names the path and reads it. Why does the split matter, and what would go wrong without it?
The answer is on page 422.

Draft, November 13, 2026. Content may change.

Check 12: complete the `_targets.R`

The worked example's `_targets.R`, first shown in the section on `targets`, is reprinted here with four decisive lines replaced by numbered placeholders. Write each missing line, using only the functions and target names the worked example defines.

```
library(targets)
library(tarchetypes)
tar_option_set(packages = c('dplyr', 'ggplot2'))

# Analysis functions live in R/ and are sourced here.
tar_source('R')

list(
  tar_target(
    raw_file,
    'data/raw_data.csv',
    # (1) ____: hash the file's contents, not an R
     ↪ object
  ),
  # (2) ____: read the raw file through its target name
  tar_target(cleaned, clean_data(raw_data)),
  tar_target(model, fit_model(cleaned)),
  # (3) ____: draw the figures from the model and the
   ↪ data
  # (4) ____: render report.Rmd, tracking its source
)
```

The answer is on page 422.

10.11. The cohort, continued

This chapter has so far reasoned about pipelines through a composite analysis. We now apply the same idea to this book's own compendium. Here is a two-node graph, raw extract to cleaned cohort, with the cost of

Draft, November 13, 2026. Content may change.

each step measured, as Table 10.1 did for the retinopathy example.

```
cohort_path <- '../data/raw_data/cohort_raw.csv'

t_load_cohort <- time_it(
  raw <- load_cohort(cohort_path))
t_clean_cohort <- time_it(
  cohort <- clean_cohort(raw))

cohort_steps <- tibble(
  Step = c('load_cohort()', 'clean_cohort()'),
  Consumes = c('cohort_raw.csv', 'the raw data frame'),
  `Elapsed (ms)` = c(t_load_cohort, t_clean_cohort),
  `Rows out` = c(nrow(raw), nrow(cohort))
)

kable(cohort_steps, align = 'llrr')
```

Table 10.2.: The two steps `load_cohort()` and `clean_cohort()` perform on the running cohort compendium, with elapsed time measured on one run on the machine that built this chapter; repeated runs vary and should not be read as a stable property of either function.

Step	Consumes	Elapsed (ms)	Rows out
load_cohort()	cohort_raw.csv	1	338
clean_cohort()	the raw data frame	3	337

The two functions are already, in effect, a two-rule pipeline. `clean_cohort()` consumes what `load_cohort()` produces, and nothing downstream exists yet. No later chapter's model or report has been wired to this extract through `targets` or a Makefile. Suppose a correction were made to `cohort_raw.csv` today, by hand. The only way to know that `clean_cohort()`'s output had gone stale would be to remember to re-run this cell. That is exactly the end-to-end doubt the orientation opened with, now visited on the book's own running example. Expressing this

Draft, November 13, 2026. Content may change.

pair as three targets would make that staleness mechanical rather than remembered:

```
list(
  tar_target(raw_file, cohort_path, format = 'file'),
  tar_target(raw, load_cohort(raw_file)),
  tar_target(cleaned, clean_cohort(raw))
)
```

The `raw_file` target is what makes `targets` watch the CSV's contents. Without it, a lone `tar_target(raw, load_cohort(cohort_path))` would track only the value of the string `cohort_path`, and a corrected CSV at the same path would leave `raw` current. The cost is a three-target scaffold in the pattern of the five-target one shown in Exercise 3's solution below. The repository holds a lockfile but no Dockerfile, which places the compendium at L1. Adding the pipeline would not move it off that rung, since a pipeline pins no determinant. It would only make the one existing transform trustworthy to rebuild without having to recall that it exists.

> Check 13: what the pipeline would, and would not, fix
>
> Suppose the three-target pipeline above has been built once, and `cohort_raw.csv` is then corrected. What would `tar_outdated()` report, and would adding the pipeline change the compendium's rung on the L0 to L3 ladder?
> *The answer is on page 423.*

10.12. A language model inside the pipeline

The rest of this chapter treats every step as code the analyst controls. A step that sends text to a language model is different in kind. Examples include coding free-text clinical notes for smoking status and extracting a variable from a radiology report. In the terms of Chapter 3, the model is a dependency that no rung pins. It runs remotely, under an identifier the provider can update or retire, and it need not return the same output twice

Draft, November 13, 2026. Content may change.

for the same input. A lockfile cannot record it, and a container cannot hold it.

The remedy is to treat the responses as data rather than as computation. Four practices follow.

- **Cache responses as versioned derived data.** Store them in a committed file, one row per input, keyed by a hash of the input text, the prompt version, and the model identifier, and let the pipeline read that file. The `_targets/` store is not enough on its own, because it is absent from a fresh clone, and a rebuild there would query the model again.
- **Record the model and its settings.** Each row carries the model identifier the endpoint reported, the parameters sent, and the date of the query.
- **Version prompts as code.** A prompt is a file in the compendium, changed only by commit, and its hash enters the key.
- **Treat a re-query as a new measurement.** Responses obtained again, after a change of prompt or model or the loss of the originals, are a second rater's codes, not a re-execution. Report their agreement with the first set rather than presenting them as the same result.

The inputs fall under the restricted-data rule of the next section before any query is made. Clinical notes are among the most identifying records a study holds. The responses derived from them inherit the notes' restrictions, which decides where the cache may be deposited.

The fragment below is illustrative and abridged, and it is not run. The query itself lives in a separate script that the analyst runs by hand and whose new rows the analyst reviews before committing. `tar_make()` never contacts the endpoint: `join_responses()` matches each note to its row and stops with a list of the missing keys rather than filling them.

```
# _targets.R (illustrative and abridged; not run)
library(targets)
tar_source('R')

list(
  tar_target(notes_file,
   ↪  'analysis/data/raw_data/notes.csv',
```


Draft, November 13, 2026. Content may change.

```
            format = 'file'),
  tar_target(prompt_file,
   ↪  'analysis/prompts/smoking-v2.txt',
            format = 'file'),
  tar_target(responses_file,
            'analysis/data/derived_data/smoking-responses
             ↪  .csv',
            format = 'file'),
  tar_target(notes, read_notes(notes_file)),
  tar_target(
    smoking,
    join_responses(notes, responses_file, prompt_file,
                   model_id = '<model identifier>')
  )
)
```

A clean rebuild then reproduces the coded data exactly. The claim it supports is correspondingly narrow: the analysis is reproducible given the committed responses, not the model's coding of the notes.

10.13. Collaborating with an LLM

A language model is a fluent author of Makefiles and `targets` pipelines. Its characteristic failures are worth anticipating, because they bear directly on the one discipline this chapter rests on, the honesty of the dependency declarations.

! Restricted data

One rule comes first (habit 3). Never paste or upload protected health information, identifiable records, or data held under a data use agreement, nor outputs derived from them, into a model or a coding agent unless three conditions all hold. The institution has approved that endpoint for that class of data. The IRB protocol and the consent permit the processing. The data use agreement, or the provider's terms, permits it. If any of the three is unclear, the answer is no.

Draft, November 13, 2026. Content may change.

> The third condition is the one most often missed. Institutional approval of a tool does not override the terms under which a data provider released the data. NIH, for example, treats sharing controlled-access data through prompts with public generative AI tools, which it glosses as third-party tools, as a breach of the data use certification. It treats a model trained on such data as a data derivative, to be shared only with other approved users and not kept after the project closes (National Institutes of Health, 2025a).
> Describe the structure instead: the column names and types, or a small synthetic example generated from the data dictionary, never edited from real rows. A coding agent sees whatever files are in its working tree, so run it in a checkout or container where the real data is not mounted.
> An agent also acts, so confine it to a branch and keep merges, pushes, and deposits for yourself (Section 2.10.1).

For a pipeline, the structure to describe is the scripts, the files each reads and writes, and the column names and types of the data. A printed data frame or a log of row-level output is derived output under the rule. A coding agent asked to convert or run a pipeline will execute it and so read every input it touches.

Prompt. 'Write a Makefile for my analysis: I have a cleaning script, a modeling script, and a report.'

Watch for. The model cannot know the analyst's true dependencies, so check every rule for a guessed one. Look for an omitted prerequisite, such as a configuration file the modeling script reads or a helper script the cleaning script sources. Such a Makefile looks complete but fails to rebuild when the undeclared input changes. Check also for spaces where Make demands a literal tab to indent recipes, a classic error that produces a 'missing separator' message or, on a recipe line containing a colon, no error at all.

Verification. Check every rule against what each step actually reads and writes: list, for each script, its real inputs, and confirm each appears as a prerequisite. The decisive test is operational. Build once, then touch each declared input in turn and confirm the expected targets rebuild. Then touch an undeclared file the step secretly reads and confirm, as a negative

Draft, November 13, 2026. Content may change.

check, that the Makefile wrongly fails to rebuild, which reveals the missing prerequisite.

Prompt. 'Convert my scripts to a `targets` pipeline.'

Watch for. Check for `format = 'file'` used incorrectly, and for targets that read files directly inside the target expression without declaring them as file targets. `targets` then cannot see the dependency and will not rebuild when the file changes. Check also that the scripts were refactored into functions that return values, not left as scripts called for their side effects, which defeats the object-tracking that is the whole reason to prefer `targets` over Make. Finally, check for a report target that calls `rmarkdown::render()` inside a plain `tar_target()`, which never tracks the report's source.

Verification. Confirm that every external file the pipeline depends on is a `tar_target(..., format = 'file')` and that intermediate results are returned R objects, not files written by side effect. Then run `tar_make()`, edit one function body, and confirm with `tar_outdated()` and a second `tar_make()` that exactly the dependent targets, and no others, rebuild. Edit one sentence of the report's prose and confirm that `tar_outdated()` lists the report. If a change to a function fails to invalidate a downstream target, the dependency is not being tracked and the pipeline is lying about what is current.

A better prompt. 'Convert the three attached scripts to a `targets` pipeline. Their inputs and outputs are listed below, with column names and types but no data. Refactor each step into a function in `R/` that returns a value, declare every file the pipeline reads as a `format = 'file'` target, and render the report with `tar_render()`. Then list, for each function, the targets an edit to it should invalidate, so that I can check the list against `tar_outdated()`.'

Prompt. 'Add a step to my pipeline that sends each free-text note to `<tool name>` and codes it as current, former, or never smoker.'

Watch for. Check where the call sits. A call inside a target expression makes every rebuild of that target a fresh query, so a corrected upstream file or a `tar_destroy()` silently re-codes every note, unchanged ones included, as a new measurement. Check also that the model identifier, the parameters, and the prompt appear in committed files rather than in the code's defaults.

Draft, November 13, 2026. Content may change.

Verification. Restructure the step as Section 10.12 describes: responses in a committed file, keyed by input hash, prompt, and model identifier, and a pipeline that reads that file and never calls the endpoint. Then run `tar_make()` from a fresh clone with the network disabled and confirm that it completes. Before any note is sent, apply the restricted-data rule above. If a coding agent builds the step, the analyst runs the query script, reviews the new responses, and commits them; an agent that runs `tar_make()` or `tar_destroy()` should do so only in a checkout where the real notes are not mounted (Section 2.10.1).

10.14. Exercises

1. Take a multi-step analysis of your own that you currently run by hand, and draw its dependency graph on paper: a node for each step, an arrow for each 'consumes the output of' relation. Confirm the graph is acyclic. Then identify, for a change to one upstream input of your choosing, exactly which nodes are downstream of it and would need to rebuild, and compare that set to the steps you would, in practice, remember to re-run by hand.

2. Write a Makefile of three or four rules for the analysis of the previous exercise. Run `make`, then edit one intermediate script and run `make` again, and record which targets rebuilt and which were skipped. Explain the result in terms of the timestamp rule. Then `touch` an input without changing its contents, run `make` once more, and explain why Make rebuilds a target whose contents did not change.

3. Convert the same analysis to a `targets` pipeline. Refactor each step into a function in `R/`, write `_targets.R`, and confirm with `tar_visnetwork()` that the graph matches the one you drew in exercise 1. Run `tar_make()`, then change the body of one function and, before rebuilding, predict with `tar_outdated()` which targets will rebuild. Run `tar_make()` and check your prediction.

4. Demonstrate the difference between Make's and `targets`' staleness checks. Choose a helper function that one of your scripts sources from a file the Makefile of exercise 2 does not list as a prerequisite. Edit its body in a way that changes the result, and confirm that `targets`

Draft, November 13, 2026. Content may change.

invalidates the dependent targets. Explain, in one or two sentences, why the same edit could be missed by a Makefile that tracked only the script files' timestamps.

5. State, in the book's capture-versus-validation terms, what your pipeline has and has not added to the reproducibility of your project. Name the level on the ladder your project sits at, and identify what you would still need to do, a lockfile, a container, a verification against a baseline, to raise it, making explicit that none of those is supplied by the pipeline itself.

6. For each of the five archetypes, the data analysis, the simulation study, the manuscript, the R package, and the blog, write one or two sentences stating whether an analysis pipeline is central, modest, or absent, and why. For the archetype in which it is absent, name the mechanism that plays the corresponding role instead.

7. Run `cohort <- clean_cohort(load_cohort('analysis/data/raw_data/cohort_raw.csv'))` and report `nrow(cohort)`. Then write the three-target `_targets.R` fragment that wraps `load_cohort()` and `clean_cohort()` on this path, following the `raw_file` and `raw` split of Exercise 3's solution, and state what `tar_outdated()` would report immediately after the first `tar_make()` and after a second one with no file changed.

8. This exercise returns to the reproducibility ladder of Chapter 3 (Chapter 3). A project has a lockfile, a container, and a `targets` pipeline that runs green on every push, but no test compares a computed quantity with a value recorded earlier. State the single rung of the L0 to L3 ladder this project occupies, and name the one artifact, absent here, whose addition would be needed to reach the next rung.

9. Take the illustrative `_targets.R` fragment of the section on a language model inside the pipeline (Section 10.12). For each of four situations, state which targets `tar_outdated()` would list and what the next `tar_make()` would do: (a) a fresh clone, with the response file committed and no `_targets/` store;

   (b) a corrected spelling in one note; (c) a new version of the prompt file; (d) a new model identifier in `_targets.R`. Then say which

Draft, November 13, 2026. Content may change.

situations call for a new query, and why its result is a new measurement rather than a re-execution.

10.15. Solutions

Attempt an exercise before expanding its solution. Every exercise in this chapter is performed on an analysis of the reader's own, so the outputs vary; the boxes record what should happen and the reasoning that makes it expected.

Exercise 1

No single correct answer. What a good answer establishes is the gap between the downstream set and the remembered set.

On acyclicity: the graph must be acyclic because a cycle would mean a step consumes, directly or indirectly, its own output. There would then be no order in which to run the steps. Readers occasionally draw what looks like a cycle and find on inspection that they have drawn two different versions of one file as a single node. Splitting the node resolves it, and the resolution is itself informative, since it names an artifact that is overwritten in place.

The instructive part is the comparison. The downstream set of a change to a raw input is, in most analyses, everything: cleaning, modeling, every figure, every table, the report. The set most analysts re-run by hand is smaller, and it is smaller in a characteristic way. People reliably re-run the step they just edited and the thing they are looking at. They reliably forget the intermediate results that nothing on screen depends on, along with any figure that is not in the document currently open.

The downstream set is a property of the graph, and a person holding the graph in their head computes it correctly only for small graphs and recent memory. A pipeline tool computes it from the graph every time.

Draft, November 13, 2026. Content may change.

Exercise 2

Editing an intermediate script rebuilds that script's target and everything downstream of it, and skips everything upstream and everything on a parallel branch. The timestamp rule explains it directly. Make rebuilds a target whose modification time is older than any of its declared prerequisites. Rebuilding a target advances its own timestamp, so staleness propagates forward one link at a time.

Why `touch` triggers a rebuild. Because Make does not compare contents at all. It compares modification times, and `touch` advances the modification time of the input without changing a byte of it. Every target depending on that input is now older than one of its prerequisites, so Make rebuilds it, and the rebuild propagates downstream exactly as a real edit would.

Make's guarantee is therefore conservative in one direction and unreliable in the other. It will happily rebuild things that did not need rebuilding, which costs time and nothing else. It will also miss a change that leaves timestamps as it found them, which costs correctness. The first behavior is the one this exercise demonstrates; the second is the subject of exercise 4.

Exercise 3

The refactoring is the substance of the exercise. Each Make rule becomes a function in `R/`, and `_targets.R` declares the targets that call them:

Draft, November 13, 2026. Content may change.

```
list(
  tar_target(
    raw_file,
    'data/raw_data.csv',
    format = 'file'
  ),
  tar_target(raw, read_raw(raw_file)),
  tar_target(cleaned, clean_data(raw)),
  tar_target(model, fit_model(cleaned)),
  tar_target(figures, make_figures(model, cleaned))
)
```

`format = 'file'` belongs on `raw_file`, not on `raw`: it tells `targets` to hash the file at the path a target returns, and only `raw_file` returns a path (the string literal above). `raw` returns the data frame that `read_raw()` builds from that path. Declaring it `format = 'file'` therefore makes `targets` try to hash a data frame as if it were a file path and error. The two-target split, one target for the tracked file and one for the object read from it, mirrors the pattern used earlier in this chapter for the same reason.

`tar_visnetwork()` should show the same nodes and arrows as the hand drawing. A mismatch is informative and usually means the hand drawing missed a dependency, since `targets` derives the graph from the code rather than from memory.

On the prediction: after editing a function body, `tar_outdated()` names that function's target and everything downstream, and `tar_make()` rebuilds exactly that set. Readers who predicted correctly have understood the propagation rule; readers who predicted too small a set usually forgot that invalidation is transitive.

The prediction is easy here and was hard in exercise 1, on the same analysis, because `targets` made the graph visible: the reader now reads the dependency structure instead of recalling it.

Draft, November 13, 2026. Content may change.

Exercise 4

Editing a function body in `R/` changes no file that a Makefile would ordinarily list as a prerequisite of the model target. The Makefile lists the *script* that fits the model, not the helper file the script sources. `targets` invalidates the dependent targets anyway, and `tar_outdated()` names them before the rebuild.

Why a Makefile could miss the equivalent edit. Because it tracks the timestamps of the files it was told to track, and nothing told it about the helper. If `fit_model.R` sources `R/helpers.R` and the rule declares only `fit_model.R` as a prerequisite, then editing `helpers.R` advances a timestamp Make never looks at. Every downstream target remains, in Make's view, up to date. The analysis then reports results computed by the old function body, with no error and no indication that anything is stale.

`targets` avoids this because it hashes the *bodies of the functions each target calls*, and the R objects those targets produce, rather than the modification times of a declared file list. Because the dependency is discovered from the code, the analyst cannot declare it incompletely. The fair statement of the comparison is that a correctly and completely written Makefile would catch this too. The difference is that `targets` makes completeness the default and Make makes it the analyst's responsibility, and the failure mode of an incomplete Makefile is silence.

Exercise 5

What the pipeline added. In the book's terms, almost nothing on the capture side and a good deal on the validation side. The pipeline pins no layer of the determinant stack: not the packages, not the environment, not the R version. What it adds is a guarantee that the outputs are *current with respect to the inputs*, which is a form of internal consistency the analysis did not previously have. It also adds an explicit, machine-readable statement of the analysis structure, which is documentation of a kind that cannot drift.

The level. The pipeline alone leaves the project where it was, which for most readers is L0. This is the part of the exercise most likely to be answered wrong, because a `targets` pipeline feels like a large step

Draft, November 13, 2026. Content may change.

forward and is, on an axis the ladder does not grade.
What would still be needed. A lockfile to reach L1, a container to reach L2, and verification against a recorded baseline to reach L3. None of the three is supplied by the pipeline, and the pipeline does not make any of them easier or harder to add; they are orthogonal.
There is one honest qualification in the pipeline's favor. Because it computes rather than remembers the downstream set, it removes a class of error, stale outputs, that would otherwise defeat an L3 verification. Such a failure would arise for reasons having nothing to do with the environment. It does not raise the level, and it makes the level easier to keep.

Exercise 6

Data analysis: central. The archetype is a chain of steps producing derived data and results, which is exactly what a dependency graph describes.
Simulation study: central. Replicates are expensive and independent, so recomputing only what is stale matters more here than anywhere, and `targets` branching maps onto the replicate structure directly.
Manuscript: modest. There is usually a pipeline behind the manuscript, but the render itself is a single terminal step and the literate document does much of the orchestration. The pipeline earns its place when the analysis feeding the document is expensive enough that re-running it on every render is intolerable. That is the same judgment the previous chapter framed as deciding what the document computes at build time.
Blog: modest to absent. Each post is typically small and self-contained, and the site generator already rebuilds changed posts. A pipeline adds machinery for little gain.
R package: absent. There is no chain of data transformations to schedule; the deliverable is a set of functions. **The mechanism that plays the corresponding role is `R CMD check` together with the test suite**, which performs the analogous job of determining what must be re-verified and doing it. The analogy is closer than it looks: both answer 'given what changed, what must be recomputed', and

Draft, November 13, 2026. Content may change.

both refuse to accept a stale result.

Exercise 7

The solution for this exercise is in the instructor guide, which is distributed separately to course instructors. Self-study readers who want to check their answer can compare the autogradable value and the `_targets.R` fragment against those reported in the chapter's own 'The cohort, continued' section.

Exercise 8

The project is at **L2, pinned environment**, and no further. The lockfile and the container pin the packages and the environment, and a `targets` pipeline that runs green on every push demonstrates that the analysis re-executes completely and reliably in that pinned environment. None of this is a baseline comparison. A green pipeline run shows that the code ran without error, not that any reported quantity still matches a value recorded earlier. This is exactly the L2-versus-L3 distinction Chapter 3 draws for a test suite with no baseline test.

The missing artifact for L3 is a baseline test, a test that recomputes a reported quantity and compares it with a value committed to the repository within a stated tolerance. Making the pipeline run the step that triggers that test inside continuous integration, as Chapter 12 describes, is what turns the green run from a statement about re-execution into evidence for L3. This chapter's worked example stops short of that step: it has no baseline test.

Exercise 9

In (a) there is no store, so every target is outdated. `tar_make()` rebuilds them all from the committed files, and `smoking` is reproduced exactly, because `join_responses()` reads the committed responses and the endpoint is never contacted. This is the case the design exists for.

Draft, November 13, 2026. Content may change.

In (b) the hash of `notes_file` changes, so `notes` and `smoking` are outdated. The corrected note has new text, and therefore a new key with no row in the response file, so `join_responses()` stops and names that one note. In (c) the hash of `prompt_file` changes, so `smoking` is outdated; because the prompt's hash enters every key, every note is now missing and the stop names them all. In (d) the command of `smoking` changes, which outdates it, and again every key is missing.

Situation (a) needs no query. Situation (b) needs one query, for the corrected note, and (c) and (d) need a query for every note. You run those queries with the separate script, review the new rows, and commit them beside the old ones. Their result is a new measurement because the model, as reached on a later date, need not return what it returned before, even under an unchanged identifier, and in (c) and (d) the instrument itself has changed. Where old and new codes exist for the same note, report their agreement, as you would for a second rater.

Answers to the checks

Check 1. *from earlier chapters* (page 381)

The layer that changed after the `survival` update is the package-versions layer. The new `renv.lock` records the newer release, and `renv::restore()` reinstalls exactly the versions the lockfile records (Chapter 6). The page cannot tell you whether the estimate moved. Under `freeze: auto`, a project render compares only with the document's own source, so an updated package leaves the frozen results in `_freeze/` in place under a fresh build date (Section 9.8). Rendering that single document always executes its code. For a deposited analysis, render in the pinned environment with freeze off, or after deleting `_freeze/` and the cache directories (Chapter 9).

Check 2. *capture or validation, again* (page 382)

No. The pipeline pins no layer of the determinant stack: the package versions, the operating system, and the R runtime are exactly as pinned, or

Draft, November 13, 2026. Content may change.

as unpinned, after `_targets.R` is added as before. What changed is that the analysis-code and input-data layers are now checked for currency on every run rather than trusted to memory. That is a validation feature, like continuous integration or a baseline test, and it leaves the project's rung on the L0 to L3 ladder exactly where it was.

Check 3. *naming the failure* (page 384)

It is staleness. The correction to the raw data left the figures, which depend on the model fitted from the old data, out of date. Re-running only the modeling script did not propagate the change the one additional step further. The stale figure gives no warning because a stale PNG is a perfectly valid image file. Nothing about its bytes records what it was computed from, so neither the file system nor a glance at the document can distinguish it from a current one.

Check 4. *whose judgment, again* (page 385)

A green run settles none of the three. It says nothing about whether the targets were cut at a sensible granularity, since a pipeline with one enormous target can run green and still cache nothing useful. It says nothing about whether the declared dependencies are complete, since an undeclared input simply never triggers a rebuild and the run stays green regardless. And it says nothing about the level reached, since re-execution and pinning are different questions entirely.

Check 5. *reading the measured costs* (page 391)

The analyst pays for the bootstrap, the most expensive step, even though a color change touches nothing the bootstrap consumes; a selective rebuild would skip it entirely. The absolute timings are properties of the one machine that happened to build this chapter. They will differ, often by a large factor, on another machine or even on a repeated run of the same one. The ratio of the bootstrap to the fit is not stable either, because the fit's few milliseconds sit near the timer's resolution. What survives is the ordering: the bootstrap is by far the largest step, and it takes nearly all of the total. The table is read for which step dominates, not for any one number.

Check 6. *what Make rebuilds* (page 394)

Draft, November 13, 2026. Content may change.

Editing `make_figures.R` advances its modification time past that of `figures.rds`, so `figures.rds` is now older than one of its prerequisites and Make rebuilds it. Rebuilding `figures.rds` advances its timestamp past that of `report.html`, so `report.html` is rebuilt too. Make skips `cleaned.rds` and `model.rds`. Neither the raw data, the cleaning script, the fitting script, nor the cleaned data changed. Each of those targets therefore remains newer than all of its prerequisites and is up to date. The expensive model fit is not repeated.

Check 7. *what `targets` tracks that Make does not* (page 397)

Make decides staleness by comparing file modification times against declared prerequisites, so a change that is not reflected in a listed prerequisite's timestamp is invisible to it. `targets` instead hashes the contents of its inputs and also the bodies of the functions each target calls and the R expression in the target itself. Editing the function body changes its hash, which invalidates every target that calls it and everything downstream, so the rebuild propagates correctly. The two things `targets` hashes that Make does not are the R function definitions and the in-memory R objects (rather than only on-disk files). Together these let it track code changes and object changes that a timestamp comparison cannot see.

Check 8. *the composed claim* (page 399)

What has been shown is that the pipeline's definition of 'rebuild' runs to completion on an external, clean machine on every change, which is strong evidence of re-execution. It is not, by itself, evidence of any rung. CI supplies the clean trigger and the pipeline supplies the complete recipe, but neither pins a package, fixes an environment, or compares an output with a recorded baseline. The rung actually reached depends entirely on whether a lockfile, a container, and a baseline comparison also exist. 'It runs on every push' answers a question about re-execution, not about level.

Check 9. *an archetype with no pipeline* (page 400)

A package has no chain of data transforms producing a report, so there is no dependency graph of the kind a pipeline tool rebuilds selectively. Its deliverable is a library of functions, not a sequence of derived outputs. `R CMD check`, together with the test suite, plays the corresponding role. It determines what must be re-verified given what changed and refuses to

Draft, November 13, 2026. Content may change.

accept a stale result. This is the same job a pipeline does for a data analysis, performed by the package toolchain instead.

Check 10. *choosing among the five tools* (page 402)

Snakemake or Nextflow fits. Both are built for exactly this setting, many languages, many steps, and execution spread across a cluster or the cloud, with native support for job schedulers. Make has none, and `targets` reaches a scheduler only through backends such as `clustermq`. `targets` would be the wrong reach because it is R-native. Its dependency inference works by hashing R function bodies and R objects. That has nothing to track in a pipeline whose steps are Python scripts and shell commands calling external tools. The pipeline idea is the same; the tool must match the language and the scale.

Check 11. *the two-target split* (page 403)

`format = 'file'` tells `targets` to hash whatever the target's own return value points to. Only a target that returns a path, such as the string `'data/raw_data.csv'`, can carry that option. `read_raw(raw_file)` returns a data frame, not a path. Declaring it `format = 'file'` would make `targets` try to hash a data frame as though it were a file path and fail. Splitting the tracked file from the object read from it is what lets `targets` watch the raw file's contents while still storing the loaded data frame as an ordinary R object.

Check 12. *complete the* `_targets.R` (page 404)

The four missing lines of the worked example's `_targets.R` are these.

(1) `format = 'file'`. It belongs on `raw_file` because that target returns a path; on a target that returns a data frame, `targets` would try to hash the data frame as a file path and fail.

(2) `tar_target(raw_data, read_raw(raw_file)),`. Naming `raw_file` in the expression is what declares the dependency. A plausible `read_raw('data/raw_data.csv')` reads the same file, but `targets` was never told about it and cannot see a change to it.

(3) `tar_target(figures, make_figures(model, cleaned)),`. Naming both targets draws the edges from `model` and from `cleaned`;

Draft, November 13, 2026. Content may change.

a figure function that fetched the data some other way would hide the second edge from the graph.

(4) `tar_render(report, 'report.Rmd')`. The tempting `tar_target(report, render_report(model, figures), format = 'file')` tracks the model, the figures, and the rendered HTML but not the source, so editing the report's prose leaves `tar_outdated()` empty.

Check 13. *what the pipeline would, and would not, fix* (page 406)

`tar_outdated()` would report all three targets, `raw_file`, `raw`, and `cleaned`, as out of date. The file target `raw_file` hashes the corrected CSV's new contents, `raw` depends on `raw_file`, and `cleaned` depends on `raw`. No, the rung would not change. The pipeline would still pin no package version, no R runtime, and no operating system. The compendium would therefore remain at L1, the rung set by what the repository holds (a lockfile but no Dockerfile), whatever `tar_outdated()` reports. The pipeline only guarantees that a rebuild, once triggered, rebuilds the right things.

10.16. Cheat sheet

Command or rule	What it means or checks
Rule: `target: prerequisites`	Declares a Make target and the files it depends on
Make recipe line	Indented with a literal tab, never spaces (Pitfall above)
`make`	Rebuilds every target older than one of its prerequisites
`touch file`	Advances a file's timestamp without changing its contents; triggers Make, not `targets`
`tar_target(x, expr)`	Declares a `targets` target; the dependency graph is inferred from `expr`

Draft, November 13, 2026. Content may change.

Command or rule	What it means or checks
`tar_target(x, path, format = 'file')`	Tracks `x` as a file, by content hash, not by R object
`tar_make()`	Builds every out-of-date target, in order, and skips the rest
`tar_outdated()`	Lists what a `tar_make()` would rebuild, without running anything
`tar_visnetwork()`	Draws the dependency graph, colored by up to date or out of date
`tar_render()` / `tar_quarto()`	Tracks a report's own source as well as the targets it reads
Capture vs. validation	A pipeline captures no determinant; it validates that an output is current
Honest claim	'The pipeline re-executes reliably' is not 'the analysis is reproducible'; name the rung separately
LLM response cache	A committed file of model responses keyed by input hash, prompt version, and model identifier; the pipeline reads it and never queries (Section 10.12)

10.17. Further reading

The definitive source on `targets` is its documentation and the short paper that introduces it, Landau (2021). That paper states the 'function-oriented, Make-like' design directly and is the natural starting point for an R analyst. Its predecessor Landau (2018) documents `drake`, which a reader will still meet in existing projects and which shares the same conceptual model. The manual-steps critique that motivates the whole chapter is put crisply by Sandve et al. (2013). Their rules on automating the path from raw data to result are the pipeline idea stated as a rule of conduct. For the language-agnostic and large-scale tools, Köster & Rahmann (2012) introduces Snakemake and Di Tommaso et al. (2017) introduces Nextflow. Both repay reading by anyone whose work will move into bioinformatics or onto a cluster. The `workflowr` approach to binding rendered results

Draft, November 13, 2026. Content may change.

to version-control commits is described in Blischak et al. (2019). GNU Make itself is documented in its own extensive manual. A reader will consult it for the pattern rules and automatic variables that this chapter's minimal treatment omitted. For data-analysis uses specifically, the broader software-carpentry literature on reproducible workflows, of a piece with Wilson et al. (2014) and Wilson et al. (2017), situates Make in the analyst's toolkit rather than the programmer's.

Draft, November 13, 2026. Content may change.

11. Testing and Code Quality

Program testing can be used to show the presence of bugs, but never to show their absence!

Edsger W. Dijkstra, *Notes on Structured Programming* (1972)

A function that silently sums instead of averaging will pass every rung of the reproducibility ladder and still be wrong on every machine that runs it. By the end of this chapter you will be able to say, of any computational result, whether it has merely been reproduced or actually checked. To make that distinction in your own analyses, the chapter develops the four kinds of test, among them the baseline test that Table 11.2 demonstrates on the book's own running cohort.

Before you start

Reading time. About 65 minutes.
Prerequisites. Chapter 3, for the L0-L3 ladder and the capture-versus-validation vocabulary this chapter relies on throughout; Chapter 5, for the `inst/tinytest/` layout and the R-package archetype. Statistically, no more than comfort reading a coefficient table from a fitted model and a percentile bootstrap interval; no new statistics are introduced. R knowledge assumed: writing and calling a function, vectors, and basic control flow.
Objectives. By the end of this chapter, can you:

- state the reproducibility paradox, and explain why reproducibility machinery cannot, by itself, detect a wrong answer?
- place testing in the validation register rather than the capture register, and say what that does and does not change about an analysis's position on the L0-L3 ladder?
- write a unit test, an integration test, a data-validation test, and

a baseline (reproducibility) test in `tinytest`, naming the oracle behind each?
- tell behavioral testing from implementation testing, and say why the second kind breaks under an honest refactor?
- explain why `lintr`, `styler`, and a pre-commit hook belong to the validation register?

Outline. We begin with the analyst's irreplaceable judgments, state the paradox precisely, and treat a pre-specified analysis plan as an oracle. We next introduce `tinytest`'s bare-expectation style, work the four kinds of test on the cohort and the retinopathy trial, and separate Monte Carlo error from a change of sampler. We then distinguish behavior from implementation, treat linting, styling, and the pre-commit gate, and set out what 'validation' means in regulated work, before turning to how one analyst reviews another's code. The later sections show how the emphasis shifts across archetypes, give a worked example of the paradox itself, discuss collaborating with an LLM, and apply all four kinds of test to the running cohort. The chapter closes with exercises, a cheat sheet, and further reading.

Check 1: from earlier chapters

Three failures reach you in one week. (a) A collaborator runs a project that has no `renv.lock`, installs newer versions of its packages than the author used, and a summary table changes. (b) A colleague's `renv::restore()` of a complete lockfile stops because a spatial package cannot find GDAL. (c) A figure still shows last month's counts after the raw extract was corrected, although the report rendered without error. For each, name the determinant layer that failed and the earlier chapter's remedy.
The answer is on page 478.

11.1. Where this sits on the stack

Testing is this book's clearest case of pure validation. It is also, among the chapters so far, one that captures nothing at all. That is, a test suite adds

Draft, November 13, 2026. Content may change.

no package to the lockfile and no layer to the image. Figure 11.1 marks that asymmetry on the shared determinant stack. The two layers this chapter writes code against, the analysis code and the input data, are validated here, not captured. The surrounding chapters captured them already, and this chapter only confirms they still behave.

```
plot_determinant_stack(
  validates = c('Analysis code', 'Input data'),
  title = 'Testing validates code and data; it captures
   ↪  neither')
```

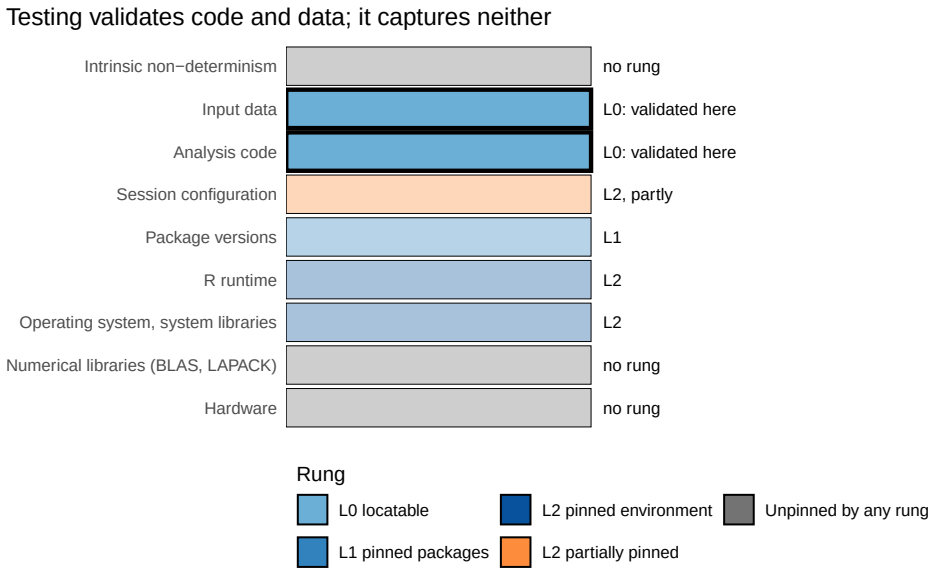


Figure 11.1.: The determinant stack of Chapter 3, with analysis code and input data marked as validated rather than captured. This chapter pins no new layer; a unit test, a data-validation test, and a baseline test each confirm that the code and the data already captured by earlier chapters still produce the answer they are supposed to, which is the capture-versus-validation distinction the chapter develops.

Draft, November 13, 2026. Content may change.

11.2. Orientation

Consider a registry study in which an analyst has written a small function to compute a mean biomarker value for each treatment arm. The function carries a simple bug: it sums the values but forgets to divide by their count. The analysis runs without error. It produces a table, a figure, a p-value. Sent to a colleague, cloned onto a second machine, restored from a lockfile inside a pinned container, it runs again and produces the identical table, the identical figure, the identical p-value. Every rung of the ladder of Chapter 3 is satisfied: the source is locatable, the packages are pinned, the environment is pinned. The result is perfectly reproducible, and it is wrong.

This is the reproducibility paradox, and it is the pivot of this chapter. A buggy function that returns the wrong answer returns the *same* wrong answer everywhere, on every machine, at every later date. Reproducibility machinery, faithfully applied, does not detect the error; it propagates it with mechanical fidelity. The lesson, pressed by Leek and Peng in their aptly titled note that reproducible research can still be wrong (Leek & Peng, 2015), is that reproducibility is a floor and not a ceiling. It guarantees that a result can be regenerated. It says nothing whatever about whether the result is correct.

The remedy is testing. But why does testing belong in a book about reproducibility, rather than in a separate book about software engineering? The reason is precisely the paradox. Pinning a package version, building a container, restoring a lockfile: each of these *captures* a determinant of the computation, and none of them examines whether the computation is right. Testing captures nothing. A test adds no package to the lockfile and no library to the image. What a test does is *confirm* that a function, a pipeline, or a dataset behaves as the analyst intended. It belongs, in the vocabulary this book has used since Chapter 1, to the validation register rather than the capture register, alongside the continuous integration and output verification of Chapter 12. Capture fixes the inputs; validation checks that the fixed inputs still produce the answer they are supposed to. A reproducible analysis without tests is trusted; a reproducible analysis with tests is checked.

There is a further and more constructive link to the ladder. The highest

Draft, November 13, 2026. Content may change.

rung, L3 verified, is reached not by capturing more but by *confirming* that a recorded computational output regenerates. That confirmation is itself a test, of a particular kind we shall meet below as the baseline test, so the topmost level is defined by a test.

Check 2: the paradox

A colleague reports that their analysis reaches L2 pinned environment and argues this means the reported hazard ratio must be correct. What is wrong with the argument?
The answer is on page 478.

11.3. The analyst's contribution

Testing tooling runs the checks, tallies the passes and failures, and reports them. It does not originate any of the judgments that give a test its value. Three of them underlie everything that follows.

1. **The analyst decides what 'correct' means.** A test compares a computed value against an expected one, and the expected one has to come from somewhere outside the code under test. This source of independent truth is called an oracle. Examples are a mean of one, two, and three that the analyst knows to be two; a count of rows that a data dictionary fixes at some known number; a coefficient that a textbook formula supplies. No tool knows the right answer. The analyst supplies it, and a test is only as trustworthy as the oracle behind it.

2. **The analyst chooses which behaviors to pin.** A function has an infinity of properties, most of them incidental to its purpose. The analyst decides which few constitute the function's contract, the behaviors a future change must not break, and writes tests for those. This is the distinction, developed below, between testing behavior and testing implementation.

3. **The analyst encodes the domain constraints.** Consider the rules that an age lies between zero and one hundred and twenty, that a treatment arm is one of a fixed set of labels, and that a probability

Draft, November 13, 2026. Content may change.

The vocabulary of this chapter

- **Unit test.** A focused check that one small piece of code, typically a single function, behaves as intended on known inputs.
- **Integration test.** A check that several components behave correctly when composed into a pipeline, from raw input to final result.
- **Data-validation test.** A check on the data itself, not the code: that a value lies in a plausible range, that no impossible ages appear, that the expected columns are present.
- **Reproducibility test.** A check that a recorded computational result regenerates. The form that can fail is the *baseline test*, which compares a fresh result with a value recorded in an earlier session and committed, within a stated tolerance. Two runs in one session do not qualify.
- **Oracle.** The source of independent truth a test compares against; the known correct answer.
- **`tinytest`.** A lightweight, zero-dependency R testing framework in which a test file is a plain script of bare, top-level expectation calls (Loo, 2020).
- **Expectation.** A single assertion, such as `expect_equal( actual, expected)`, that passes or fails.
- **Linting.** Static analysis of source code for stylistic and likely-error patterns, performed by `lintr`, without running the code.
- **Styling.** Automatic reformatting of source code to a consistent layout, performed by `styler`.
- **Pre-commit hook.** A script that git runs automatically before a commit is recorded, used here to run the linter and the styler as a gate.

Draft, November 13, 2026. Content may change.

Check 3: three judgments

Name the three judgments that a testing tool cannot make on its own, and that fall to the analyst instead.
The answer is on page 479.

11.4. The reproducibility paradox, stated precisely

A computational result depends on the code, the data, and the stack of determinants beneath them that Chapter 3 enumerated. The capture tools of Part II fix those determinants, so that the mapping from input to output is held constant across machines and across time. But holding a mapping constant says nothing about whether the mapping is the intended one. If the code encodes the function 'sum' where the analyst meant the function 'mean', then capture faithfully preserves 'sum', and every reproduction returns the sum. The error is not in the reproduction; the reproduction is flawless. The error is in the mapping, and the only instrument that examines the mapping is a test.

Two consequences follow, and they cut in opposite directions. On the one hand, a result that fails to reproduce is thereby shown to be untrustworthy, which is a genuine service; irreproducibility is a detectable defect. On the other hand, a result that does reproduce has been shown only to be stable, not to be right. Stability is easily mistaken for correctness by an analyst who has watched the same number appear three times. Testing exists to break the false inference from 'it came out the same' to 'it came out correct'.

! The honest level

This chapter's own practice reaches a baseline test on one bootstrap bound (Table 11.2) and a short, illustrative suite of unit, integration, and data-validation checks on hand-picked examples. It establishes that those particular results regenerate and that those particular functions behave as intended. It says nothing whatsoever about any function, dataset, or computation this chapter did not choose to test.

Draft, November 13, 2026. Content may change.

Check 4: capture, faithfully

If the code encodes 'sum' where the analyst meant 'mean', what does capturing the environment do to that error, and what does it not do? *The answer is on page 479.*

11.5. The analysis plan as an oracle

The oracle introduced above sits at the level of a single function: a hand-computed mean, a row count a data dictionary fixes. The same structure scales up to the level of the whole analysis's primary result, and doing so closes a gap the paradox otherwise leaves open.

A **pre-specified analysis plan** states in advance the primary endpoint, the model, the covariates, and the comparison that will answer the study's main question. It is also called a statistical analysis plan, or, when it is registered publicly before the data are seen, a pre-registration. Written and dated before the analyst has looked at the outcome data, it is computed independently of any code's output in exactly the sense the Vocabulary box above requires of an oracle. That is, it specifies what the correct analysis looks like, not by running anything, but by committing in writing to a choice that the data cannot later be allowed to revise.

A function-level oracle, such as the hand-computed mean described above, catches a coding error: the function returns the wrong number for a known input. It says nothing about whether the analysis itself asks the right question. Consider an analysis built by trying several models, several covariate sets, or several endpoints and reporting the one that reached significance. It can be bug-free in every function and pass every unit test. It can also still reproduce exactly on a second machine, because reproduction only checks that the same code returns the same output. The reproducibility paradox stated above applies again, one level up. A fully reproducible analysis can still be the wrong analysis, chosen with the benefit of having already seen the answer. A pre-specified plan is the oracle that guards against this failure, because it fixes the intended analysis before the data could have shaped it.

Draft, November 13, 2026. Content may change.

The two oracles differ in what independence they supply. A function's oracle is independent of the code that computes the value; a pre-specified plan is independent of the data that will later be analyzed. Both are necessary, and neither substitutes for the other. A bug-free implementation of an unregistered, data-driven analysis is still untrustworthy in the way the plan would have prevented. Conversely, a faithful implementation of a pre-specified plan can still contain a coding error that only a function-level oracle would catch.

In a research compendium, a pre-specified plan is a dated document, committed to the repository (or deposited externally, as on the Open Science Framework) before the raw data arrive, or before they are unblinded. Checking an analysis against it is not an `expect_equal()` call but a review step: do the fitted model's formula, the stated primary comparison, and the decision rule actually implemented match what the plan specified? Where the match can be stated mechanically, for example that `fit_model()` fits the formula the plan names, a test can assert it directly. Such a test compares the model's formula (as text, or as a parsed call) against a string recorded from the plan rather than against anything the analysis itself computed.

Check 5: two oracles

A colleague's bootstrap confidence interval regenerates bit for bit on a second machine, and every unit test in the package passes. Is this enough to trust the reported result as the study's primary finding? What additional check, and what additional oracle, would you ask for?
The answer is on page 479.

11.6. tinytest: bare expectations

The framework this book uses adopts `tinytest` (Loo, 2020) as its testing tool. The choice rewards a moment's explanation because it shapes how the tests look. `tinytest` is lightweight and carries no dependencies of its own, which matters in a reproducible compendium. Every additional test dependency enlarges the lockfile and the container image, and a testing framework that dragged in a dozen packages would tax the very machinery it

is meant to protect. The more visible consequence is stylistic. A `tinytest` test file is an ordinary R script containing bare, top-level expectation calls, with no wrapping function around them.

A reader who has met `testthat` will notice the difference at once. Where `testthat` nests each assertion inside a `test_that('description', { ... })` block, `tinytest` writes the assertion directly:

```
# inst/tinytest/test-calculate_mean.R
expect_equal(calculate_mean(c(1, 2, 3)), 2)
expect_equal(calculate_mean(5), 5)
expect_true(is.nan(calculate_mean(numeric(0))))
```

Each line is a complete test. The file is also a runnable script, so during development an analyst can source it directly and watch the expectations evaluate. The available expectations cover the common cases (`expect_equal()`, `expect_true()`, `expect_false()`, `expect_identical()`, `expect_error()`, `expect_inherits()`, and a few others). A compendium's tests are collected in `inst/tinytest/` and driven by a small `tests/tinytest.R` runner that `R CMD check` invokes. `tinytest` runs each file with the test directory as its working directory. A test that reads a project file from the root therefore reaches it through `../../`, as the listings below do. We follow the convention throughout: bare top-level expectations, one concern per file, named `test-<thing>.R`.

> Check 6: reading a bare file
>
> A `tinytest` test file contains no `test_that()` call and no function definition around its expectations. What does that tell you about how to run one line of it during development?
> *The answer is on page 479.*

11.7. The four kinds of test

A research compendium wants four kinds of test, and they form a progression from the smallest unit of code to the reproduction of a whole result. We

Draft, November 13, 2026. Content may change.

take them in turn, illustrating each on synthetic data of the sort a health study might produce.

11.7.1. Unit tests of functions

A unit test checks that one function behaves as intended on inputs whose correct outputs are known independently. The mean function of the orientation, here called `calculate_mean`, is the canonical case. Its contract is that it returns the arithmetic mean, and the oracle is arithmetic done by hand:

```
# inst/tinytest/test-calculate_mean.R
expect_equal(calculate_mean(c(2, 4, 6)), 4)
expect_equal(calculate_mean(c(-1, 0, 1)), 0)
expect_true(is.nan(calculate_mean(numeric(0))))
expect_equal(calculate_mean(c(1, NA, 3),
                            na.rm = TRUE), 2)
```

The first two lines pin the ordinary behavior against hand-computed answers. The third and fourth pin the edge cases, the empty vector and the missing value, which are exactly the inputs on which an untested function tends to misbehave. A unit test suite is, in effect, a written record of what the analyst believes each function promises.

11.7.2. Integration tests of the pipeline

A unit test exercises a function in isolation, but a health analysis is rarely one function. It is a pipeline, in which raw data is loaded, cleaned, modeled, and summarized, and the components can be individually correct yet compose incorrectly. An integration test runs the pipeline end to end and checks the result:

```
# abridged from inst/tinytest/test-cohort.R
raw <- load_cohort('../../analysis/data/raw_data/cohort_ra⌋
 ↪  w.csv')
cohort <- clean_cohort(raw)
```


Draft, November 13, 2026. Content may change.

```
fit <- fit_model(cohort)
summary_tbl <- summarise_fit(fit)

expect_true(is.numeric(coef(fit)))
expect_true(all(c('term', 'estimate') %in%
                names(summary_tbl)))
expect_equal(nrow(summary_tbl), 3L)
```

The first expectation tests the contract that `coef(fit)` returns numeric coefficients, which is what `summarise_fit()` needs to proceed. Testing `expect_inherits(fit, 'lm')` would also pass here, because `glm` inherits from `lm`, but it would break the moment the function was upgraded to `fit_cox()` returning a `coxph`, which does not. Testing the interface ('does `coef()` return numeric values?') survives that migration; testing the class does not. The book's own compendium does not perform this migration: `fit_cox()` was added beside `fit_model()` rather than replacing it, and its tests sit in a file of their own, `inst/tinytest/test-fit-cox.R`.

The integration test does not re-check what the unit tests already cover. Instead it checks the seams: that the output of `clean_cohort` is a valid input to `fit_model`, and that the model summary has the expected shape. A pipeline that passes its unit tests can still fail here, if a column is renamed in one step and read under its old name in the next.

11.7.3. Data-validation tests

The third kind of test turns its attention from the code to the data. Much of what goes wrong in a health analysis is not a coding error but a data defect. Examples are an age of two hundred, a negative laboratory value, a treatment label misspelled, or a column silently dropped by an upstream export. A data-validation test encodes the analyst's domain knowledge as executable constraints and fails when the data violates them.

Before writing constraints, we look first at what we are constraining. Table 11.1 tabulates the retinopathy trial introduced in the conventions. It is included here for a reason beyond illustration: every cell in it can be counted by hand from the data, which is precisely the property an oracle

Draft, November 13, 2026. Content may change.

needs. A test asserting that the untreated arm contains 197 eyes, written as that literal number, is checkable by someone who does not trust the function that produced it. A test asserting that it contains `nrow(subset(d, trt == 0))` eyes is not a test at all; it is the function agreeing with itself.

```
library(tibble)
library(knitr)

d <- survival::diabetic
d$arm <- factor(d$trt, levels = c(0, 1),
                labels = c('Untreated', 'Laser'))

arms <- tibble(
  Arm = levels(d$arm),
  Eyes = as.integer(table(d$arm)),
  Events = as.integer(tapply(d$status, d$arm, sum)),
  `Event rate` = sprintf(
    '%.3f', as.numeric(tapply(d$status, d$arm, mean))
  )
)

kable(arms, align = 'lrrr')
```

Table 11.1.: Eyes and events by treatment arm in the diabetic retinopathy trial (`survival::diabetic`), the real dataset this book computes on. Each count is verifiable by hand from the 394 records, which is what makes the table usable as a test oracle rather than merely as a description.

Arm	Eyes	Events	Event rate
Untreated	197	101	0.513
Laser	197	54	0.274

The difference between the arms is large and in the expected direction. That is a reason to be careful rather than pleased, because a validation test should assert what the protocol requires, not what this particular extract

Draft, November 13, 2026. Content may change.

happens to show. An assertion that the event rate in the laser arm is below one third would pass today. It would fail on a different trial, or on a corrected export, for reasons having nothing to do with a defect.

```
# abridged from inst/tinytest/test-cohort.R
raw <- load_cohort('../../analysis/data/raw_data/cohort_ra
 ↪  w.csv')
cohort <- clean_cohort(raw)

# the expected columns are present
expect_true(all(c('mrn', 'age_at_enroll', 'sex',
 ↪  'incident_dm')
                %in% names(raw)))

# ages lie within the enrollment criteria, not merely
# within the observed range
expect_true(all(cohort$age_at_enroll >= 18 &
                cohort$age_at_enroll <= 105, na.rm = TRUE))

# sex takes only its two coded levels once cleaned
expect_true(all(cohort$sex %in% c('F', 'M')))

# outcome missingness stays within what the analysis
# can tolerate
expect_true(mean(is.na(cohort$incident_dm)) < 0.15)
```

These are the tests that most directly repay a health scientist, because they catch the errors that no amount of correct code can prevent, the errors that live in the data before the analysis begins. In a data-analysis or simulation archetype, where there may be few reusable functions to unit-test, the data-validation tests often carry most of the suite's weight.

11.7.4. Reproducibility tests

The fourth kind closes the loop back to the ladder. A reproducibility test checks that a recorded result regenerates. Its most tempting form runs

Draft, November 13, 2026. Content may change.

a seeded procedure twice in one session and asserts that the two results agree:

```
# a same-session check: it cannot detect what it
# appears to guard against
expect_equal(bootstrap_ci(cohort, B = 2000, seed = 2024),
             bootstrap_ci(cohort, B = 2000, seed = 2024))
```

Note first where the seed lives. `bootstrap_ci()` takes it as an argument and sets it internally, with `withr::with_seed()`, which also restores the caller's generator state on exit. As a result, the function is deterministic given its arguments and does not disturb any random stream around it. A procedure that depends on the caller setting a seed first is one `set.seed` away from irreproducible, and moving the seed into the signature makes forgetting it impossible.

> **Pitfall**
>
> A same-session check cannot fail for the reason it seems to serve. Both calls run in the same R, under the same random-number defaults, with the same package versions. When an upgrade changes how `sample()` draws, as R 3.6.0 did, both calls change together and still agree. What the check does catch is narrower: a procedure that ignores its seed. That is worth one line in a unit test, but it is not a reproducibility test.

A reproducibility test that can fail compares today's result with one recorded in an earlier session and committed to the repository. We call it a *baseline test*. It has three parts: a script, run once, that records the baseline; the committed baseline file; and a test file that recomputes the quantities and compares them with a stated tolerance.

```
# analysis/scripts/generate_baseline.R, abridged. Run once
# from the project root to record the baseline, then commit
# inst/baseline/cohort_baseline.rds.
# The book's own baseline was recorded this way, and the
# file is now committed at that path; regenerate it only
# if the generator, the seed, or the model change.
```


Draft, November 13, 2026. Content may change.

```
result <- run_canonical_analysis(
  'analysis/data/raw_data/cohort_raw.csv'
)
dir.create('inst/baseline', recursive = TRUE,
           showWarnings = FALSE)
saveRDS(result, 'inst/baseline/cohort_baseline.rds')
```

```
# inst/tinytest/test-baseline-cohort.R, abridged: the
# real file also searches fallback paths for both files
raw_path <- '../../analysis/data/raw_data/cohort_raw.csv'
if (!file.exists(raw_path)) {
  exit_file('raw extract absent; run simulate_cohort.R')
}
baseline <- readRDS(
  system.file('baseline', 'cohort_baseline.rds',
              package = 'reproducibility'))
result <- run_canonical_analysis(raw_path)

expect_equal(result$n_rows, baseline$n_rows,
  info = 'cleaned row count matches baseline')
expect_equal(result$n_events, baseline$n_events,
  info = 'event count matches baseline')
expect_equal(result$hr, baseline$hr, tolerance = 1e-8,
  info = 'Cox HR matches committed baseline')
expect_equal(result$lower, baseline$lower,
  tolerance = 1e-8,
  info = 'lower CI bound matches committed baseline')
expect_equal(result$upper, baseline$upper,
  tolerance = 1e-8,
  info = 'upper CI bound matches committed baseline')
```

Three details carry the weight. The path is relative to `inst/tinytest/`, where `tinytest` runs the file. The guard lets the file exit early under `R CMD check`, which installs the package without `analysis/`. The mirror in `inst/testdata/` covers the unit and data-validation tests; the baseline test requires the raw extract and skips cleanly without it. Finally, the tolerance is stated and chosen, not left at a default. `1e-8` relative is wide enough to

Draft, November 13, 2026. Content may change.

absorb floating-point summation reordered by a different BLAS or thread count (Chapter 7), and far too narrow for a change in the fitted coefficients to pass. A comparison with no tolerance would fail on that reordering, and a check that fails for reasons unrelated to the analysis trains its reader to ignore it.

The test can be watched failing for the reason it exists. The chunk below plays both sessions on the synthetic cohort. It records the 97.5 percent bootstrap bound for mean BMI under the sampler R used before version 3.6.0 (`sample.kind = 'Rounding'`), as an analysis recorded on an older R would have. It then reruns it with the same seed and sampler, and reruns it again with the same seed under the current default (`'Rejection'`). The comparison is the one `expect_equal()` makes, `all.equal()` at the stated tolerance. `withr::with_seed()` sets the sampler only for the duration of each call, and the final `stopifnot()` confirms the session's generator is as it was.

```
library(knitr)

cohort <- read.csv('../data/raw_data/cohort_raw.csv')
bmi <- cohort$bmi[!is.na(cohort$bmi)]

boot_upper <- function(x, seed, sample_kind, reps = 2000) {
  withr::with_seed(
    seed,
    {
      means <- replicate(reps, mean(sample(x, replace =
 ↪  TRUE)))
      unname(quantile(means, 0.975))
    },
    .rng_sample_kind = sample_kind
  )
}

kind_before <- RNGkind()[3]

# R warns whenever the pre-3.6.0 sampler is selected;
# the warning is expected here and muffled.
baseline <- suppressWarnings(boot_upper(bmi, 2024,
 ↪  'Rounding'))
```

```
rerun_same <- suppressWarnings(boot_upper(bmi, 2024,
 ↪  'Rounding'))
rerun_new <- boot_upper(bmi, 2024, 'Rejection')

tol <- 1e-8
passes <- function(current) {
  isTRUE(all.equal(current, baseline, tolerance = tol))
}

checks <- data.frame(
  run = c('recorded baseline (Rounding)',
          'rerun, same seed and sampler (Rounding)',
          'rerun, same seed, current default (Rejection)'),
  upper_bound = c(baseline, rerun_same, rerun_new),
  passes = c('(baseline)',
             ifelse(passes(rerun_same), 'yes', 'no'),
             ifelse(passes(rerun_new), 'yes', 'no'))
)

stopifnot(identical(RNGkind()[3], kind_before))

kable(checks, digits = 4,
      col.names = c('Run', '97.5% bootstrap bound',
                    'Passes at tolerance 1e-8'))
```

Table 11.2.: A baseline test across sessions. The bound recorded under the pre-3.6.0 sampler is reproduced exactly by a rerun with the same seed and sampler, and is not reproduced seed-exactly by a rerun with the same seed under the current default sampler, so the comparison at tolerance 1e-8 passes once and fails once. A same-session check would have passed in both cases.

Run	97.5% bootstrap bound	Passes at tolerance 1e-8
recorded baseline (Rounding)	29.1400	(baseline)
rerun, same seed and sampler (Rounding)	29.1400	yes

Draft, November 13, 2026. Content may change.

Table 11.2.: A baseline test across sessions. The bound recorded under the pre-3.6.0 sampler is reproduced exactly by a rerun with the same seed and sampler, and is not reproduced seed-exactly by a rerun with the same seed under the current default sampler, so the comparison at tolerance 1e-8 passes once and fails once. A same-session check would have passed in both cases.

Run	97.5% bootstrap bound	Passes at tolerance 1e-8
rerun, same seed, current default (Rejection)	29.1436	no

The rerun under the current sampler moves the bound by 0.0036 units of BMI, from 29.1400 to 29.1436. That is a small number, and it is not a numerical artifact: the resamples themselves are different. The baseline test reports it; two calls in one session, both under whichever sampler the session uses, would have agreed with each other and reported nothing.

Failing the seed-exact comparison is not the same as failing to reproduce the result, because the bound is itself a Monte Carlo estimate. The chunk below estimates its Monte Carlo standard error (MCSE) as the standard deviation of the bound over 40 seeds, at the table's 2000 replicates, and then applies a two-tier check to the two reruns.

```
mc_bounds <- vapply(1:40, function(s) boot_upper(bmi, s,
 ↪  'Rejection'),
                    numeric(1))
mcse <- sd(mc_bounds)
two_tier <- compare_to_baseline(c(rerun_same, rerun_new),
                                c(baseline, baseline), mcse
                                 ↪  = mcse)
two_tier
#> [1] "exact"                    "within Monte Carlo
 ↪  error"

# Decimals an MCSE supports: the MCSE must be below half a
# unit of the last reported digit.
```


Draft, November 13, 2026. Content may change.

```
digits_supported <- function(mcse) max(0, floor(-log10(2 *
 ↪  mcse)))
d_bmi <- digits_supported(mcse)
# An MCSE from 40 seeds is itself uncertain by about 11
# percent, so two significant figures are all it supports.
sig2 <- function(x) {
  formatC(signif(x, 2), digits = 2, format = 'fg', flag =
   ↪  '#')
}
stopifnot(round(baseline, d_bmi) == round(rerun_new,
 ↪  d_bmi))
```

The MCSE of the bound is 0.015, so the sampler change is about 0.2 of one MCSE. The seed-exact comparison fails, yet the change is a fraction of one MCSE, and the statistical result reproduces.

This is the two-tier baseline check. The first tier compares at the numerical tolerance `1e-8`; a pass means the draws themselves regenerated. On a failure, the second tier accepts a difference of at most three MCSE, with the MCSE recorded beside the baseline value. `compare_to_baseline()` in the book's package implements both tiers, and it returns 'exact' for the same-sampler rerun and 'within Monte Carlo error' for the current sampler. Both the recorded value and the rerun carry Monte Carlo error, so their difference has a standard deviation of about $\sqrt{2}$ MCSE. Under normality, three MCSE then flags a pure change of draws about 3.4 percent of the time, which is the price of a second tier that still catches a real change.

The second tier applies only to a quantity that has Monte Carlo error, such as a bootstrap bound or a simulation summary. The Cox hazard ratio and its confidence limits in `test-baseline-cohort.R` are deterministic functions of the data, so only the first tier applies to them, and `compare_to_baseline()` with `mcse = NULL` reduces to it. The precision of the table follows the same distinction. Its four decimals are there for the seed-exact tier. As a statistical result, an MCSE of 0.015 supports reporting the bound as 29.1 under either sampler, by the rule the next section gives.

Draft, November 13, 2026. Content may change.

Verification

To see this hold on your own machine: generate the raw extract (`Rscript analysis/scripts/simulate_cohort.R` from the project root), then run `RENV_CONFIG_AUTOLOADER_ENABLED=FALSE Rscript -e 'pkgload::load_all("."); tinytest::run_test_dir("inst/tinytest")'` and confirm `test-baseline-cohort.R` reports five passes (habit 4). To make it fail on purpose, open `inst/baseline/cohort_baseline.rds`, read it with `readRDS()`, change `hr` by more than `1e-8` relative, write it back, and rerun. The comparison above, run with `sample.kind = 'Rejection'` against a baseline recorded under `'Rounding'`, is this exact failure, executed in the chapter itself.

A baseline test is what L3 requires. The top rung, as Chapter 3 defines it, asks that recorded outputs be shown to regenerate within a stated tolerance from a clean rebuild. The baseline test supplies the recorded outputs and the tolerance; running it inside an environment rebuilt from its recipe supplies the clean rebuild, and Chapter 12 assembles the two in continuous integration. A suite with no baseline test can show that the code runs and that its functions are correct on known inputs. However, it cannot show that a recorded result regenerates, so it does not reach L3 however thorough it is. In the zzcollab composition, `zzc verify --full` applies this rule. It stamps L3 only when the image rebuilds, the test suite passes, and at least one `inst/tinytest/test-baseline*.R` file exists and is named in the test log. Otherwise it reports the project as rebuilt and tested, at L2, with no baseline comparison. A baseline whose guard exits early is still named in the log, so the guard must not fire in the image.

11.8. Monte Carlo error and parallel streams

Table 11.2 fixed the seed and varied only the sampler (the default that R changed in version 3.6.0), which isolated one source of change in the bootstrap bound. A second source is always present and has nothing to do with R versions. A bootstrap estimate is itself a Monte Carlo quantity, built from a finite number of resamples. It carries its own sampling variability on top of the statistical sampling variability of the original cohort. Two

Draft, November 13, 2026. Content may change.

bootstrap runs at different seeds, same data, same number of replicates `B`, will not return exactly the same bound. The standard deviation of the bound across seeds is its Monte Carlo standard error (MCSE), distinct from the standard error the bootstrap is estimating.

But how much does such a bound move in practice? The practical diagnostic needs no extra theory: rerun the same call at many seeds for a given `B`, and take the standard deviation of the result; then raise `B` and watch it shrink. The chunk below does this with `bootstrap_ci()` on the HbA1c odds ratio, at `B = 250` and at the package default `B = 1000`, with 20 seeds each.

```
cohort <- clean_cohort(
  load_cohort('../../analysis/data/raw_data/cohort_raw.csv
  ↪ ')
)
bounds_at <- function(B) {
  t(vapply(1:20, function(s) bootstrap_ci(cohort, B = B,
  ↪ seed = s),
           numeric(2)))
}
b250 <- bounds_at(250)
b1000 <- bounds_at(1000)
mcse250 <- apply(b250, 2, sd)
mcse1000 <- apply(b1000, 2, sd)
ratio <- mcse250 / mcse1000
# With 20 seeds the log of a ratio of two SDs has a
↪ standard
# error near sqrt(2 / 38); the prose calls the ratios
# consistent with 2 only while they are within two of them.
stopifnot(all(abs(log(ratio / 2)) < 2 * sqrt(2 / 38)))
d_upper <- digits_supported(mcse1000['upper'])
d_lower <- digits_supported(mcse1000['lower'])
stopifnot(d_upper < 2)
ci_default <- bootstrap_ci(cohort)
# Replicates for the MCSE to fall below half a unit of one
# more decimal, scaling the observed MCSE by 1/sqrt(B).
B_for <- function(mcse, d, B = 1000) {
  ceiling(B * (mcse / (0.5 * 10^-d))^2)
}
```


Draft, November 13, 2026. Content may change.

```
B_upper_next <- B_for(mcse1000['upper'], d_upper + 1)
```

At `B = 250` the MCSE of the upper bound is 0.43 and that of the lower bound is 0.072. At `B = 1000` they are 0.22 and 0.040, on bounds near 5.5 and 1.5. An MCSE that scales as $1/\sqrt{B}$ halves when `B` is quadrupled, and the observed ratios are 2.0 for the upper bound and 1.8 for the lower. With 20 seeds, each estimated standard deviation is itself uncertain by about 16 percent, so these ratios are consistent with that scaling but do not pin it closely.

The standard guidance for simulation studies is to report the MCSE of every Monte Carlo estimate and to choose the number of repetitions from the precision required (Morris et al., 2019). Applied to reported digits, it gives a rule for the number of replicates. Choose `B` so that the MCSE of each reported quantity is below about half a unit of its last reported digit, or report only the digits the MCSE supports. At the default `B = 1000` the rule allows the lower bound 1 decimal place and the upper bound 0, so the default call's interval should be reported as 1.5 to 5. Reporting the upper bound to one more decimal would need about 19,000 replicates. A two-decimal interval from `bootstrap_ci()` at its default therefore prints digits that are Monte Carlo noise. The baseline test's `1e-8` in Table 11.2 is a numerical tolerance for one fixed sequence of draws, and it says nothing about this. The MCSE is estimated once, when choosing `B`, and recorded with the baseline as the scale of the second tier; the test suite does not re-estimate it on every run.

Resampling at the `B` a real analysis needs is often parallelized across cores, and the random streams then need as much care as the seed. Naive parallelization fails first by irreproducibility. Under the default Mersenne-Twister generator and the default `mc.set.seed = TRUE`, `parallel::mclapply()` discards the inherited generator state in each child process, which then seeds itself from the clock and its process id. A `set.seed()` before the call therefore does not fix the result, and two runs differ. With `mc.set.seed = FALSE` the children inherit the parent's state, and the workers draw identical streams, so supposedly independent replicates are duplicated. The L'Ecuyer-CMRG generator, which `RNGkind('L\'Ecuyer-CMRG')` selects, was built to provide many independent streams. Under it, `mclapply()` gives each child its own stream, and `parallel::clusterSetRNGStream()`

Draft, November 13, 2026. Content may change.

gives one to each worker of a cluster. Per-worker streams are independent, but the result still depends on the number of workers. The replicates are divided among the workers in chunks, so a given replicate draws from a different stream when the worker count changes.

Invariance to the worker count needs per-task streams: one stream for each replicate, fixed before any worker starts. `parallel::nextRNGStream()` can precompute them, the `future.apply` package does so with `future.seed = TRUE`, and the `doRNG` package does so for a `foreach` loop. The book's package provides `replicate_streams()`, which precomputes the streams with `nextRNGStream()` and runs the replicates with `mclapply()`. The chunk below runs the same 2000 bootstrap means of BMI on one worker and on two. Forking is unavailable on Windows, where both runs are sequential and the comparison is trivially true.

```
boot_mean <- function(i) mean(sample(bmi, replace = TRUE))
one_worker <- replicate_streams(2000, boot_mean, seed =
 ↪  2024,
                                 cores = 1)
two_workers <- replicate_streams(2000, boot_mean, seed =
 ↪  2024,
                                  cores = 2)
identical(one_worker, two_workers)
#> [1] TRUE
```

The two runs are identical (TRUE), replicate by replicate, because replicate `i` draws from stream `i` wherever it runs. The same seed discipline that makes a serial bootstrap reproducible extends to a parallel one only in this per-task form.

> Check 7: two kinds of instability
>
> A bootstrap bound changes when rerun with a different seed at the same `B`. It also changes when rerun with the same seed after an R version upgrade changes the default sampler (Table 11.2). Are these the same problem, and does increasing `B` fix both?
> *The answer is on page 479.*

Draft, November 13, 2026. Content may change.

Each of the four kinds catches defects the others do not see, and Figure 11.2 sets four representative defects against them. Start with the bottom row. Re-execution alone, with no test of any kind, catches none of the four, because every one of these defects is deterministic and will reproduce perfectly. That row is the reproducibility paradox in tabular form.

```
library(ggplot2)
library(tibble)

defects <- c('dropped rows change\nthe sample size',
             'ages encoded\nin months',
             'variance returned\nfor a standard deviation',
             'seeded bootstrap moves\nafter an R upgrade')
kinds <- c('Re-execution alone',
           'Reproducibility test',
           'Data-validation test',
           'Integration test',
           'Unit test')

cover <- expand.grid(defect = defects, kind = kinds,
                     stringsAsFactors = FALSE)
cover$defect <- factor(cover$defect, levels = defects)
cover$kind <- factor(cover$kind, levels = kinds)

hits <- c('Unit test|variance returned\nfor a standard
 ↪  deviation',
          'Integration test|dropped rows change\nthe sample
           ↪  size',
          'Data-validation test|dropped rows change\nthe
           ↪  sample size',
          'Data-validation test|ages encoded\nin months',
          'Reproducibility test|seeded bootstrap
           ↪  moves\nafter an R upgrade')
cover$catches <- paste(cover$kind, cover$defect, sep = '|')
 ↪  %in% hits

ggplot(cover, aes(x = defect, y = kind)) +
  geom_tile(aes(fill = catches), color = 'white', linewidth
   ↪  = 2) +
```


Draft, November 13, 2026. Content may change.

```
  scale_fill_manual(values = c('TRUE' = '#14746f',
                               'FALSE' = '#eef2f2'),
                    guide = 'none') +
  labs(x = NULL, y = NULL) +
  theme_minimal(base_size = 9) +
  theme(
    panel.grid = element_blank(),
    axis.text.x = element_text(size = 7.6, lineheight =
     ↪  0.95),
    axis.text.y = element_text(face = 'bold', size = 8.4)
  )
```

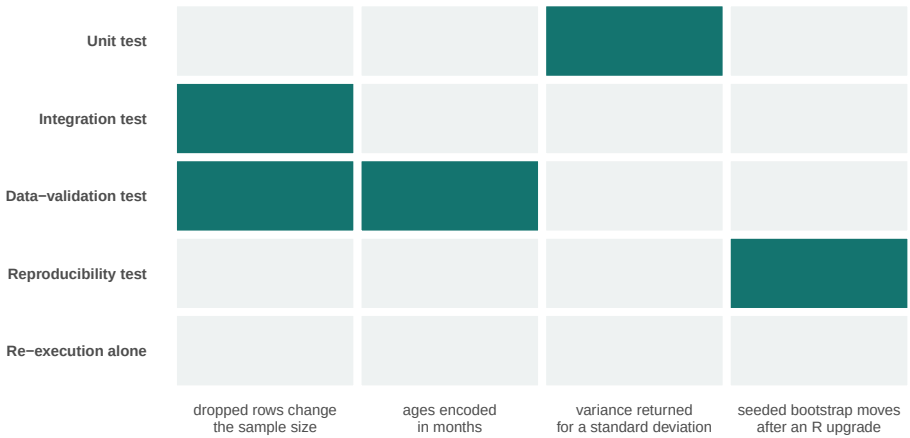


Figure 11.2.: Four representative defects against the four kinds of test, with a fifth row for re-execution alone. A filled cell marks the kind of test most likely to catch that defect. The matrix is close to diagonal, which is the point: the kinds are complements rather than alternatives, and a suite missing one kind is blind to a whole class of defect. The bottom row is empty because every defect shown is deterministic, so re-running the analysis reproduces it faithfully.

Draft, November 13, 2026. Content may change.

Check 8: which test catches it?

For each defect, name the kind of test most likely to catch it. (a) A colleague refactors `clean_cohort` and accidentally drops the rows with a missing outcome, changing the sample size. (b) An upstream export encodes age in months rather than years, so ages read as up to fourteen hundred. (c) A function meant to return a standard deviation returns a variance. (d) After an R upgrade, a seeded bootstrap returns different draws than the manuscript reported.
The answer is on page 480.

11.9. Test behavior, not implementation

A test is worth writing only if it survives changes that ought not to break it and fails on changes that ought to. The way to secure this is to test what a function does, its observable behavior, rather than how it does it, its internal implementation. The distinction is easiest to see in a pair of tests for the same function:

```
# fragile: pins an implementation detail
expect_true('filter' %in% all.names(body(my_clean)))

# robust: pins the behavior
result <- my_clean(data.frame(x = c(1, NA, 3)))
expect_false(anyNA(result))
```

The first test asserts that `my_clean` uses a function called `filter` internally. If the analyst later rewrites `my_clean` in base R, removing the `filter` call while leaving its behavior unchanged, the first test fails. It fails for no good reason: the function still does exactly what it promised. Such a test punishes honest refactoring and trains the analyst to distrust the suite. The second test asserts only that the cleaned data carries no missing values, which is the behavior that actually matters. It passes under any correct implementation. As a general rule, a test should fail when, and only when, the function's contract is broken. Tests that fail for other reasons are a tax on maintenance. Wilson and colleagues, in their best-practices

and good-enough-practices guidance, make behavioral testing a recurring recommendation (Wilson et al., 2014; Wilson et al., 2017).

A corollary concerns how much to test. A test that merely confirms a result is a data frame, without checking any value in it, tests too little and gives false comfort. A test that pins every incidental property tests too much and breaks on every revision. Deciding what to pin is the second of the analyst's contributions named above, and no tool makes that decision.

Check 9: fragile or robust

`expect_equal(environment(my_clean)$helper_count, 3)` checks that an internal counter variable inside `my_clean` equals three. Is this a behavioral test or an implementation test, and what happens to it the first time `my_clean` is honestly rewritten?
The answer is on page 480.

11.10. Code quality as a validation feature

Testing confirms that code is correct. A second family of tools confirms that code is *legible and consistent*. Although these tools do not check correctness directly, they belong to the same validation register and serve the same reproducibility end. The argument is one of mechanism. Code that a reader can follow is code whose errors a reader can find. Consistent formatting removes the visual noise that hides a transposed argument or a misplaced parenthesis. Moreover, a single house style across a compendium means that a difference in the code is always a difference in meaning, never merely a difference in whitespace. Wilson and colleagues, in both of their practice papers, treat readable, conventional code as a component of reproducible work rather than an aesthetic preference (Wilson et al., 2014; Wilson et al., 2017). Sandve and colleagues, in their ten simple rules, make the companion point that the provenance and the recorded results of an analysis must themselves be preserved (Sandve et al., 2013).

Two tools mechanize this, and a third enforces them.

Linting is static analysis: `lintr` reads the source without running it and flags stylistic deviations and likely errors. Examples are an assignment with

Draft, November 13, 2026. Content may change.

`=` where the house style is `<-`, a line past the width limit, a variable assigned and never used, and a call to a function that does not exist. Because it does not execute the code, it catches a class of mistake that tests, which only exercise the paths they call, may miss.

Styling is automatic reformatting: `styler` rewrites the source into a consistent layout, fixing indentation, spacing, and alignment, so that formatting is never a matter for human attention or for a code review. The two are complementary; `styler` fixes the formatting that `lintr` would otherwise complain about.

In the framework, both run inside the project container through Makefile targets, so that no host R installation is required and every collaborator runs the identical checks:

```
$ make style   # reformat R code with styler
$ make lint    # analyze R code with lintr
```

A pre-commit hook enforces them. A hook is a script that git runs automatically at a defined moment; a pre-commit hook runs before a commit is recorded, and can refuse the commit if a check fails. Configuring the linter and styler as a pre-commit hook means that unstyled or unlinted code cannot be committed from a clone where the hook is installed, unless the analyst bypasses it with `git commit --no-verify`. That is a stronger guarantee than a convention that relies on memory, but a clone does not carry the hook with it. The configuration lives in a `.pre-commit-config.yaml` file at the project root:

```
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/lorenzwalthert/precommit
    rev: v0.4.3
    hooks:
      - id: style-files
      - id: lintr
      - id: readme-rmd-rendered
      - id: no-browser-statement
```


Draft, November 13, 2026. Content may change.

With this in place, an attempt to commit code that `styler` would reformat, or that `lintr` flags, or that contains a stray `browser()` left over from debugging, is stopped before the commit completes. The analyst is then prompted to fix it. The hook is the fast local gate. A collaborator who has not installed it is caught only if continuous integration (Chapter 12) runs the same checks on the server. The workflows in that chapter do not, so a project that relies on the hook should add a step running `lintr::lint_dir()` there.

Check 10: capture or validation?

Classify each of the following as belonging to the capture register or the validation register, and say what would be lost by removing it. (a) The `renv.lock` lockfile. (b) A unit test of `calculate_mean`. (c) The `.pre-commit-config.yaml` hook. (d) The Dockerfile.
The answer is on page 480.

11.11. Validation in the regulated sense

Every use of 'validation' in this chapter has been the statistical and software-engineering sense: checking that code is correct, that data meet a constraint, that a result regenerates. In pharmaceutical, medical-device, and other FDA- or EMA-regulated work, 'validation' names a broader and more procedural idea. Chapter 3's warning box introduces the distinction; this section develops the three practices that fill it out for testing specifically.

Computer system validation. Under 21 CFR 11.10(a), a closed system that creates, modifies, maintains, or transmits regulated electronic records must be validated 'to ensure accuracy, reliability, consistent intended performance, and the ability to discern invalid or altered records' (*21 CFR Part 11*, n.d.). The European Medicines Agency sets parallel expectations for computerized systems in clinical trials, and names statistical software among them (European Medicines Agency, GCP Inspectors Working Group, 2023). This is validation of the system across its lifecycle, documented once and maintained by a change-control process, not a one-time test run. The regulation does not say how that validation is shown. Installation, operational, and performance qualification (IQ, OQ, and PQ) come from

Draft, November 13, 2026. Content may change.

qualification practice, in which a system qualification typically executes all three (Nicholls et al., 2020), not from the text of 11.10. The requirement's reach is also narrower than its wording suggests. FDA's 2003 guidance on the scope and application of Part 11 states that the agency will exercise enforcement discretion over the Part 11 validation requirement, while the predicate rules, the underlying requirements of the Act and of FDA's other regulations, still apply, their own validation requirements included (U.S. Food and Drug Administration, 2003). The R Validation Hub concludes that Part 11 is not relevant or mandatory for statistical analysis software itself, though elements of it apply when R is used as part of a validated system (Nicholls et al., 2020). What an inspector asks in practice is whether the statistical computing environment is in a documented, validated state under the sponsor's standard operating procedures and the predicate rules. A `tinytest` suite that passes is evidence toward that state. It is not, by itself, that state, which also rests on qualification records and a record of who approved each change and when.

Double programming. Independent of any code-based test, a regulated submission is commonly checked by *double programming*. Two analysts, or one analyst and an independent reviewer, implement the same analysis separately, from the same specification, usually in isolation from each other's code. The two results are then compared. This is a different reproducibility practice from everything else in this chapter. A baseline test reruns the *same* code and checks that it still returns the same answer; double programming runs *two different* implementations of the same specification and checks that they agree. Consider a bug shared by both implementations, because both analysts misread the same ambiguous protocol sentence. It would pass a baseline test (the code is unchanged) and would also pass double programming (the two analysts made the same mistake). What double programming catches is a bug particular to one implementation, the error a single analyst's code review is least likely to find in their own work.

Package risk assessment. Pinning package versions, as Chapter 6 teaches, fixes which code runs. It says nothing about how much to trust that code. But how is that trust to be judged? The R Validation Hub, an R Consortium working group formed for exactly this question, proposes a risk-based answer (Nicholls et al., 2020). The first step is to classify each package as either 'intended for use' (called directly by the analysis) or an unexamined dependency reached only through another package. The second is, for each

Draft, November 13, 2026. Content may change.

intended-for-use package, to assess its purpose, maintenance practice, community usage, and existing test coverage. The third is to have a qualified reviewer record a documented risk judgment. Higher risk earns additional targeted testing. Base R and the recommended packages are treated as minimal risk. For other widely used packages, such as the tidyverse, the white paper says only that there is a reasonable case to treat their maintainers as a trusted resource, and leaves that decision to each organization. Whatever the risk score, tests are still required for the statistical modeling functions of an intended-for-use package. The `riskmetric` package operationalizes several of these metrics so the assessment does not start from nothing. This is a judgment about trustworthiness layered on top of the pinning Chapter 6 already does, not a replacement for it.

Check 11: three validations

A compendium has a committed `renv.lock`, a passing `tinytest` suite, and a documented sign-off that `dplyr` 1.1.4 has been risk-assessed as trusted. An inspector asks whether the statistical computing environment is in a documented, validated state under the sponsor's standard operating procedures. What is missing from the three items given?
The answer is on page 481.

11.12. Reviewing analysis code

Double programming is costly, because it implements every output it covers twice. Most analyses, regulated or not, are checked instead by review: a second analyst reads a proposed change, usually as a pull request (Chapter 4), and approves it before it merges. The two practices find different errors. A reviewer reads one implementation against the plan, and is well placed to see a wrong exclusion, a covariate the plan does not name, or a seed set in the wrong place. A reviewer is poorly placed to see an arithmetic slip in code that reads correctly, which is the error a second, independent implementation exposes. The choice between them is therefore a judgment about risk, made per output and recorded. One defensible arrangement is to double-program the outputs a decision rests on, such as the primary estimate, and to review the rest.

Draft, November 13, 2026. Content may change.

A reviewer needs a short list rather than a general impression. For a pull request that changes an analysis, we suggest six questions.

1. Does every output the change affects, whether a table, a figure, or a number in the prose, trace to committed code, with no value typed by hand?
2. Are the inputs read only from the raw data, never from a hand-edited intermediate file or an object left in a session?
3. Are the exclusions and derived variables the ones the analysis plan states, with the same thresholds and the same handling of missing values?
4. Is each seed set at the call that uses it, in the analysis code, rather than in `.Rprofile` or nowhere?
5. Did the tests change alongside the code, and does the pull request say why? An expected value edited to match new output needs the oracle that justifies the edit.
6. Does the log of the render or the test run show warnings nobody expected? A clean log is one in which every warning has been read and then removed or explained.

The list draws on the book's gates, but it is not a gate. It asks for the judgments that continuous integration cannot make (Chapter 12): whether a threshold matches the plan, and whether a changed test was changed for a good reason. The approval on the pull request then records who checked the change and when, the same kind of record the validated state described above rests on. Questions 3 and 5 take the longest, because answering them means opening the plan and the test diff rather than reading the code alone.

11.13. Testing across the archetypes

The four kinds of test are the same in every compendium, but their relative weight shifts with the archetype. Table 5.2 in Chapter 5 is the reference. We state the shift here so that a student who has met testing only in the context of R packages does not wrongly conclude that a data analysis has nothing to test. Testing is native to the **R-package** archetype, whose suite is dominated by unit tests that `R CMD check` runs as a matter of

Draft, November 13, 2026. Content may change.

course. The **data-analysis** and **simulation** archetypes, by contrast, may have few reusable functions and no exported interface. In these the data-validation and reproducibility tests carry the weight instead. A simulation in particular leans on a baseline test that fixes the seed and compares the result with a committed value, since its entire output is a function of that seed. Chapter 12 develops the point at length. The **manuscript** archetype adds a test that the report renders and that its headline numbers match a baseline. The **blog** archetype tests the functions and example computations its posts depend on. That is a reproducibility obligation to readers no smaller than a manuscript's to its reviewers.

> Check 12: shifting weight
>
> A data-analysis compendium with no exported functions and no package structure still needs tests. Which two kinds of test carry most of that weight, and why?
> *The answer is on page 481.*

11.14. Worked example

We now make the paradox concrete and executable. The following chunk defines the `calculate_mean` function in two forms, one correct and one carrying the bug of the orientation, and checks each against three inputs whose means are known by hand. Rather than call the `tinytest` package, which may be absent from a bare R session, we hand-roll a one-line expectation. That is all a bare check really is: a comparison of an actual value against an expected one.

```
# the function under test, in a correct and a
# buggy form
calculate_mean <- function(x, na.rm = FALSE) {
  if (na.rm) x <- x[!is.na(x)]
  sum(x) / length(x)
}

calculate_mean_buggy <- function(x) {
```


Draft, November 13, 2026. Content may change.

```
  sum(x)                  # bug: division dropped
}

# a minimal hand-rolled expectation: no testing
# package required
expect <- function(actual, expected, tol = 1e-8) {
  isTRUE(abs(actual - expected) < tol)
}

# three inputs whose means are known independently
cases <- list(c(1, 2, 3),
              c(10, 20, 30, 40),
              c(5, 5, 5, 5, 5))
oracle <- c(2, 25, 5)

results <- data.frame(
  input = vapply(cases,
                 function(x) paste(x, collapse = ', '),
                 character(1)),
  expected = oracle,
  correct = vapply(cases, calculate_mean, numeric(1)),
  buggy = vapply(cases, calculate_mean_buggy,
                 numeric(1))
)
results$correct_passes <-
  mapply(expect, results$correct, oracle)
results$buggy_passes <-
  mapply(expect, results$buggy, oracle)

knitr::kable(results)
```

Table 11.3.: A correct function and a buggy variant checked against three inputs with hand-computed means. The buggy variant returns the same wrong value on every input and would return it on every machine and every rerun: it is perfectly reproducible and still wrong.

input	expected	correct	buggy	correct_passes	buggy_passes
1, 2, 3	2	2	6	TRUE	FALSE
10, 20, 30, 40	25	25	100	TRUE	FALSE
5, 5, 5, 5, 5	5	5	25	TRUE	FALSE

The correct function passes every check; the buggy function fails every check, and it fails them identically. Were we to rerun this chunk a thousand times, on a thousand machines, the `buggy` column would hold the same wrong values every time. The bug is not a source of variability, which reproducibility tooling might eventually surface. It produces the same error on every run, and only a comparison against an independent oracle can catch it. That comparison is what a test is.

In a real compendium, the hand-rolled `expect` gives way to the `tinytest` expectations, and the checks live in a file under `inst/tinytest/`. The following is that file, shown as a static listing that is not run here. It is exactly the unit test that would have caught the bug before it reached a result:

```
# inst/tinytest/test-calculate_mean.R
expect_equal(calculate_mean(c(1, 2, 3)), 2)
expect_equal(calculate_mean(c(10, 20, 30, 40)), 25)
expect_equal(calculate_mean(rep(5, 5)), 5)
expect_true(is.nan(calculate_mean(numeric(0))))
expect_equal(calculate_mean(c(1, NA, 3),
                            na.rm = TRUE), 2)
```

The runner `tests/tinytest.R` that `R CMD check` invokes is a three-line file:

Draft, November 13, 2026. Content may change.

```
# tests/tinytest.R
if (requireNamespace('tinytest', quietly = TRUE)) {
  tinytest::test_package('mycompendium')
}
```

and the pre-commit configuration shown earlier completes the picture. The linter and styler guard the code's legibility on every commit, and the tinytest suite guards its correctness. Continuous integration re-runs both on a clean machine, so that neither guarantee depends on the author's local setup.

Check 13: the oracle in the table

In Table 11.3, why is `oracle <- c(2, 25, 5)` written by hand in the chunk rather than computed by calling `calculate_mean()` on the three `cases`?
The answer is on page 481.

11.15. Collaborating with an LLM

A language model is a capable assistant for writing tests, and a dangerous one for a specific reason: it can readily produce tests that pass, which is not the same as tests that are correct. The analyst's verification duty is heaviest exactly where the model is most fluent.

! Restricted data

One rule comes first (habit 3). Never paste or upload protected health information, identifiable records, or data held under a data use agreement, nor outputs derived from them, into a model or a coding agent unless three conditions all hold. The institution has approved that endpoint for that class of data. The IRB protocol and the consent permit the processing. The data use agreement, or the provider's terms, permits it. If any of the three is unclear, the answer is no.
The third condition is the one most often missed. Institutional approval of a tool does not override the terms under which a data provider

released the data. NIH, for example, treats sharing controlled-access data through prompts with public generative AI tools, which it glosses as third-party tools, as a breach of the data use certification. It treats a model trained on such data as a data derivative, to be shared only with other approved users and not kept after the project closes (National Institutes of Health, 2025a).
Describe the structure instead: the column names and types, or a small synthetic example generated from the data dictionary, never edited from real rows. A coding agent sees whatever files are in its working tree, so run it in a checkout or container where the real data is not mounted.
An agent also acts, so confine it to a branch and keep merges, pushes, and deposits for yourself (Section 2.10.1).

Tests sit close to the data, so the derived outputs the rule covers include a failing test's printout of offending rows. The book's simulated cohort is a suitable synthetic example to describe instead.

Prompt. 'Here is my `calculate_mean` function. Write `tinytest` unit tests for it, including edge cases.'

Watch for. Check whether each expected value was derived by running the function it was given, rather than by independent computation. If the function is buggy, an oracle derived that way matches the bug, producing tests that pass against wrong behavior and thereby certifying the error. Check also for tests of only the paths the code obviously takes, which miss the empty vector or the missing value.

Verification. Check every expected value by hand or from an independent formula, not by running the function, since the oracle must come from outside the code under test. Confirm that the edge cases you care about, the empty input, the single value, the missing value, each have a test.

Prompt. 'Here is the data dictionary, not the data, for my cohort: `age_at_enroll` (integer, years), `sex` (character, coded `F` or `M` after cleaning), `bmi` (numeric, kg/m^2), `incident_dm` (integer, 0 or 1). Write `tinytest` data-validation tests for a data frame with these columns.'

Watch for. Check for generic checks, such as non-missing identifiers and numeric outcomes, that are reasonable but do not encode *your* domain. The

Draft, November 13, 2026. Content may change.

dictionary gives types and codes. It does not give the protocol's eligibility ages, the plausible range of BMI in your population, or how much outcome missingness the analysis can tolerate, so any such bound in the answer was supplied by the model rather than by your protocol.

Verification. Supply the domain constraints yourself, or check that the generated ranges match your data dictionary. A validation test asserting a plausible-looking but wrong range is worse than no test, because it lends false authority.

Prompt. 'Explain why this test is failing.'

Watch for. The model may recommend changing the test to match the current output, which makes the failure disappear without establishing whether the code or the test was wrong. A failing test is sometimes correct: it may be reporting a real bug.

Verification. Decide first, from the oracle, whether the code or the expectation is at fault, and only then change the one that is wrong. Never silence a failing test by editing its expected value to whatever the code currently returns; that converts a test into a tautology.

A better prompt. 'The attached `tinytest` file fails on the line shown, with this message. The expected value was computed by hand from the analysis plan. Do not edit the test file. List the possible causes in the function, and for each give an input I can run, with its hand-computed answer, that would confirm or rule it out.'

A coding agent with write access need not stop at a recommendation: asked to make the suite pass, it can make the edit itself, and the change arrives as an ordinary commit. Treat `inst/tinytest/` and `inst/baseline/` as protected paths, and read every line of any diff that touches them before it merges, as question 5 of Section 11.12 asks. What an agent may change and what stays with the analyst is set out in Section 2.10.1.

11.16. The cohort, continued

The chapter has introduced four kinds of test. This section applies all four to the running cohort compendium, using the canonical Cox analysis as the analysis under test.

Draft, November 13, 2026. Content may change.

The canonical analysis. The primary question is: does BMI predict time to incident diabetes after adjusting for age and sex? The model is a Cox proportional-hazards model fit to the cleaned extract, with BMI expressed per 5 kg/m^2 so that the hazard ratio has a clinically natural scale. The analysis runs clean only after `clean_cohort()` has repaired the four planted defects.

```
library(survival)
cohort_path <- file.path(
  '..', '..', 'analysis', 'data', 'raw_data',
  'cohort_raw.csv')
raw    <- load_cohort(cohort_path)
cohort <- clean_cohort(raw)
fit    <- fit_cox(cohort)
res    <- summarise_cox(fit)
```

The hazard ratio for BMI per 5 kg/m^2 is 1.83 (95% CI 1.32–2.52), based on 48 events from 337 cleaned rows. Uncleaned, the analysis would still run, but it would fit a different model to the 338-row raw extract, with the duplicate and the miscoded sex included.

Unit tests of `fit_cox()`. A unit test checks the contract of one function in isolation. For `fit_cox()` the contract is about which rows the model uses. `coxph()` drops every row with a missing value on any variable it reads, so the model uses the complete cases on `followup_days`, `incident_dm`, `bmi`, `age_at_enroll`, and `sex`, and its event count is the number of events among those cases, not the number of non-missing outcomes. The oracle is computed from the data directly rather than from the fit, so a change to the exclusion rule fails the test. The assertions below are in `inst/tinytest/test-fit-cox.R`.

```
model_vars <- c('followup_days', 'incident_dm', 'bmi',
                'age_at_enroll', 'sex')
complete <- stats::complete.cases(cohort[, model_vars])

expect_equal(fit$n, sum(complete),
  info = 'the model uses the complete cases on its
   ↪  variables')
```


Draft, November 13, 2026. Content may change.

```
expect_equal(fit$nevent,
             sum(cohort$incident_dm[complete] == 1),
  info = 'the event count is the events among complete
   ↪  cases')
expect_true(fit$nevent < sum(!is.na(cohort$incident_dm)),
  info = 'events are fewer than non-missing outcomes')
```

On the cleaned cohort the model uses 305 of the 337 rows, and its 48 events are the rows among them with `incident_dm` equal to 1. A count of non-missing outcomes, here 305, would count the zeros as events too. The same file also removes the age of one event row and checks that exactly one event and one row leave the model.

Integration test of the pipeline. An integration test checks that the pipeline stages compose correctly end to end: `load_cohort` feeds `clean_cohort` feeds `fit_cox` feeds `summarise_cox`, and the final result is a list with the expected names and no `NA` values. This listing and the two after it are illustrative and abridged, and the book's suite does not run them as written.

```
result <- run_canonical_analysis(cohort_path)
expect_true(all(c('n_rows', 'hr', 'lower',
                  'upper', 'n_events')
                %in% names(result)),
  info = 'canonical result has expected names')
expect_true(all(!is.na(unlist(result))),
  info = 'canonical result contains no NA values')
```

Data-validation tests. The planted pathologies give the data-validation suite something real to catch. The test file checks each defect before and after cleaning, so a change to `clean_cohort()` that accidentally fixes or breaks a defect fails a test instead of silently altering the baseline.

```
# Duplicate before cleaning, gone after
expect_equal(sum(duplicated(raw$mrn)), 1,
  info = 'raw extract has exactly one duplicated MRN')
expect_equal(sum(duplicated(cohort$mrn)), 0,
```


Draft, November 13, 2026. Content may change.

```
  info = 'cleaned cohort has no duplicated MRN')

# Implausible age before cleaning
expect_true(any(raw$age_at_enroll > 105, na.rm = TRUE),
  info = 'raw extract has at least one implausible age')
expect_true(all(is.na(cohort$age_at_enroll) |
                cohort$age_at_enroll <= 105),
  info = 'cleaned cohort has no age above 105')
```

Reproducibility (baseline) test. The baseline test is the L3 step. It reruns `run_canonical_analysis()` and compares the hazard ratio and confidence limits with values in `inst/baseline/cohort_baseline.rds`, within a relative tolerance of `1e-8`.

```
baseline <- readRDS(
  system.file('baseline', 'cohort_baseline.rds',
              package = 'reproducibility'))
result   <- run_canonical_analysis(cohort_path)

expect_equal(result$hr, baseline$hr,
  tolerance = 1e-8,
  info = 'Cox HR matches committed baseline')
expect_equal(result$lower, baseline$lower,
  tolerance = 1e-8,
  info = 'lower CI bound matches committed baseline')
```

A failure here means either the extract changed (check `shasum -a 256 cohort_raw.csv`), the `survival` package version changed (check `renv.lock`), the R version changed (check `sessionInfo()`), or the model specification changed (check the diff of `R/cohort.R`). A baseline that also records a Monte Carlo quantity, such as a bootstrap bound, has a fifth cause: the random-number generator changed (check `RNGkind()` against the recorded kind). The question there is whether the difference is within Monte Carlo error, which the second tier of `compare_to_baseline()` answers. Each cause has a different fix; the baseline test names the symptom and the diagnosis follows from the audit.

Draft, November 13, 2026. Content may change.

11.17. Exercises

1. Take a function you have written for a current analysis and write three `tinytest` unit tests for it: one ordinary case with a hand-computed oracle, and two edge cases. State, for each, where the expected value came from.

2. Introduce the `calculate_mean` bug of this chapter into a copy of one of your own summary functions, and confirm that the analysis still runs and produces a stable result. Then write the single unit test that would have caught it. Reflect in a sentence on why reproduction did not catch it.

3. Write a data-validation test file for a dataset you use, encoding at least four domain constraints. These are an expected set of columns, a plausible range for a continuous variable, an admissible set of labels for a categorical variable, and a bound on the missing-data rate. Justify each range from a data dictionary or from clinical knowledge.

4. Classify each test you wrote in exercises 1 and 3 as belonging to the capture register or the validation register, and state what removing it would and would not change about the analysis's position on the L0 to L3 ladder.

5. Write a reproducibility test of the baseline form: run a small analysis, save its result with `saveRDS`, and write a test that recomputes the result and compares it against the saved baseline with an appropriate tolerance. Then change one line of the analysis and observe the test fail.

6. Install the `precommit` package, add the `.pre-commit-config.yaml` of this chapter to a project, and attempt to commit deliberately misformatted R code. Report what the hook does, and contrast it with what continuous integration would do in Chapter 12 if the hook were absent.

7. On the running cohort compendium, run `load_cohort()` and `clean_cohort()` from `R/cohort.R` against `analysis/data/raw_data/cohort_raw.csv`, then write a data-validation test file that checks, against the real cleaned cohort, this chapter's four constraints: the expected columns, the enrollment age range, the two coded sex

Draft, November 13, 2026. Content may change.

values, and a bound on outcome missingness. Report three numbers a grading script could check without fitting any model: the raw row count, the cleaned row count, and the number of rows removed as duplicates.

8. This exercise returns to the committing discipline of the version-control chapter and habit 1, 'commit before you compute.' The baseline test of this chapter depends on a committed file, `inst/baseline/cohort_baseline.rds`, which is itself a recorded result. State which must be committed first, the analysis code that the recording script runs or the baseline file that running it produces, and explain what goes wrong for a collaborator who pulls the repository if the order is reversed.

9. **Paired review.** Work with a partner, each on your own copy of the cohort compendium. Treat the following as the cohort's analysis plan for cleaning: keep one row per `mrn`; parse both date formats in `enroll_date`; code `sex` as `F` or `M`; set ages above 105, the upper enrollment limit, to missing rather than dropping the row; and keep rows with a missing outcome. Without consulting `clean_cohort()`, each of you writes a cleaning function to this plan on a branch and opens a pull request. Then review your partner's pull request against the six questions in Section 11.12, leaving each finding as a review comment, and answer the comments on your own. Finally, run both functions on `cohort_raw.csv` and compare the two cleaned data frames. Report one discrepancy that review found and one that only the comparison found, or say that there was none.

11.18. Solutions

Attempt an exercise before expanding its solution. You perform these exercises on your own functions and data, so the code will vary; the boxes record the form the answer should take and the reasoning behind it.

Exercise 1

The form of the answer matters more than the function chosen. For a hypothetical `summarize_visits()`:

Draft, November 13, 2026. Content may change.

```
# inst/tinytest/test-summarize_visits.R
expect_equal(summarize_visits(c(2, 4, 6)), 4)
expect_true(is.nan(summarize_visits(numeric(0))))
expect_error(summarize_visits('a'))
```

Where each expected value came from is the substance of the exercise, and the three answers should be different.
The ordinary case comes from arithmetic done by hand, independently of the function. This is what makes it an oracle. If the expected value were computed by running the function, the test would assert only that the function agrees with itself, which is the commonest way a test suite becomes worthless.
The empty-input case comes from a decision about the contract. There is no arithmetic to do; the analyst must decide what the function should do with no data and then encode the decision. The test is as much a specification as a check.
The wrong-type case comes from the same contract, asserting that the function refuses input it cannot handle rather than returning something misleading.
If you cannot state where an expected value came from, you have usually written a change-detector rather than a test.

Exercise 2

The bug is the dropped division, so the function returns a sum where a mean is intended:

```
summarize_visits <- function(x) sum(x)   # bug
```

The analysis still runs, produces a number, and produces the *same* number on every machine and every rerun. If the project is containerized and its packages pinned, that number is perfectly reproducible.
The catching test is the ordinary case from exercise 1:

```
expect_equal(summarize_visits(c(2, 4, 6)), 4)
```

It fails immediately, returning 12 against an expected 4.

Draft, November 13, 2026. Content may change.

Why reproduction did not catch it. Because reproduction asks whether the same inputs give the same outputs, and they do. The buggy function is deterministic; nothing about it varies between machines or over time. Reproducibility machinery captures determinants and propagates whatever the code computes with mechanical fidelity, including the wrong thing. The only way to detect the error is to compare the output against a value obtained *independently of the code*. That is what the hand-computed oracle supplies, and what no amount of environment capture can.

Exercise 3

The four constraints, written for the running cohort in the form the chapter uses; your own dataset will have its own names and bounds:

Draft, November 13, 2026. Content may change.

```
# inst/tinytest/test-data-cohort.R
raw <- load_cohort('../../analysis/data/raw_data/cohor
 ↪  t_raw.csv')
cohort <- clean_cohort(raw)

expect_true(
  all(c('mrn', 'age_at_enroll', 'sex', 'incident_dm')
   ↪  %in%
        names(cohort)),
  info = 'the expected columns are present'
)
expect_true(
  all(cohort$age_at_enroll >= 18 &
        cohort$age_at_enroll <= 105, na.rm = TRUE),
  info = 'age lies within the enrollment criteria'
)
expect_true(
  all(cohort$sex %in% c('F', 'M')),
  info = 'sex takes only its two coded levels'
)
expect_true(
  mean(is.na(cohort$incident_dm)) < 0.15,
  info = 'outcome missingness stays under fifteen
   ↪  percent'
)
```

Each justification should come from outside the data.
The **column set** comes from the data dictionary: these are the variables the analysis consumes, and a missing one means an upstream export changed. The **age range** comes from the protocol's eligibility criteria, not from the observed minimum and maximum. Deriving the bound from the data would make the test unable to detect the very excursion it exists to catch. The **label set** comes from the coding sheet. The **missingness bound** comes from what the analysis can tolerate before its conclusions become unsafe, which is a judgment the analyst must make and record.

Exercise 4

All of them belong to the validation register. Unit tests and data-validation tests alike pin no determinant: they add no package to the lockfile and no library to the image. They confirm that what was captured behaves as intended.

What removing them would change. Nothing about the position on the L0 to L3 ladder. An analysis at L2 with a full test suite and the same analysis at L2 with no tests sit on the same rung. The rung grades how far down the determinant stack the result is locked, and tests lock nothing.

What removing them would change instead is whether anyone has reason to believe the result is right. The analysis becomes trusted rather than checked, and the class of error the tests were catching, a wrong function and an out-of-range dataset, returns to being invisible.

One qualification concerns L3. L3 requires that recorded outputs be shown to regenerate within a stated tolerance from a clean rebuild, and the evidence for that is a baseline test. The unit and data-validation tests of exercises 1 and 3 are not baseline tests, so removing them leaves L3 where it was. Removing the baseline test, by contrast, would drop a project from L3 to L2, because the one piece of evidence the rung requires would be gone. It is the *verification role* that matters there, not the testing role.

Exercise 5

```
# once, from the project root, to record the baseline
cohort <- clean_cohort(
  load_cohort('analysis/data/raw_data/cohort_raw.csv')
)
saveRDS(coef(fit_model(cohort)),
        'inst/tinytest/baseline-coefs.rds')
```


Draft, November 13, 2026. Content may change.

```
# inst/tinytest/test-baseline-coefs.R
data_file <-
 ↪  '../../analysis/data/raw_data/cohort_raw.csv'
if (!file.exists(data_file)) exit_file('no analysis
 ↪  data')
baseline <- readRDS('baseline-coefs.rds')
cohort <- clean_cohort(load_cohort(data_file))
current <- coef(fit_model(cohort))
expect_equal(current, baseline, tolerance = 1e-8,
             info = 'model coefficients match the
              ↪  baseline')
```

The model fit draws no random numbers, so no seed is needed; a stochastic quantity would take its seed as an argument, as `bootstrap_ci()` does in the chapter. The file name begins `test-baseline`, the convention by which a verification step finds baseline tests. Changing one line of the analysis makes the test fail, reporting the coefficients that moved.

The tolerance is the part of this exercise that needs thought. An exact comparison is the strongest check, and it will produce failures that mean nothing. Floating-point summation reorders under a different BLAS or a different thread count, as Chapter 7 discusses, and the last digits move without the analysis changing. Two different tolerances are in play, and you should name each. The *numerical* tolerance, `1e-8` relative here, accepts that floating-point noise and still catches any change in the fitted model large enough to alter a reported figure. The *reporting* tolerance is the precision at which results are published; for a Monte Carlo quantity it is set by the MCSE, as the second tier of the chapter's baseline check is. Because these coefficients draw no random numbers, only the numerical tolerance applies to them. A numerical tolerance set loosely, near the reporting precision, stops detecting real changes.

What this test is and is not. It is the kind of baseline test L3 requires, and it is a *change detector*. It asserts that today's answer matches the recorded one, not that either is correct, because the baseline was produced by the same code it now guards. If the code was wrong when the baseline was recorded, this test will faithfully

defend the wrong answer. That is the paradox reappearing one level up. The unit tests of exercise 1 are what establish correctness; this test establishes stability.

Exercise 6

What the hook does. The commit is refused. The formatting hook rewrites the offending files in place and aborts, so the commit does not complete and the working tree now contains the reformatted code, which must be staged and committed again. The effect is that misformatted code cannot enter the history at all.

The contrast with continuous integration. CI would let the commit through. The code enters the history, the push succeeds, and the check runs afterward on the server, reporting a failure against a commit that already exists. Fixing it requires a second commit, so the history now records both the violation and its repair.

The differences worth naming are three. **Timing:** the hook acts before the record is written, CI after. **Locality:** the hook runs on the analyst's machine and can therefore be skipped, with `--no-verify` or by a collaborator who never installed it, whereas CI runs on the server and cannot. **Scope:** the hook must be fast enough to sit inside a commit, so it does cheap things like formatting and linting, whereas CI can rebuild the container and run the full suite.

The two are complements rather than alternatives, and the chapter's framing explains why. The hook is a convenience that keeps trivia out of the history. CI is the gate that actually guarantees anything, because it is the one nobody can skip.

Exercise 7

The solution for this exercise is in the instructor guide, which is distributed separately to course instructors. Self-study readers who want to check their answer can compare the three autogradable values against those reported in the chapter's own 'The cohort, continued' section.

Draft, November 13, 2026. Content may change.

Exercise 8

The code must be committed first, then the recording script is run against that committed code, and the baseline file it writes is committed second, in its own commit. This is habit 1 applied to a derived artifact rather than to the primary analysis: a committed file must always be traceable to the committed code that produced it.

Reversing the order breaks that traceability. Suppose the baseline is regenerated and committed while the code change that motivated it is still uncommitted. A collaborator who pulls the repository then receives the new baseline paired with the *old* code, because the code change never reached the shared history. Their copy of the baseline test then fails immediately, not because anything is wrong with their checkout, but because the repository itself never recorded a consistent pair of code and baseline. Worse, if the local, uncommitted code change is later lost, the committed baseline becomes orphaned. It is then a recorded result with no committed code that anyone can point to as having produced it, which is exactly the provenance gap habit 1 exists to close.

Exercise 9

Your functions and findings will differ; the answer takes the form of the review comments, the replies, and the comparison. Four places in this plan repay a reviewer's attention, because each is where your two functions can diverge while both run without error.

Ages. The plan sets an implausible age to missing. A function that drops the row instead changes the row count, and the change appears only if you count rows before and after. This is question 3.

Missing outcomes. The plan keeps rows whose outcome is missing, because dropping them is an analytic decision that the model step makes. A cleaning function that drops them has moved that decision into the wrong place.

Duplicates. 'One row per `mrn`' does not say which copy to keep. Check whether the duplicated rows agree on every column. If they do not, the plan is ambiguous, and the fix belongs in the plan, recorded and dated, not in whichever choice each of you happened to code.

Draft, November 13, 2026. Content may change.

Coding and parsing. Look at every distinct value of `sex` and every format in `enroll_date` in the raw file before deciding that the plan's rules cover them. A value the rules do not name should become missing by a stated rule, not by accident.
The comparison is double programming in miniature. Use `all.equal()` on the two cleaned data frames, after sorting both by `mrn`, and read every difference it reports. Review is suited to the departures from the plan that are visible in the code, such as a dropped row; the comparison is suited to the slips that read correctly, such as a date format parsed in the wrong order. If the two functions agree, that is evidence for both, but not proof: you may have read the same sentence of the plan the same wrong way, as the section on double programming warns.

Answers to the checks

Check 1. *from earlier chapters* (page 428)

Of the three failures in this warm-up, (a) is the package-version layer. With no lockfile the project has not reached L1, and the remedy is the L1 step of Chapter 6: a lockfile written by `renv::snapshot()` and read by `renv::restore()`. (b) is the operating-system and system-library layer. A lockfile pins the R packages and nothing beneath them, and GDAL is not an R package. The remedy is the container of Chapter 7, which reaches L2 only with a digest-pinned base, a dated snapshot, and the lockfile restored inside it. (c) is a stale derived file, not an environment layer at all: the correction never reached the figure. The remedy is a pipeline in which the raw extract is a `format = 'file'` target, so that `targets` rebuilds everything downstream of it (Chapter 10).

Check 2. *the paradox* (page 431)

L2 is a claim about capture, that the packages and the environment are pinned so the same code produces the same output everywhere. It is not a claim about correctness: a pinned, containerized analysis still reproduces a wrong hazard ratio exactly, if the code that computes it is wrong. Only

Draft, November 13, 2026. Content may change.

a test, comparing against an independent oracle, bears on whether the number is right.

Check 3. *three judgments* (page 433)

The first is what 'correct' means for a given input, supplied by an oracle outside the code under test. The second is which behaviors of a function constitute its contract and so deserve a test. The third is which domain constraints, such as a plausible age range, the data itself must satisfy.

Check 4. *capture, faithfully* (page 434)

Capture preserves the error exactly: every reproduction on every pinned machine returns the same sum, forever. It does nothing to detect the error, because capture examines only whether the mapping from input to output is held constant, never whether the mapping is the intended one. Only a test, comparing against an independent oracle, examines the mapping itself.

Check 5. *two oracles* (page 435)

No. Reproducing exactly and passing every unit test shows that the code is a faithful, bug-free implementation of whatever analysis it runs. Neither check examines whether that analysis is the one the study committed to before seeing the data. The additional check is whether the fitted model, the endpoint, and the comparison match a pre-specified analysis plan dated before the data arrived. That plan is the oracle: a source of what the correct analysis should look like, supplied independently of any result the code could produce.

Check 6. *reading a bare file* (page 436)

Because each expectation is a bare, top-level statement, the file is an ordinary runnable R script. An analyst can `source()` the whole file, or send a single line to the console, and watch that one expectation evaluate immediately, with no need to locate it inside a nested block first.

Check 7. *two kinds of instability* (page 450)

In size, largely yes; for the seed-exact check, no. The seed-to-seed change at fixed `B` is Monte Carlo error, and raising `B` shrinks it as $1/\sqrt{B}$. A new sampler gives a given seed different resamples, which in effect draws the bound at a new seed, so raising `B` shrinks that change at the same rate. The

Draft, November 13, 2026. Content may change.

two samplers still disagree in the last digits at any `B`, so the seed-exact tier keeps failing. Pinning the sampler restores seed-exact agreement; the second tier, within three MCSE, confirms that the statistical result reproduced without it.

Check 8. *which test catches it?* (page 453)

(a) A data-validation test asserting the expected sample size, or an integration test that asserts the cleaned row count. The integration test shown earlier checks only the shape of the model summary, so it would need that line added. The book's baseline test would also catch it, because it compares the cleaned row count with the recorded one, though it was not written for that purpose. (b) A data-validation test on the age range.

(b) A unit test against a hand-computed standard deviation. (d) A baseline test comparing the seeded result with a value recorded before the upgrade and committed, which is the comparison L3 requires; a same-session test would miss it. Note that the unit and data-validation tests catch defects that reproduction alone would happily propagate.

Check 9. *fragile or robust* (page 454)

It is an implementation test: it pins a detail of how `my_clean` is built, an internal counter, rather than what it does. A rewrite that preserves the function's behavior but removes or renames that counter breaks the test for no good reason, training the analyst to distrust or delete it.

Check 10. *capture or validation?* (page 456)

(a) Capture: it fixes the package versions; removing it drops the analysis from L1 to L0. (b) Validation: it checks that a captured function is correct; removing it leaves the analysis at the same level but now trusted rather than checked. (c) Validation: it checks that code is consistently formatted; removing it captures nothing less but permits legibility to decay. (d) Capture: it fixes the environment; removing it drops the analysis from L2 to L1. The rule of thumb is that removing a capture feature lowers the level, while removing a validation feature leaves the level unchanged and only stops the checking. There is one exception: the baseline test is the evidence L3 requires, so removing it drops an analysis from L3 to L2.

Draft, November 13, 2026. Content may change.

Check 11. *three validations* (page 458)

All three are real evidence, but none is computer system validation by itself, and the three together are not automatically enough either. The lockfile is capture; the test suite is software validation in this chapter's sense; the risk assessment is a package-level trust judgment. A validated state is defined by the sponsor's procedures and required by the predicate rules, not by these artifacts. It concerns the environment as a whole across its lifecycle, maintained through a change-control process. Under qualification practice that means installation, operational, and performance qualification records, and sign-off on each subsequent change; 11.10(a) itself names none of these. It is not met by a snapshot alone of the code, the tests, and one package's status at a single point in time.

Check 12. *shifting weight* (page 460)

Data-validation tests and reproducibility (baseline) tests. With few reusable functions, the unit-test surface is small. The risk therefore shifts to the two places such a compendium still has: the data it reads, which may be malformed, and the result it reports, which must regenerate. Those are exactly what data-validation and baseline tests check.

Check 13. *the oracle in the table* (page 463)

Because an oracle has to come from outside the code under test. Computing the expected values by calling `calculate_mean()` would make the check compare the function against itself. It would pass identically whether `calculate_mean` were correct or carried the dropped-division bug, since both forms are internally consistent with their own output. The hand-written vector is what lets the table show the buggy column failing every row.

11.19. Cheat sheet

Command or rule	What it does
`expect_equal(a, e)`	Compares `a` to oracle `e` with `all.equal()`'s default tolerance.

Draft, November 13, 2026. Content may change.

Command or rule	What it does
`expect_equal(a, e, tolerance = 1e-8)`	The baseline-test comparison: a stated, not default, tolerance (habit 5).
`expect_true()`, `expect_false()`, `expect_identical()`, `expect_error()`, `expect_inherits()`	The common `tinytest` expectations beyond equality.
`tinytest::test_package('pkg')`	Runs the suite under `inst/tinytest/` via the `tests/tinytest.R` runner.
`exit_file('message')`	Exits a test file early, for a test that needs data `R CMD check` does not install.
`make lint` / `make style`	Run `lintr` / `styler` inside the project container.
`test-baseline*.R` naming	The convention `zzc verify --full` looks for to certify L3.
Rule of thumb	Test behavior, not implementation; a test should fail only when the contract breaks.
Rule of thumb	An oracle must be computed independently of the code under test, never by calling it.
Habit 4	Make every gate able to fail, and watch it fail once.
Habit 5	Compare against a recorded baseline, with a stated tolerance.
Code review, six questions (Section 11.12)	Output traces to code; raw inputs; the plan's exclusions; seed at the call; tests changed with a reason; clean log

Draft, November 13, 2026. Content may change.

11.20. Further reading

The single most important reference for this chapter is the short note by Leek and Peng, that reproducible research can still be wrong (Leek & Peng, 2015). It states the paradox that motivates every test in a compendium, and it argues for a prevention-oriented rather than a correction-oriented stance toward analytical error.

For the testing tool itself, van der Loo's documentation of `tinytest` (Loo, 2020) explains the design rationale for a zero-dependency, bare-expectation framework. It is worth reading for the argument that a testing framework should not itself be a heavy dependency.

For the practice of testing and code quality as components of scientific computing, the two papers by Wilson and colleagues are the standard references. They are the best-practices paper (Wilson et al., 2014) and the good-enough-practices paper (Wilson et al., 2017), the latter pitched deliberately at researchers without a software background. Sandve and colleagues' ten simple rules (Sandve et al., 2013) cover the same ground in a compact, rule-by-rule form, and situate version control and recorded results within a general reproducibility discipline.

The connection between testing and the highest rung of the ladder is taken up in Chapter 12, where the reproducibility test of this chapter reappears at the scale of the whole compendium as output verification. Chapter 3 gives the capture-and-validation distinction that places testing in the validation register. Peng's early account of reproducibility as a spectrum (Peng, 2011) remains the clearest short statement of why reproducibility can serve as a minimum scientific standard when full independent replication is not possible.

Draft, November 13, 2026. Content may change.

12. Continuous Integration and Verification

Provide Public Access to Scripts, Runs, and Results.

Sandve, Nekrutenko, Taylor, and Hovig, *Ten Simple Rules for Reproducible Computational Research*, Rule 10 (2013)

A lockfile that has not been rebuilt since it was committed, and a Dockerfile that has not been rebuilt since it was written, are untested claims. By the end of this chapter you will be able to look at a green build and say exactly what it has shown: which of three automated gates ran, what each one would have caught had it been absent, and what a missing baseline test still lets through unnoticed when every gate is green. You will also know how to turn each gate red on purpose, because a check that has never been seen to fail has not been shown to work.

Before you start

Reading time. About 70 minutes.
Prerequisites. Chapter 11 (Chapter 11) supplies `tinytest`, the four kinds of test, and the baseline test this chapter's verification gate runs. Chapter 3 (Chapter 3) supplies the L0-L3 ladder and the capture-versus-validation vocabulary this chapter makes operational. Chapter 6 (Chapter 6) supplies the dependency invariant the dependency-manifest gate automates, and Chapter 7 (Chapter 7) supplies the Dockerfile the verify job rebuilds. The R knowledge assumed is writing and calling a function, together with enough comfort reading YAML to follow a workflow file line by line; no YAML is written from scratch here.
Objectives. By the end of this chapter, can you:

- explain why a CI runner, carrying none of the author's undeclared state, catches a 'works on my machine' failure that no test run on the author's own machine can?
- name the three automated gates a CI workflow runs, state what each one proves, and demonstrate how to make each one fail on purpose?
- say why verification (L3) compares computational outputs against a committed baseline, within a stated tolerance, rather than comparing rendered PDFs or HTML?
- read and extend a checksum manifest that guards the integrity of raw input data?
- match the three gates, plus verification, to a compendium's archetype, so that a package, a manuscript, and a simulation each run only the checks appropriate to them?

Outline. We begin with where CI sits on the stack, a captured environment that nobody has yet rebuilt, and the analyst's three irreplaceable judgments. We then turn to the machine that is not yours, to the three gates and where their bugs hide, to secrets, tokens, and pinned actions, and to two CI dialects. The middle of the chapter treats verification and level L3, the checksum manifest, the framework's own capture test matrix, and how CI adapts to five archetypes. We then consider independent external verification, a worked example, debugging a failed reproduction, and a gate demonstrated on this book's own cohort compendium. Lastly, we discuss collaborating with an LLM, followed by exercises, a cheat sheet, and further reading.

Check 1: from earlier chapters

Two colleagues rerun a compendium in the same digest-pinned image, built from a complete lockfile. Colleague A mounts the project without its `.Rprofile`, where the author had put `RNGkind(sample.kind = 'Rounding')` and `set.seed(2024)`, and the bootstrap interval moves. Colleague B, on a different processor, obtains regression coefficients that agree with the author's except in their last few digits. For each discrepancy, name the determinant layer behind it, say whether any rung of the ladder closes it, and give the earlier remedy.

Draft, November 13, 2026. Content may change.

The answer is on page 537.

12.1. Where this sits on the stack

Continuous integration pins nothing, and Figure 12.1 marks that explicitly. The three layers its gates touch (package versions, analysis code, and input data) are validated here, not captured, because the lockfile, the scripts, and the raw extract were already pinned or written by the chapters that came before. This chapter's whole contribution is confirming, on a machine that is not the author's, that those captures still hold.

```
plot_determinant_stack(
  validates = c('Package versions', 'Analysis code', 'Input
   ↪  data'),
  title = 'CI captures nothing; it validates three layers
   ↪  already captured')
```


Draft, November 13, 2026. Content may change.

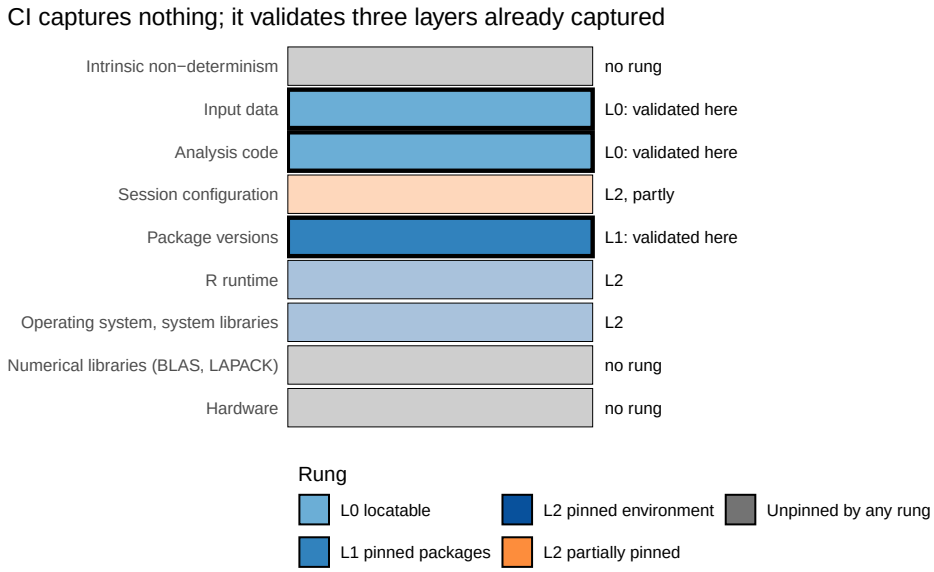


Figure 12.1.: The determinant stack of Chapter 3, with package versions, analysis code, and input data marked as validated rather than captured. CI pins no new layer: the dependency-manifest gate confirms packages are still pinned, `R CMD check` and the render gate confirm the analysis code still runs, and the checksum manifest confirms the input data is unchanged, which applies the capture-versus-validation distinction to the three gates of this chapter.

12.2. Orientation

Consider a registry study whose analysis is finished, whose manuscript is written, and whose lockfile and container recipe are committed to version control. The author has done, to all appearances, everything this book has asked. The packages are pinned (Chapter 6), the environment is fixed in an image (Chapter 7), the report is literate and regenerates its own figures (Chapter 9). The compendium sits, by the author's account, at level L2, a pinned environment. But how does the author know that the captured environment actually rebuilds, on a machine that is not the one

Draft, November 13, 2026. Content may change.

where the work was done? Until something tests it, the author does not know. A lockfile can reference a package version that has been pulled from its repository. A Dockerfile can depend on a base image whose digest has drifted. A script can silently rely on a file that lives only in the author's home directory and was never committed. None of these announce themselves on the machine where the work already runs, because on that machine the missing piece is present.

In this chapter we take up the instrument that closes the gap between a reproducibility claim and its demonstration. Continuous integration, abbreviated CI, is a service that rebuilds the compendium from its committed inputs on every change. It works in a fresh environment that carries none of the developer's accumulated and undeclared state, and runs a battery of checks against the result. We shall see that CI captures no new layer of the determinant stack. Its whole contribution is to *validate* that the layers already captured actually suffice. The distinction between capturing reproducibility and validating it was drawn in Chapter 1 and carried through every chapter. It reaches its operational form here: pinning is capture, and CI, together with the verification command it runs, is validation.

We then take the ladder to its top rung. Verification, the L2-to-L3 transition, confirms what has been captured by regenerating the computational outputs. It compares them, number for number and within a stated tolerance, against a recorded baseline. The subtlety, which we shall press, is in what gets compared. We compare a derived dataset, a fitted model object, or a results table, and emphatically not the rendered PDF or HTML, whose bytes carry timestamps and font identifiers that defeat a naive comparison. We close with the schemes by which reproducibility is checked from outside the author's own project entirely, during peer review.

Check 2: what CI does not know

An author claims their compendium is at level L2 because a `Dockerfile` is committed and builds on their laptop. Does a green CI run of that same `Dockerfile` raise the compendium's level? What, precisely, does the green run add?
The answer is on page 537.

Draft, November 13, 2026. Content may change.

12.3. The analyst's contribution

Before turning to the machinery, we note what it cannot do. Continuous integration is highly automated, but copying a workflow file into a repository and forgetting it leaves three decisions unmade. They remain the analyst's, and no tool makes them.

1. **The analyst decides what counts as a correct output, and records the baseline.** Verification can compare a regenerated result against a recorded one, but it cannot know which results matter or what tolerance is acceptable. The baseline may be a derived dataset, a coefficient table, or a single summary statistic. The agreement required may be to the last bit or to a stated numerical tolerance. Both are scientific choices the analyst must make and defend.

2. **The analyst decides which gates the compendium's archetype warrants.** A CI workflow can run `R CMD check`, a render gate, and a dependency-manifest gate, but not every archetype needs all three. An R-package archetype has no report to render; a simulation archetype may run a reduced pipeline under CI and the full one elsewhere. Matching the gates to the archetype is the analyst's call. The aim is that CI tests what actually needs testing, without imposing a render gate on a project that has nothing to render.

3. **The analyst reads a CI failure as evidence.** A red build is CI doing its job: it has found, on a clean machine, a dependency the author's machine supplied silently. The discipline is to treat the failure as a true report about the compendium's reproducibility and to fix the captured inputs rather than the check.

> The vocabulary of this chapter
>
> - **Continuous integration (CI).** A hosted service that, on every committed change to a repository, checks out the code onto a fresh virtual machine, rebuilds the project, and runs a defined set of checks, reporting success or failure automatically.
> - **Workflow.** The file, written in YAML, that tells the CI service what to do: which events trigger it, what environment to run

Draft, November 13, 2026. Content may change.

in, and what commands to execute. On GitHub these files live under `.github/workflows/`; on GitLab the single file is `.gitlab-ci.yml`.

- **Runner.** The fresh, ephemeral machine (usually a virtual machine or a container) on which the CI service executes the workflow. It is discarded after the run, so nothing persists between runs except what the workflow explicitly restores.
- **`R CMD check`.** R's own quality gate for a package, which among many checks verifies that every dependency declared in `DESCRIPTION` is installed and findable. A research compendium built as an R package inherits this gate for free, for its package skin only: `analysis/` is excluded from the build and never checked.
- **Render gate.** A CI check that re-renders the compendium's report (an R Markdown or Quarto document) from source and fails the build if the render errors.
- **Dependency-manifest gate.** A CI check that enforces the dependency invariant of Chapter 6. In its strict, R-package form it requires every package the code uses to be declared in `DESCRIPTION` and pinned in `renv.lock`. A research compendium's `DESCRIPTION` declares only the API exported from `R/`, while its analysis scripts legitimately draw on packages present in `renv.lock` but not in `DESCRIPTION`. For a compendium, therefore, the gate enforces the weaker inclusion that every package the code uses is pinned in `renv.lock`.
- **Verification.** The act of rebuilding the compendium's environment from its recipe and re-running the analysis in it. The regenerated computational outputs are then compared with a committed baseline, within a stated tolerance, by means of a baseline test (Chapter 11); passing it is what the ladder calls level L3.
- **Checksum.** A short fingerprint, computed from a file's or an object's bytes by a hash function such as SHA-256, with the property that any change to the input, however small, changes the fingerprint. A stored checksum lets a later run detect that an input has been altered.
- **Forge.** The platform that hosts the repository and runs its CI,

Draft, November 13, 2026. Content may change.

such as GitHub or GitLab. The forge governs where the code lives and which CI dialect the workflow is written in; it pins nothing about the computational environment.

Check 3: the analyst's irreplaceable judgments

Name the one of the three analyst judgments that a workflow file, however carefully written, cannot make for itself, and say why.
The answer is on page 537.

12.4. The machine that is not yours

We begin with the reason a second machine is needed at all. An analysis reproduces on the machine where it was written partly because the machine is generous. It supplies, without being asked, a package installed months ago and never recorded, or a system library that some other project needed. It may equally supply a data file sitting in a home directory, or an environment variable set once in a shell profile. Every one of these is undeclared state, and every one of them is invisible precisely because it is present. The 'works on my machine' report is true: it describes a machine that happens to carry more than the compendium declares.

Continuous integration removes the generosity. When a change is pushed, the CI service provisions a fresh runner, a virtual machine or container that has just been created and carries nothing but a clean operating system. It checks out exactly what is under version control, no more, and attempts to rebuild the compendium from those committed inputs alone. If the analysis depended on anything the author never declared, the rebuild fails. It fails here, rather than in a reviewer's hands or a colleague's re-analysis a year on. The CI machine catches the 'works on my machine' failure at its source for a reason that is almost tautological: the CI machine is, by construction, not the author's. It cannot supply the author's undeclared state because it does not have it.

Continuous integration does not pin a package version, fix an operating system, or record a session option; those are the work of the lockfile and the container, done before CI runs. What CI adds is the repeated, adversarial

Draft, November 13, 2026. Content may change.

test that the capture already performed is sufficient. A captured-but-never-retested compendium suffers a slow decay. In this failure mode a declared environment silently stops rebuilding as its external references drift, and no one notices because no one has tried to rebuild it. CI is the standing test that would notice. Removing CI from a repository lowers no level on the ladder; the compendium is exactly as reconstructable as before. It simply stops being checked, and becomes trusted rather than tested.

Check 4: why the runner is generous nowhere

Explain, in one or two sentences, why a freshly provisioned CI runner cannot supply the undeclared state that makes an analysis 'work on my machine', and why this is close to a tautology rather than a clever design feature.
The answer is on page 538.

12.5. What the checks are: the three gates

A CI run on a research compendium executes, in the general case, three gates. We take them in turn, because the division of labor among them is exact.

The first is **`R CMD check`**. Because a research compendium is structured as an R package (Chapter 5), it inherits R's own quality gate at no additional cost. Among its many checks, the one that matters most for reproducibility is early and simple. It verifies that every package the code in `R/` uses is declared in `DESCRIPTION`, and that every declared package is installed and findable. On a fresh runner this is a real test, because the runner has only what the lockfile declared.

Its scope is narrower than it looks. The check runs on the package that `R CMD build` makes, and a compendium's `.Rbuildignore` excludes `analysis/`. The analysis scripts are therefore never parsed, never run, and never scanned for packages (Chapter 5). `R CMD check` validates the package skin, `R/`, the tests, and `DESCRIPTION`, and nothing else. Within that scope there is a second trap. An undeclared package used in `R/` is reported as a WARNING, not an ERROR, so the gate fails on it only if it is told to fail on warnings. A workflow that passes `error_on = 'error'`

Draft, November 13, 2026. Content may change.

to `rcmdcheck`, as many hand-written workflows do, lets the undeclared dependency through with a green build.

The second is the **render gate**. Where the compendium carries a report, an R Markdown or Quarto document that regenerates its own figures and tables, the workflow re-renders it from source on the runner. A render that errors, because a chunk references an object that no longer exists or a package that was not restored, fails the build. Of the three gates it is the one that executes the analysis code: a report that sources the scripts under `analysis/`, or calls the functions they define, runs them on the runner, against the restored library and nothing else. The render gate is the literate document's validation counterpart: Chapter 9 seals the prose to the code, and the render gate confirms, on a clean machine, that the seal holds. It needs the data on the runner. The cohort this book uses is synthetic and committed for that reason. Where the real data are restricted, CI renders against a synthetic stand-in, and the real render runs inside the environment approved for the data. Committing the data to turn a red render gate green is never the fix.

The third is the **dependency-manifest gate**, the automated form of the dependency invariant developed in Chapter 6. In its strict form, appropriate to the R-package archetype, the invariant is a double inclusion. The set of packages the code actually uses must be a subset of the set declared in `DESCRIPTION`, which must in turn be a subset of the set pinned in `renv.lock`. A research compendium is looser. Its `DESCRIPTION` declares only the API exported from `R/`, while its analysis scripts legitimately use packages that appear in `renv.lock` but never in `DESCRIPTION`. For a compendium the operative invariant is therefore the single inclusion that every package the code uses is pinned in `renv.lock`, and the strict `DESCRIPTION`-inclusion gate does not apply. The gate scans the source for `library()`, `require()`, and `pkg::fun()` calls. It then reads `DESCRIPTION` and `renv.lock`, and stops with an error if a used package is not pinned. Because it reads the analysis scripts, it is the gate that covers what `R CMD check` cannot see. Its core needs no special tool; `renv` itself can find the packages the code uses. The minimal version below scans only `analysis/`, the part `R CMD check` never reads:

```
# ci/dependency-gate.R
used <- unique(renv::dependencies('analysis')$Package)
```


Draft, November 13, 2026. Content may change.

```
locked <- names(renv::lockfile_read('renv.lock')$Packages)
base_pkgs <- rownames(installed.packages(priority =
 ↪  'base'))
missing <- setdiff(used, c(locked, base_pkgs))
if (length(missing) > 0) {
  stop('Used in analysis/ but not pinned in renv.lock: ',
       toString(missing))
}
```

A companion tool, `zzrenvcheck`, does the same scan with more care (tests, vignettes, version constraints, and the stricter `DESCRIPTION` inclusion for the package archetype). Its `error_on_fail` argument decides whether a failed check stops the step with a nonzero exit status or only prints a report. The default is `!interactive()`, so a call from `Rscript` already stops, but CI should pass `error_on_fail = TRUE` explicitly so that the gate does not depend on how R was started. It is not on CRAN, so the workflow installs it from its source repository, pinned to a commit. Either form catches the most common drift: a package installed interactively during analysis, used in the code, and never snapshotted, so that the lockfile is silently incomplete.

How do the three gates fit together in practice? A condensed GitHub Actions workflow that runs them on every push and pull request illustrates the shape of the thing. It uses the R community's `r-lib/actions` for the setup and the check, at the major versions current when this chapter was revised in November 2026 (`actions/checkout@v7`, `r-lib/actions/*@v2`). We show it as a static file. Its YAML was checked for syntax, and the command in each gate was run against a toy compendium. However, the workflow as a whole is executed by the CI service, never by this book.

```
name: compendium

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]
```

```
permissions:
  contents: read

jobs:
  gates:
    runs-on: ubuntu-24.04
    env:
      GITHUB_PAT: ${{ secrets.GITHUB_TOKEN }}
    steps:
      - uses: actions/checkout@v7

      # Install the R version recorded in renv.lock.
      - uses: r-lib/actions/setup-r@v2
        with:
          r-version: renv
          use-public-rspm: true

      - uses: r-lib/actions/setup-pandoc@v2

      # Restore renv.lock into the project library. A
      # stock renv .Rprofile activates that library in
      # every later R step run from the repository root.
      - uses: r-lib/actions/setup-renv@v2

      # The gate tools are not analysis dependencies, so
      # they are not in renv.lock. zzrenvcheck comes from
      # a personal repository and is pinned to a commit.
      - name: Install the gate tools
        shell: Rscript {0}
        run: |
          renv::install(c('rcmdcheck', 'remotes',
↪   'pkgload'))
          remotes::install_github(
            'rgt47/zzrenvcheck',
            ref =
↪   '6f53235668cb4048c1ae793d64b10d6505bf69a2')
```


Draft, November 13, 2026. Content may change.

```
    # Gate 1: R CMD check of the package skin. Any
    # WARNING, such as an undeclared import, fails.
    - uses: r-lib/actions/check-r-package@v2
      with:
        args: 'c("--no-manual")'
        error-on: '"warning"'

    # Gate 2: every package the code uses, analysis/
    # included, is declared and pinned.
    - name: Dependency-manifest gate
      shell: Rscript {0}
      run: |
        zzrenvcheck::check_packages(strict = TRUE,
                                    error_on_fail = TRUE)

    # Gate 3: the report re-renders from source.
    - name: Render gate
      shell: Rscript {0}
      run: |
        rmarkdown::render('analysis/report/report.Rmd',
                          output_format =
↪  'html_document',
                          output_dir = tempdir())
```

Three choices in the file merit a word. First, `error-on` is set to `"warning"` explicitly, although that is also the action's default. The setting is the whole difference between a gate that catches an undeclared import and one that does not. Second, the render gate renders to HTML into a temporary folder. It tests that every chunk runs, and leaves PDF typesetting, which needs a LaTeX installation, to a separate step if the deliverable is a PDF. Third, nothing in the file is marked `continue-on-error`; a step that is allowed to fail without failing the job is a report, not a gate.

The house rule of this book is that a gate is not shown to work until it has been seen to fail. Each of the three can be turned red on purpose, in a branch, in a few minutes:

Gate	Deliberate breakage	What the run reports
`R CMD check`	Call `sandwich::vcovHC()` in a function in `R/` without adding `sandwich` to `DESCRIPTION`	A WARNING, '`::` or `:::` import not declared from', which `error-on: '"warning"'` turns into a failed step
Dependency manifest	Add `library(sandwich)` to a script in `analysis/` without snapshotting	The missing package is listed and the step stops with an error
Render	Add a chunk that loads a package absent from `renv.lock`, or uses an object no chunk creates	`rmarkdown::render()` stops at that chunk and names it

We ran each breakage against a toy compendium (R 4.6.1, `rcmdcheck` 1.4.0, `zzrenvcheck` from its source repository in October 2026), and each command exited with a nonzero status. The first breakage also shows why the setting matters: with `error_on = 'error'` the same check returned normally, the WARNING printed and ignored. Placing `library(sandwich)` in `analysis/` instead of `R/` shows the scope of the first gate. In that case `R CMD check` passes and the dependency gate goes red; the render gate goes red too if the report runs that script.

12.5.1. Where CI bugs hide: cross-boundary state

The workflow above takes care over where the pinned library is restored and from which directory each R step starts. That care avoids a subtle bug whose cause generalizes well beyond the particular case.

The trouble is that a CI run is not one continuous process. It is a sequence of separate steps, each a fresh shell, and within a step the check itself runs in a further separate subprocess. State does not automatically cross these boundaries. In one documented case, the step that restored the

Draft, November 13, 2026. Content may change.

pinned packages installed them correctly. The very next step, running `R CMD check`, then reported a required package as 'not available'. The package was on disk; the check simply could not see where. Two boundaries conspired. The restore step adjusted the library path in its own shell, but the next step is a fresh shell and inherited none of that. And even within a step, `R CMD check` runs in a child subprocess whose library path must be passed to it explicitly. Adjusting the path in the calling session does not propagate to the child. The fix is to launch the check from a session in which the restored library is already on the library path, and to make sure that path travels to the child.

In the workflow above, three pieces combine to do this. `setup-renv` restores into the project library. Every later R step starts in the repository root, where the project's `.Rprofile` activates that library. And `rcmdcheck()` hands the calling session's `.libPaths()` to the check subprocess, which is the default of its `libpath` argument. A workflow that restores into some other folder must pass that folder to `libpath` by hand.

The second piece is an assumption, and it is the same cross-boundary bug in waiting. It holds for the stock `.Rprofile` that `renv::init()` writes, which sources `renv/activate.R` in every session. It does not hold for every compendium. The `.Rprofile` that zzcollab scaffolds activates renv only when `ZZCOLLAB_CONTAINER` is set to `true`. On a CI host it skips renv, and later steps see the system library rather than the restored one. Such a compendium must activate the library explicitly on the runner: set that variable in the job's environment, or begin each R step with `source('renv/activate.R')`.

But why did the bug stay hidden? The workflow had passed for several repositories, because each of them imported only packages that the generic container base image already provided; the restored library was never actually needed. The defect surfaced only when a repository imported a specialized modeling package absent from the base image. The broader literature on why deposited analyses fail to re-run (Trisovic et al., 2022) documents how often re-execution simply fails. The particular fault here is cross-boundary state, the state that must pass from one step, shell, or process to the next and silently does not. It is where CI bugs hide. A dormant defect of this kind needs an adversarial input, a dependency outside the comfortable base image, to bring it to light. A check that passes tells

Draft, November 13, 2026. Content may change.

us only that nothing has yet exercised it hard enough to fail.

Check 5: capture versus validation

A colleague proposes to save CI minutes by deleting the continuous-integration workflow from a compendium that already has a committed `renv.lock` and `Dockerfile`. They argue that since the lockfile and the container 'contain the reproducibility', removing CI cannot lower the compendium's reproducibility level. Are they correct about the level? Are they correct that nothing is lost?
The answer is on page 538.

12.6. Secrets, tokens, and pinned actions

Before going further with verification, we pause on a matter of hygiene. A CI runner holds credentials the author's own laptop never needed, to comment on a pull request, push a built image, or deploy a rendered report. A leaked credential is a worse incident than a failed reproducibility check. It warrants the same discipline this chapter applies elsewhere: nothing should reach the runner, or leave it, undeclared.

The first rule is never to let a secret reach a log or a commit. GitHub's guidance on securing Actions workflows warns that sensitive values 'should never be stored as plaintext in workflow files' (GitHub, n.d.-d). A workflow instead references a stored secret through the `secrets` context, `${{ secrets.NAME }}`. The runner substitutes the value at execution time, and GitHub attempts to redact it from the log. The redaction is best effort: 'automatic redaction is not guaranteed', and the documentation recommends auditing how a workflow handles secrets to confirm one is never explicitly printed (GitHub, n.d.-d). The condensed workflow earlier in this chapter already follows the pattern. Its line `GITHUB_PAT: ${{ secrets.GITHUB_TOKEN }}` passes a credential into the job's environment without ever writing its value into the file.

The second rule is least privilege. The default permissions of the token GitHub Actions issues a job come from a repository, organization, or enterprise setting, which may be permissive. A workflow with no `permissions` block inherits that setting and can run with more access than most jobs

Draft, November 13, 2026. Content may change.

need. GitHub's hardening guidance calls a default of read-only repository contents 'good security practice', with individual jobs granted more only where they need it (GitHub, n.d.-d). The chapter's own example already does this: it declares `permissions: contents: read` at the top of the workflow.

The third rule has a direct parallel already established in this book. Chapter 7 showed that pinning a base image by tag is not enough. A tag is a mutable pointer that a maintainer can repoint to different bytes with no change to the Dockerfile that names it; pinning by digest is what actually fixes the bytes a build receives. A third-party GitHub Action carries the same hazard. A version tag such as `@v7` can be moved to a different commit by its maintainer, or by anyone who compromises the maintainer's account. The workflow that trusts the tag then runs whatever that commit contains on its next trigger, with no diff in the consuming repository to review. GitHub's own hardening guidance recommends pinning a third-party action to a full-length commit SHA rather than a tag. It states the rationale plainly: a tag 'can be moved or deleted' by anyone with write access to the action's repository, while forging a given commit SHA would need an attacker to produce a SHA-1 collision (GitHub, n.d.-d):

```
- uses: actions/checkout@3d3c42e5aac5ba805825da76410c18127
  ↪ 3ba90b1 # v7.0.1
```

The trailing comment records which release the SHA was taken from, since forty hex characters say nothing to a human reader. A SHA pin does not move on its own, so it also does not pick up a security fix. Dependabot or Renovate can watch the pinned actions and open a pull request that replaces the SHA and its comment when a new release appears, so that each update is a reviewed commit rather than a silent change.

The chapter's own workflow examples fall short of their own advice. They should be read as a teaching simplification, not a model to copy unchanged. The actions above (`actions/checkout@v7` and `r-lib/actions/*@v2`) and the GitLab example's `rocker/verse:4.6.1` all pin by tag. The CRAN gate tools are installed at whatever version is current. Only `zzrenvcheck`, which comes from a personal repository, is pinned to a commit. This book's Dockerfile explains a recipe by tag before showing the digest-pinned version,

Draft, November 13, 2026. Content may change.

and for the same reason: a tag is legible in a worked example in a way forty hex characters are not. A compendium's actual CI configuration should pin every third-party action to a commit SHA, exactly as its Dockerfile pins its base image by digest. A dependency that can change with no commit to the project's own history is undeclared state, whether it lives in a package repository, a container registry, or an action's git history.

Check 6: what a version tag on an action actually promises

A teammate argues that pinning an action with `uses: some-org/some-action@v3` is already safe, since the tag is version-controlled and anyone can see which commit `v3` points to on GitHub. What is wrong with this argument?
The answer is on page 538.

12.7. Two dialects: GitHub Actions and GitLab CI

We turn now from what the checks are to where they run. Continuous integration is a general idea with several implementations, and a health researcher should know at least two, because the choice is often not theirs to make. GitHub Actions and GitLab CI are two dialects of the same underlying activity, and the difference between them is one of syntax and hosting, not of principle.

The forge, the platform hosting the repository, matters for research in a way it does not for general software development. Research compendia are disproportionately authored on institutional, often self-hosted, GitLab. Many universities and laboratories run their own GitLab servers and cannot, or will not, push a repository to github.com. This is particularly so in Europe and wherever data-governance rules discourage commercial cloud hosting. Consider a clinical or registry study whose data-handling agreement forbids external hosting. For it, the ability to run the same reproducibility checks on institutional GitLab is a precondition. A framework that assumed GitHub would simply be unavailable to a large part of its intended audience.

The two dialects differ structurally. GitHub Actions uses one or more workflow files under `.github/workflows/`, each organized into jobs and,

Draft, November 13, 2026. Content may change.

within a job, a sequence of named steps. GitLab CI uses a single `.gitlab-ci.yml` organized into stages, each stage running a script. The same three gates, expressed in the GitLab dialect, look like this. We again show a static file that GitLab executes and this book does not:

```
stages:
  - gates
  - verify

gates:
  stage: gates
  # The tag must name the R version recorded in renv.lock.
  image: rocker/verse:4.6.1
  script:
    - Rscript -e 'renv::restore(prompt = FALSE)'
    - Rscript -e 'renv::install(c("rcmdcheck", "remotes",
        "pkgload"))'
    - Rscript -e
 ↪  'remotes::install_github("rgt47/zzrenvcheck",
        ref = "6f53235668cb4048c1ae793d64b10d6505bf69a2")'
    # Gate 1: R CMD check of the package skin
    - Rscript -e 'rcmdcheck::rcmdcheck(args =
 ↪  "--no-manual",
        error_on = "warning")'
    # Gate 2: the dependency manifest, analysis/ included
    - Rscript -e 'zzrenvcheck::check_packages(strict =
 ↪  TRUE,
        error_on_fail = TRUE)'
    # Gate 3: the report re-renders from source
    - Rscript -e 'rmarkdown::render(
        "analysis/report/report.Rmd",
        output_format = "html_document",
        output_dir = tempdir())'

verify:
  stage: verify
  # A shell-executor runner with Docker installed; the
  # tag is whatever the institution assigns to it.
```

```
  tags: [docker-shell]
  script:
    - docker build -t compendium:ci .
    - docker run --rm -e
↪   RENV_CONFIG_AUTOLOADER_ENABLED=FALSE
        -v "$PWD":/work -w /work
        compendium:ci Rscript ci/run-tests.R
```

The two files say the same thing in different words. Each runs on the R version the lockfile records, restores the pinned library into the project library, and installs the gate tools beside it. Each then runs the three gates from the repository root, so that every R step, and the check subprocess it launches, sees the restored library; the `.Rprofile` assumption of 'Where CI bugs hide' applies to both. The image tag above names R 4.6.1, and this book's own lockfile records R 4.6.1; a compendium on another release names its own. The `rocker/verse` image is used on GitLab because it ships pandoc, which the render gate needs; on GitHub the `setup-pandoc` action supplies it. Each script line is a separate shell command, and GitLab fails the job at the first one that exits with a nonzero status. Every gate here can therefore fail the pipeline.

The second job, `verify`, rebuilds the project image and runs the baseline tests inside it; it is the subject of the next section. On GitLab it needs a runner that can build images, which is an institutional decision. The common alternative, Docker-in-Docker, requires a privileged runner that many institutions do not allow.

On the plus side, maintaining a repository on whichever forge an institution mandates is then a matter of translating the dialect, and the checks carry over. On the minus side, a workflow written twice, once per dialect, is a workflow maintained twice. Every future change must be made in two incompatible formats, a standing divergence hazard. The resolution a mature framework adopts is to single-source the logic in a neutral place, the project's own `Makefile`. It holds one target per gate (say `check`, `deps`, and `render`), each carrying the commands shown above. Each forge's CI file then shrinks to a thin shim that checks out the code and calls `make`. The dialect differs only in its outermost ten lines, and the checking behavior is defined once.

Draft, November 13, 2026. Content may change.

A further design point is that a workflow need not be configured by hand: it can adapt itself to what the repository contains. A feature is treated as present exactly when its defining file exists. If `renv.lock` is present, the workflow restores the pinned library; otherwise it installs from `DESCRIPTION`. If a `Dockerfile` or a Nix `flake.nix` is present, the workflow builds and runs inside that environment; otherwise it runs on the host runner. Such a workflow survives a project's toggling of `renv` or of its container backend with no edit, and one file can serve every archetype and every capture level. The workflows in this chapter are not written that way. They assume a compendium with both a lockfile and a `Dockerfile`, so that every step is visible. A presence-driven workflow wraps each step in a condition, in GitHub's dialect `if: hashFiles('Dockerfile') != ''`. That convenience has a cost to weigh. A condition is also a way for a gate to disappear without notice, since a misnamed file switches the step off and the run stays green. A workflow that adapts should therefore print which gates it ran and which it skipped.

Check 7: the presence-driven trap

A presence-driven workflow skips the render gate when `if: hashFiles('report.Rmd') != ''` evaluates to false. Describe a concrete way this condition can silently disable the gate without anyone intending it to, and what single habit of the workflow catches the problem.
The answer is on page 538.

12.8. Verification and level L3

The first three gates confirm that the compendium builds, checks, and renders on a clean machine. They do not confirm that it produces the *same numbers*. That is the work of verification, the L2-to-L3 transition and the top of the ladder, which confirms what has been captured.

L3, as Chapter 3 defines it, requires that recorded outputs be shown to regenerate within a stated tolerance from a clean rebuild. Each part of that sentence names a mechanism. The recorded outputs and the tolerance live in a *baseline test* (Chapter 11): a committed file of key estimates is

Draft, November 13, 2026. Content may change.

recorded once, and a test file named `test-baseline-*.R` recomputes them and compares with `expect_equal(..., tolerance = ...)`. The clean rebuild is the project's own image, built from its `Dockerfile` on a machine that has never seen it. Verification is running the first inside the second, and it fails if either fails. It fails if the image does not rebuild, if any test fails, or if no baseline test runs at all. A test suite without a baseline test can pass in a rebuilt image and show that the environment is sound, which is L2 confirmed. However, it has compared nothing with a recorded result, so it is not L3.

In continuous integration this is a second job in the same workflow, run after the gates pass:

```yaml
verify:
  needs: gates
  runs-on: ubuntu-24.04
  steps:
    - uses: actions/checkout@v7

    # L2: the project's own Dockerfile must rebuild.
    - name: Rebuild the project image
      run: docker build -t compendium:ci .

    # L3: the test suite, with at least one baseline
    # test, must pass inside the rebuilt image.
    - name: Run the tests in the image
      run: >
        docker run --rm
        -e RENV_CONFIG_AUTOLOADER_ENABLED=FALSE
        -v "$PWD":/work -w /work
        compendium:ci Rscript ci/run-tests.R
```

The test step runs a short script from the repository. The obvious one-line call, `tinytest::run_test_dir('inst/tinytest')`, returns its results and exits with status 0 even when an expectation fails; we confirmed this with a deliberately failing test. The script turns failures into a nonzero exit. It also refuses to pass if no baseline test produced a result, which is the case when the baseline file is missing or when its guard found no data and exited early:

Draft, November 13, 2026. Content may change.

```
# ci/run-tests.R
pkgload::load_all('.', quiet = TRUE)
res <- tinytest::run_test_dir('inst/tinytest')
print(res)
ran <- unique(as.data.frame(res)$file)
if (!any(grepl('^test-baseline', ran))) {
  stop('No baseline test ran, so this is not an L3 check.')
}
if (!tinytest::all_pass(res)) quit(status = 1)
```

Two assumptions are built in and should be checked against the image in use. The image must contain `tinytest` and `pkgload`, which means they must be in the lockfile it restores. And because `RENV_CONFIG_AUTOLOADER_ENABLED=FALSE` stops the mounted project from activating its own library, the image must already have the pinned packages on R's default path, for example in the site library. The house rule applies here as well. Changing one value in the committed baseline file by more than the tolerance turns the job red at the baseline expectation. Deleting or renaming the `test-baseline-*.R` file turns it red at the 'no baseline test ran' check. Breaking a `RUN` line in the `Dockerfile` turns it red at the build. Against the toy compendium, outside a container, we ran the first two. We also ran a baseline file whose guard found no data and exited early. Each run of the script exited with status 1, and the unbroken run exited with 0. The image build and the job as a whole were not run for this chapter.

In the framework this book uses, the same composition is the command `zzc verify`, which runs in two tiers. A default *coherence* tier requires no rebuild. It checks that the captured artifacts agree with each other and that the R version named in the lockfile matches the one recorded when the Dockerfile was generated. It also checks that the backend is unambiguous, and that the raw data still matches its recorded checksums. A `--full` *reproduction* tier rebuilds the image and runs the test suite inside it. It stamps L3 into the project's state record only when three things hold: the image rebuilt, the tests passed, and at least one `inst/tinytest/test-baseline*.R` file exists and is named in the test log. The log names a file even when its guard exits early with no tests, so `ci/run-tests.R`, which counts results, is the stricter check. Without a baseline test it reports the project as rebuilt and tested, at L2, with no baseline comparison, and does not stamp L3.

Draft, November 13, 2026. Content may change.

The stamp is removed when zzcollab regenerates the Dockerfile or when a later `--full` run falls short of L3. A change made outside zzcollab, such as `renv::snapshot()` or an edited script, leaves it in place, so a stamp older than the last such change is stale. In regulated work, the stamp is not a validation record: it carries no signer, no stated meaning of a signature, and no quality review, which a controlled record of validation status requires.

But what, precisely, should be compared? Verification compares **computational outputs**, such as the row count of a derived dataset, a fitted coefficient, or a bootstrap bound, and *not* the rendered PDF or HTML. The reason is that a rendered document is not a deterministic function of the analysis. As Chapter 9 discussed, a PDF embeds the time of its creation and the identifiers of the fonts the rendering machine happened to have. Two renders of an unchanged analysis, minutes apart, produce PDFs whose bytes differ, even though every number in them is identical. Comparing PDFs would therefore report a difference on every render and catch nothing real. Comparing the estimates themselves, within a tolerance, reports a difference exactly when a number changed by more than floating-point noise, which is what we want to detect. A hash is the right instrument for a file whose bytes should not change, such as a raw data extract (the next section). It is the wrong one for an estimate, whose last binary digit can move under a different BLAS without the analysis changing. There exist techniques to normalize a document's embedded timestamps, by setting a fixed `SOURCE_DATE_EPOCH`, and to fix its build path. With enough care a byte-identical PDF is attainable. But the more general practice is to verify the computational outputs directly and leave the document to the render gate.

> **!** The honest level
>
> The three gates do not reach L3; only the verify job can, by running a baseline test in a rebuilt image. This chapter ran the gate logic and the test script, but not the image build or the job as a whole. A `verified` stamp older than the last change to the lockfile, the Dockerfile, or the code is stale, not an L3 claim. The gates alone say nothing about whether the raw data changed; the checksum manifest of the next section does. In regulated work, an L3 stamp is also not a validation record (Chapter 11). Nor is a hosted run a lasting record:

Draft, November 13, 2026. Content may change.

GitHub deletes workflow logs and artifacts after 90 days by default, a period an organization can change within limits (GitHub, n.d.-b). A log or artifact that supports a claim must therefore be downloaded and kept with the records it supports, for as long as they are kept.

Check 8: trusting a stamp

A compendium was stamped `verified` three months ago. Last week its `renv.lock` was updated to pin a new package version, and CI has not run since. Is the `verified` stamp still an honest L3 claim? Why or why not?
The answer is on page 539.

12.9. Data integrity: the checksum manifest

We now turn to an input that sits outside everything the lockfile and the container pin: the raw data itself. A container fixes the software that reads the data, and a lockfile fixes the packages. However, neither notices if the data file on disk is silently altered between the analysis and its re-execution (a row edited, a value corrected, a column recoded). An altered input produces a changed result with no error and no warning.

The guard is a **checksum manifest**, a small file, conventionally `data-manifest.sha256`, that records the SHA-256 checksum of each raw data file. Verification reads the manifest, recomputes the checksum of each file as it now stands, and reports a mismatch. Because any change to a file's bytes changes its checksum, a silently altered input is caught the next time verification runs.

To illustrate, consider a small synthetic example. We write a tiny data file to disk, record its checksum, write the same values again from a different kind of R object, write a third copy with a single value altered, and hash each file. The checksum is computed from the file's bytes, not from the R object in memory. The point to observe is that a one-cell change, a shift of a single outcome from 11.1 to 11.2, produces a completely different digest, so the alteration cannot pass unnoticed.

Draft, November 13, 2026. Content may change.

```
library(tibble)
library(digest)
library(knitr)

cohort <- tibble(
  id      = 1:5,
  arm     = c('treatment', 'placebo', 'treatment',
              'placebo', 'treatment'),
  outcome = c(12.4, 9.8, 11.1, 10.3, 13.0)
)

hash_csv <- function(x) {
  path <- tempfile(fileext = '.csv')
  write.csv(x, path, row.names = FALSE)
  digest(file = path, algo = 'sha256')
}

altered <- cohort
altered$outcome[3] <- 11.2

digests <- c(hash_csv(cohort),
             hash_csv(as.data.frame(cohort)),
             hash_csv(altered))

manifest <- tibble(
  file = c('cohort.csv (baseline)',
           'same values, from a data.frame',
           'one cell altered'),
  digest_first16 = substr(digests, 1, 16),
  matches_baseline = ifelse(digests == digests[1],
                            'yes', 'no')
)

kable(
  manifest,
  col.names = c('File', 'SHA-256 (first 16 chars)',
                'Matches baseline'),
  align = 'llc'
```


Draft, November 13, 2026. Content may change.

```
)
```

Table 12.1.: A single altered value changes the digest of a data file. The three CSV files are hashed by their bytes on disk. The second holds the same values as the first, written from a plain data frame rather than a tibble, and its digest matches. The third differs in exactly one cell, and its SHA-256 digest differs from the first character, so a checksum manifest detects the change.

File	SHA-256 (first 16 chars)	Matches baseline
cohort.csv (baseline)	cc03e329ae1b1114	yes
same values, from a data.frame	cc03e329ae1b1114	yes
one cell altered	632857fa64a59ad4	no

The altered file's digest differs from the first character, although the files differ in a single cell. That sensitivity to any change is what makes a checksum a reliable integrity guard. Suppose a verification run recomputes these digests and finds the third where it expected the first. It has then caught an altered input, and has done so without needing to know what the correct value was, only that it has changed since the manifest was recorded. The second row shows why the file, and not the object, is what gets hashed. Hashing the tibble and the data frame themselves with `digest()` would serialize each object, class attributes included, and give two different digests for identical values: here `digest(cohort) == digest(as.data.frame(cohort))` is FALSE. The bytes on disk carry only the values.

A manifest line pairs a digest with a path, in the format `shasum -a 256` writes and `shasum -a 256 -c` checks. For this book's raw extract it reads:

```
raw_path <- '../data/raw_data/cohort_raw.csv'
cat(digest(file = raw_path, algo = 'sha256'), '  ',
    'analysis/data/raw_data/cohort_raw.csv', '\n', sep =
     ↪  '')
#> aec0973593e41aa0186ca57d44949a369323a71d9e21fcc25d6cc02
 ↪  5b134aea7  analysis/data/raw_data/cohort_raw.csv
```


Draft, November 13, 2026. Content may change.

Check 9: what to hash

A student proposes to verify their manuscript compendium by recording the SHA-256 checksum of the rendered `report.pdf` and rechecking it on every rebuild. They reason that if the PDF is byte-identical, the whole analysis must have reproduced. Identify the flaw, and say what they should hash instead.
The answer is on page 539.

12.10. Proving the framework itself: the capture test matrix

One rung below the compendium's own verification sits an easily overlooked question: how does the framework know that a project configured for, say, L1 actually reaches L1? A claim that a given combination of toggles produces a given level is itself a claim that can be tested, and a responsible framework tests it rather than asserting it.

The instrument is a **capture test matrix**. It scaffolds a throwaway compendium for each combination of the capture axes: the package backend present or absent, and the container environment present or absent. For each, it asserts that the result passes `R CMD check`, is a loadable workspace, and sits at its stated level. The matrix has the following shape. Its axes are the capture axes; CI is the harness that runs the matrix, not a dimension of it.

Backend (`renv`)	Environment (Docker)	Level	Assertion
off	off	L0	check passes; valid workspace
on	off	L1	check passes; `renv` activates on host

Draft, November 13, 2026. Content may change.

Backend (`renv`)	Environment (Docker)	Level	Assertion
off	on	L0	check passes; image installs from `DESCRIPTION`, so the environment is fixed but the packages are unpinned
on	on	L2	check passes; image restores from lockfile

The third row sits at L0 although it builds an image, because the rungs are cumulative. L2 presupposes the lockfile of L1, and an image that installs from `DESCRIPTION` pins no package version.

Two points about the matrix deserve emphasis. First, Nix, the alternative backend of Chapter 8, is verified as a *substitution*. It replaces the `renv`-plus-Docker combination and must reach L2 on its own, so it is checked as a peer of that combination rather than added as a fourth column. Second, the configuration that renders a report with `renv` on but Docker off is, in practice, the most error-prone, and it is exercised explicitly. Because container builds under emulation are expensive, the matrix is tiered. The host-only rows run on every pull request, and the rows that build an image run on merge and on a nightly schedule.

Check 10: testing the claim, not just the tool

Why does a responsible framework test the claim ‘this combination of toggles reaches level L1’ rather than simply asserting it in documentation?
The answer is on page 539.

Draft, November 13, 2026. Content may change.

12.11. How continuous integration adapts to the archetype

Continuous integration adapts its gates to the archetype; Table 5.2 in Chapter 5 is the reference. Of the three gates of this chapter, `R CMD check` and the dependency-manifest gate are universal, run by every archetype. The latter, however, runs in an archetype-conditional form. It is the strict `DESCRIPTION`-inclusion check for an R package, and the weaker Code ⊆ renv.lock inclusion for a compendium, as the section on the three gates established. The render gate is the conditional one. It is present wherever a report exists, and absent from an **R-package archetype**, which has none.

The case most worth stating here is the **simulation archetype**, which raises a distinct concern of cost. A full run can take hours or days. The idiomatic arrangement therefore runs a *reduced* pipeline under CI: a handful of iterations with a fixed seed, enough to confirm the pipeline executes and the outputs have the right shape. The full pipeline runs elsewhere, on a cluster or a scheduled job, and is verified against a baseline separately. The seeding discipline of Chapter 11 is what makes the reduced run meaningful. A fixed seed makes even a short run reproducible, so CI checks reproducibility rather than merely execution. The presence-driven workflow discovers the archetype from what the repository contains. The analyst's contribution is therefore to ensure the archetype and its gates are matched.

Check 11: when a package does need the render gate

An R-package archetype is said to have no report, so no render gate. Under what condition does the render gate apply to a package archetype after all, and what does it render in that case?
The answer is on page 540.

12.12. Independent external verification

The checking we have described so far is the author's own, run on the author's forge against the author's baseline. It is necessary and it is not

Draft, November 13, 2026. Content may change.

sufficient. An author who has misunderstood their own reproducibility can build a CI pipeline that faithfully confirms the misunderstanding. The complement is verification from outside the project entirely, by someone with no stake in its passing.

The most developed such scheme is CODECHECK, described by Nust and Eglen (Nüst & Eglen, 2021). In it an independent codechecker re-executes an analysis during peer review, on their own machine, and issues a certificate recording what was run and what was reproduced. The value of the exercise lies exactly in its independence. The codechecker carries none of the author's undeclared state; this is the same property that makes a CI runner effective, here held by a person instead of a virtual machine. In a parallel development, major computing venues have formalized artifact review and badging. Under it, a submission's computational artifacts are evaluated and, if they reproduce, marked with a badge on the published paper (Association for Computing Machinery, 2020). Stodden and colleagues have documented the persistence of the reproducibility problem. They have also documented the effectiveness, and limits, of journal policies that attempt to compel deposition and checking (Stodden et al., 2018). The ten rules, for their part, ask for automated steps and public access to scripts, runs, and results, which is what makes independent re-execution possible (Sandve et al., 2013).

These schemes validate and capture nothing: they add no lockfile and no container, only a second, independent execution of the analysis. Continuous integration is a machine that is not yours, and independent verification is a person who is not you. Both catch the reproducibility failures that the author's own generous machine had hidden, at the last responsible moment before a result enters the scientific record.

Check 12: what independence adds beyond CI

An author's CI pipeline passes every gate, including verification against a baseline. Why might a CODECHECK reviewer still find something CI did not, given that both are described as 'a machine, or a person, that is not yours'?
The answer is on page 540.

Draft, November 13, 2026. Content may change.

12.13. Worked example

To make the machinery concrete, consider a plausible sequence of events in a cohort-study compendium, of the kind that recurs in practice.

The analyst is refining a model and, mid-session, reaches for a package that computes robust standard errors, `sandwich`. They install it interactively with `install.packages()`, and add `library(sandwich)` to `analysis/scripts/02_model.R`, which the report sources. They obtain the result they wanted, and commit. On their own machine everything runs: the package is installed, the script loads it, the report renders, the numbers appear. The compendium looks, to its author, finished.

The push triggers CI, and the fresh runner tells a different story, though not at the gate a reader might expect. The runner checks out the committed files and restores the pinned library from `renv.lock`. This lockfile was never updated, because the interactive install was never snapshotted. Gate 1, `R CMD check`, passes. It is not at fault: the script lives in `analysis/`, which the build excludes, so the check never reads it. Gate 2, the dependency-manifest gate, reads `analysis/` and goes red. It names `sandwich` as a package the code uses and the lockfile does not pin. Had that gate been absent, gate 3 would have failed the same run from the other direction. The report sources the script, the script calls `library(sandwich)`, and the restored library has no such package, so the render stops at that chunk. Either way, CI has found on a clean machine precisely the undeclared state the author's machine supplied silently. Had the workflow run only `R CMD check`, as many compendium workflows do, the build would have been green.

The fix is not to weaken the check but to repair the capture. The analyst records the new package in the lockfile by snapshotting the environment, commits the updated `renv.lock`, and pushes again. This time the runner restores a library that contains the package. The dependency gate passes, the report renders, and the gates go green. The `verify` job then rebuilds the image from the new lockfile and runs the baseline test. This is where the analyst learns whether the change moved a recorded result, such as a confidence limit. If it did, the baseline is re-recorded deliberately, in a commit that says why, not loosened until it passes.

Draft, November 13, 2026. Content may change.

Check 13: which gate, and why

In the worked example, `R CMD check` passes even though `sandwich` is in fact unpinned. Why does this gate miss the defect that the dependency-manifest gate catches, and is that a flaw in `R CMD check`? *The answer is on page 540.*

Check 14: complete the workflow file

The worked example narrates a run rather than printing a workflow file, so this check uses the chapter's main GitHub Actions example, the condensed `compendium` workflow of the three gates. It is reprinted verbatim except for four placeholders, two of which stand for a short block rather than a single line. Write each one. For (2), write the line as Section 12.6 says a compendium should pin it, not as the condensed file prints it.

```
name: compendium

# (1) ____: the trigger, every push to main and every
#     pull request into main

permissions:
  contents: read

jobs:
  gates:
    runs-on: ubuntu-24.04
    env:
      GITHUB_PAT: ${{ secrets.GITHUB_TOKEN }}
    steps:
      # (2) ____: check out the commit, pinned by SHA

      # Install the R version recorded in renv.lock.
      - uses: r-lib/actions/setup-r@v2
        with:
          r-version: renv
          use-public-rspm: true

      - uses: r-lib/actions/setup-pandoc@v2

      # Restore renv.lock into the project library. A
      # stock renv .Rprofile activates that library in
      # every later R step run from the repository
↪   root.
      # (3) ____: restore the lockfile

      # The gate tools are not analysis dependencies,
↪   so
      # they are not in renv.lock. zzrenvcheck comes
↪   from
      # a personal repository and is pinned to a
↪   commit.
      - name: Install the gate tools
        shell: Rscript {0}
        run: |
          renv::install(c('rcmdcheck', 'remotes',
↪   'pkgload'))
          remotes::install_github(
            'rgt47/zzrenvcheck',
```


Draft, November 13, 2026. Content may change.

The answer is on page 540.

12.14. Debugging a failed reproduction

The worked example above is the easy case: CI catches the defect the same day, on the same commit that introduced it. The harder case arrives later. A compendium that rendered cleanly in March fails in October, when a reviewer clones it, a collaborator resumes it, or the author themselves reruns it for a revision. Nothing in this section is generic R debugging, which is out of scope for this book. It is a diagnostic sequence specific to one question: it worked before, and it does not now, so what changed?

The first move is to localize the failure in time, not in code. Suppose the history between the last known-good state and the present failure holds more than a handful of commits. Reading each one is then slower than letting git find the boundary. `git bisect` does this by binary search. Told one commit that is good and one that is bad, it checks out the midpoint and asks for a verdict, halving the remaining range at each step.

```
git bisect start <bad-commit> <good-commit>
git bisect run Rscript -e 'rmarkdown::render("report.Rmd")'
git bisect reset
```

`git bisect run` automates the verdict. It checks out each candidate commit and runs the given command, taking a nonzero exit status as 'bad' and zero as 'good'. It continues until a single first-bad commit remains. We verified the mechanism on a disposable four-commit toy repository, under git 2.55.0. The first commit established a check that passed. The second, an unrelated-looking refactor, silently changed a value the check depended on, and two further commits added only comments. We marked the fourth commit bad and the first good with `git bisect start`, then ran `git bisect run Rscript check.R`. It tested the midpoint, found it still bad, and tested the commit below that. It then named the second commit as the first bad one, in two bisection steps rather than three linear checks. The command handed to `bisect run` need not be a toy. It can be

Draft, November 13, 2026. Content may change.

the render gate or a baseline test from Chapter 11, which is exactly what lets the search run unattended across dozens of commits.

Once the boundary commit is known, the next question is whether the commit changed code, or whether something outside the commit changed under it, because `git bisect` can only blame a commit if the environment at each checkout is otherwise identical. Consider a pipeline that reruns the same pinned commit weeks apart and gets a different answer. It did not fail because of anything in the diff, and the fix is to compare environments, not history. `sessionInfo()` (Chapter 6 and the glossary) is the cheapest comparison. It is run in the working environment and the failing one, and the two are diffed by eye for an R version, platform, or package version that moved. `renv::status()` goes further, comparing the project library with the lockfile and reporting packages that are missing or installed at a different version. It is the first thing to run when the suspicion is drift in the restored library rather than in the analysis code. When the suspicion lands on the lockfile itself, `git diff <good-sha> <bad-sha> -- renv.lock` shows exactly which package or version entries changed between the two commits. This narrows 'something in the environment changed' to the one or two packages responsible.

If the analysis is a `targets` pipeline (Chapter 10), the diagnostic moves from history into the dependency graph. `tar_outdated()` lists which targets the pipeline considers stale and therefore due to rerun. `tar_visnetwork()` draws the same information as a graph. The analyst can then see where in the chain of targets the invalidation begins, rather than at which commit it was introduced. This is the pipeline-native counterpart to bisecting, since `targets` already tracks what depends on what. The question 'what broke, and what does it affect' can therefore be answered from the dependency graph instead of from commit history. This holds provided the pipeline's metadata store, `_targets/`, survived the problem. On a fresh CI runner it is absent unless it was committed or the workflow restores it, and a pipeline run there without it rebuilds every target.

The last diagnostic is reading a CI log on its own terms. This chapter has already named the three gates and the discipline of reproducing a failure exactly as CI ran it, not as is convenient to invoke by hand ('Where CI bugs hide', above). Applied to a failed reproduction, that discipline has three parts. First, identify which of the three gates reported red: `R CMD check`,

Draft, November 13, 2026. Content may change.

the dependency manifest, or the render. Second, read what it printed, not just that it failed, since the WARNING text or the `stop()` message names the missing package or the broken chunk directly. Third, reproduce that exact step locally, from a session with the same restored library CI used. This is preferable to re-running the whole render and hoping the same failure reappears in a more informative form.

None of these five moves is a substitute for the others, and the order matters more than any single step. Bisection is cheap to run and narrows the search space before anything else is attempted. Session-info and lockfile comparisons cost nothing once the boundary commit is known. They settle, quickly, whether the defect is in the diff or in the environment around it. `targets` diagnostics apply only when a pipeline is involved. Reading the CI log carefully is usually the last step, because by the time it is needed the first four moves have already said which gate to expect it from.

Check 15: bisecting an environment problem

A pipeline's render step fails on a Tuesday rerun of a commit that rendered cleanly the previous Friday. The commit has not changed. Why will `git bisect` be useless here, and what should be compared instead?
The answer is on page 541.

12.15. Demonstrating a gate on the cohort compendium

The logic that runs on a CI runner also runs, in miniature, on this book's own cohort compendium, without any CI infrastructure at all. Below, a short script mirrors the dependency-manifest gate of the earlier section. It finds every package `R/cohort.R` actually uses, compares that list against what this compendium currently declares as pinned, and reports whether the gate passes.

```
used <-
 ↪  unique(renv::dependencies('../../R/cohort.R')$Package)
#> Finding R package dependencies ... Done!
```


Draft, November 13, 2026. Content may change.

```
locked <-
 ↪  names(renv::lockfile_read('../../renv.lock')$Packages)
base_pkgs <- rownames(installed.packages(priority =
 ↪  'base'))

gate_check <- function(missing) {
  if (length(missing) > 0) {
    stop('Used but not pinned in renv.lock: ',
     ↪  toString(missing))
  }
  'gate passed'
}

missing_now <- setdiff(used, c(locked, base_pkgs))
cat('Packages R/cohort.R uses:', toString(used), '\n')
#> Packages R/cohort.R uses: parallel, stats, survival,
 ↪  tibble, utils, withr
cat('Missing from the pinned set:',
    if (length(missing_now)) toString(missing_now) else
     ↪  '(none)', '\n')
#> Missing from the pinned set: (none)
cat('Result:', gate_check(missing_now), '\n')
#> Result: gate passed
```

The gate passes. `R/cohort.R` uses `parallel`, `stats`, `survival`, `tibble`, `utils`, and `withr`, and each is either pinned in `renv.lock` or part of base R. The house rule of this book applies here as well: a gate shown only to pass has not been shown to work. We therefore run the same script against a `used` list with the chapter's own running example of an unpinned dependency added. It shows what a real CI log would print the moment someone adds `library(sandwich)` to a cohort script without snapshotting it:

```
used_broken <- c(used, 'sandwich')
missing_broken <- setdiff(used_broken, c(locked,
 ↪  base_pkgs))
result <- tryCatch(gate_check(missing_broken),
```


Draft, November 13, 2026. Content may change.

```
                    error = function(e) conditionMessage(e))
cat('Missing from the pinned set:',
 ↪  toString(missing_broken), '\n')
#> Missing from the pinned set: sandwich
cat('Result:', result, '\n')
#> Result: Used but not pinned in renv.lock: sandwich
```

The injected package is named, and the step would stop with a nonzero exit status in real CI. The failure points at the undeclared dependency, which is what the dependency-manifest gate is for. Restoring `used` to its real value (dropping the injected `sandwich`) returns the gate to green, as reverting the offending commit would on a real runner.

Check 16: what this demonstration does and does not show

This section ran the dependency-gate logic locally, inside the rendered report, rather than on an actual CI runner. What does that leave unverified about the compendium's real CI setup, if one exists?
The answer is on page 542.

12.16. Collaborating with an LLM

A large language model is a capable assistant for CI work, which is heavy on boilerplate YAML. It is also a hazardous one, because CI failures are subtle and a plausible-looking workflow can still be wrong. We give three prompt patterns, each with what to watch for and what the analyst must independently verify.

! Restricted data

One rule comes first (habit 3). Never paste or upload protected health information, identifiable records, or data held under a data use agreement, nor outputs derived from them, into a model or a coding agent unless three conditions all hold. The institution has approved that endpoint for that class of data. The IRB protocol and the consent

permit the processing. The data use agreement, or the provider's terms, permits it. If any of the three is unclear, the answer is no.
The third condition is the one most often missed. Institutional approval of a tool does not override the terms under which a data provider released the data. NIH, for example, treats sharing controlled-access data through prompts with public generative AI tools, which it glosses as third-party tools, as a breach of the data use certification. It treats a model trained on such data as a data derivative, to be shared only with other approved users and not kept after the project closes (National Institutes of Health, 2025a).
Describe the structure instead: the column names and types, or a small synthetic example generated from the data dictionary, never edited from real rows. A coding agent sees whatever files are in its working tree, so run it in a checkout or container where the real data is not mounted.
An agent also acts, so confine it to a branch and keep merges, pushes, and deposits for yourself (Section 2.10.1).

CI touches data in ways that are easy to miss. A CI log that prints rows of a failing render or test is a derived output in the sense of the rule above, so a log excerpt is pasted only after those rows are removed. For CI work the structure to describe is the workflow file and the error message.

Prompt. 'Write a GitHub Actions workflow that runs `R CMD check` on my compendium inside the `rocker/tidyverse` container and restores packages from `renv.lock`.'

Watch for. A workflow that restores the library in one step and runs the check in another without ensuring the restored library reaches the check subprocess. As we saw, adjusting the library path in one step does not carry to the next, and `R CMD check` runs in a child process that sees only the library path it is handed. Check for the naive two-step form, which passes only when every dependency happens to be in the base image. Watch also for `error_on = 'error'`, a dependency step that reports without failing, and `continue-on-error: true`. Each of these leaves a gate that cannot go red. Remember, too, that `R CMD check` alone never looks at `analysis/`.

Verification. Do not trust the workflow because it is syntactically valid.

Draft, November 13, 2026. Content may change.

Confirm it fails correctly by introducing a deliberate gap: import a specialized package absent from the base image, do not restore it, and check that the build goes red. A CI check that has never been seen to fail has not been shown to work. Seeing means reading the run log yourself: a coding agent's report that 'CI is green' is a claim about the run, not an observation of it.

An agent asked to make a failing build pass can reach green by the patterns this triple lists, adding `continue-on-error: true` or loosening `error_on`, and the change arrives as an ordinary commit. Treat `.github/workflows/` as a protected path and read every line of any diff that touches it before it merges. CI is also where embedded instructions matter most. An agent that reads a log, an issue, or a downloaded file can be steered by text planted in it, and in CI it may run with the repository's secrets. Grant its jobs the least privilege of Section 12.6, and keep pushes and merges with the analyst (Section 2.10.1).

Prompt. 'Add a verification step that checks my analysis reproduces, by hashing the output and comparing it to a saved value.'

Watch for. A step that hashes the rendered PDF or HTML. The report is the most visible output and the wrong thing to hash, because its bytes carry timestamps and font identifiers and change on every render.

Verification. Insist that the target is a computational output. Estimates should be compared with a committed baseline within a stated tolerance, in a `test-baseline-*.R` file, while hashes are kept for files whose bytes should not change. Confirm by rendering the report twice without changing the analysis: the report's own bytes should differ while the baseline test still passes. If the model's chosen target changes between the two renders, it is the wrong target.

Prompt. 'My compendium is a simulation study that takes six hours to run. Write CI for it.'

Watch for. A workflow that attempts the full run on every push, which will time out or consume the CI budget, or one that runs so few iterations that it no longer tests anything meaningful.

Verification. Confirm the workflow runs a reduced, fixed-seed pipeline under CI and defers the full run elsewhere. Check that the reduced run is seeded

Draft, November 13, 2026. Content may change.

so that its short output is itself reproducible. Verify that the number of iterations in the reduced run is large enough to exercise the code paths that matter, not merely one.

A better prompt. 'My compendium's simulation runs `<B>` replicates in about six hours; `R/` holds the functions and `inst/tinytest/` the tests. Write a GitHub Actions workflow that, on each push, runs the tests and a fixed-seed run of `<b>` replicates within 20 minutes, pins each action by commit SHA, and never sets `continue-on-error`. Say where the full run should happen instead, and what deliberate change I should make to confirm that each step can fail.'

In each case a syntactically valid workflow is the easy part to check. The semantics, what the check actually proves and whether it can fail for the right reason, are what the analyst must verify and cannot delegate.

12.17. Exercises

1. **Capture versus validation.** In your own words, explain why removing a continuous-integration workflow from a compendium does not lower its position on the reproducibility ladder, and state precisely what is lost when CI is removed. Give one concrete failure that CI would have caught and that its absence now permits to go unnoticed.

2. **The three gates.** For each of the three CI gates (`R CMD check`, the render gate, the dependency-manifest gate), describe a specific defect in a compendium that the gate catches and the other two do not. Then name one reproducibility failure that none of the three catches, and say which gate, verification or data integrity, addresses it.

3. **What a digest can and cannot tell you.** Take the code in Table 12.1 as a starting point. Extend it to hash the coefficients of a small linear model fitted to the synthetic `cohort` data with `lm(outcome ~ arm, data = cohort)`, record the digest, refit the identical model in the same session, and confirm the digest is unchanged. Then alter one outcome value, refit, and confirm the digest changes. Explain why this hash reliably detects that *something* changed. Then explain why it is still the wrong tool for the question a baseline test answers, whether a reported estimate has drifted. Say what would make the digest an

Draft, November 13, 2026. Content may change.

unreliable guard even when the coefficients themselves are unchanged, and why a tolerance-based comparison is the correct instrument for a floating-point result.

4. **Two dialects.** Take the condensed GitHub Actions workflow shown in this chapter and translate it into the equivalent `.gitlab-ci.yml`. Identify the two structural differences between the dialects (jobs-and-steps versus stages-and-scripts, and per-file versus single-file configuration), and explain why single-sourcing the check logic in a `Makefile` would let both files shrink to a thin shim.

5. **Archetype and gates.** For each of the five archetypes (data analysis, manuscript, R package, simulation, blog), state which of the three CI gates apply and why. For the simulation archetype, describe specifically how you would arrange a reduced pipeline under CI and a full pipeline elsewhere, and what role a fixed seed plays in making the reduced run a genuine reproducibility test.

6. **Independent verification.** Read the CODECHECK description of Nust and Eglen (Nüst & Eglen, 2021). Explain in what respect an independent codechecker plays the same role, for a submitted analysis, that a CI runner plays for a committed change. Then identify one class of reproducibility failure that an independent human codechecker would catch and that the author's own CI, however well configured, cannot.

7. **Cumulative: gate the cohort compendium.** Using the pattern of 'Demonstrating a gate on the cohort compendium', run the following from the repository root (not from `analysis/report/`), so that the pinned set is read from the real lockfile rather than typed: `script <- 'R/cohort.R'`, `used <- unique(renv::dependencies(script)$Package)`, `locked <- names(renv::lockfile_read('renv.lock')$Packages)`, and `missing <- setdiff(used, c(locked, rownames(installed.packages(priority = 'base'))))`. Report the length of `missing` (an autogradable answer: a single integer). Then copy `R/cohort.R` to a new file, add the line `library(sandwich)` to the copy, set `script` to the copy's path, re-run the `used` and `missing` lines, and report the exact character vector `missing` now contains (a second autogradable

Draft, November 13, 2026. Content may change.

answer: the package names, spelled exactly as they would appear in a real CI log).

8. **Spacing: commit before you compute, revisited.** This exercise returns to the commit discipline of Chapter 4 (Chapter 4). A CI workflow triggers on `push`, so it can only test what has actually been committed. Any undeclared state that exists solely in an uncommitted file or an author's working directory is invisible to it, in exactly the way it is invisible to a colleague who clones the repository. Explain, in your own words, why this makes 'commit before you compute' (habit 1) a precondition for CI to mean anything at all, and give one concrete example of an uncommitted change that would let a build pass locally and fail the moment it is pushed.

12.18. Solutions

Attempt an exercise before expanding its solution. Exercises 3 and 4 are hands-on and their outputs will vary; the boxes record the expected result and the reasoning.

Exercise 1

The ladder grades **capture**: how far down the determinant stack a result is locked. CI locks nothing. It installs no package into the lockfile and no library into the image, so removing it leaves every pinned layer exactly as pinned as before. A compendium at L2 with CI and the same compendium at L2 without it sit on the same rung.
What is lost is assurance rather than level. Without CI nobody rebuilds the captured environment on a clean machine and re-runs the analysis in it. The compendium becomes trusted rather than tested, and the distance between 'this should rebuild' and 'this does rebuild' stops being measured.
A concrete failure. A base image is repointed by its maintainer and the Dockerfile pins it by tag rather than digest, so the environment changes underneath the project. With CI, the next push rebuilds and the check fails, naming the day it broke. Without CI, the compendium looks healthy on the author's machine, where an old image is cached

Draft, November 13, 2026. Content may change.

locally. It then fails for the first reader who builds it from scratch, possibly years later, with no indication of when or why it broke. Equally good answers include a package pulled from CRAN and no longer restorable, or a system library the analysis silently depended on and never declared. A chunk that only ever ran because of state left in the author's session is another.

Exercise 2

`R CMD check` catches a defect in the package skin as code. One example is a function in `R/` calling a package that is used but not declared in `DESCRIPTION` (a WARNING, so the gate must fail on warnings). Others are a syntax error in a file under `R/` that nothing currently calls, or a documented argument that no longer exists. The render gate would not see it if the report does not touch that function. The manifest gate checks declarations rather than whether the code loads. The same defects under `analysis/` are invisible to `R CMD check`, which never sees that directory.

The render gate catches a defect that appears only when the document is built. Examples are a chunk that fails on a clean machine because it read an uncommitted file, a figure whose code errors, or a cross-reference to a label that no longer exists. `R CMD check` never renders, so it sees none of this.

The dependency-manifest gate catches drift between the code and the manifests: a package used in a script under `analysis/` and missing from `renv.lock`, which `R CMD check` never sees, or, for the package archetype, a package declared in `DESCRIPTION` and missing from `renv.lock`. The code may load and the document may render perfectly on a machine where the package happens to be installed, so neither other gate fires.

A failure none of the three catches: a change in the *data*. A raw extract silently replaced with a corrected version leaves the code valid, the document renderable, and the manifests consistent. Meanwhile every number in the report changes. All three gates pass. The **data-integrity gate** addresses it, by hashing the inputs and derived outputs and comparing against a recorded manifest.

Draft, November 13, 2026. Content may change.

The **verification gate** addresses the adjacent case in which the code changes a result without any of the three noticing. It does so by comparing computed outputs against recorded baselines.

Exercise 3

```
library(digest)
m1 <- lm(outcome ~ arm, data = cohort)
d1 <- digest(coef(m1), algo = 'sha256')

m2 <- lm(outcome ~ arm, data = cohort)
same_refit <- digest(coef(m2), algo = 'sha256') == d1
same_refit
#> [1] TRUE

cohort_alt <- cohort
cohort_alt$outcome[1] <- cohort_alt$outcome[1] + 1
m3 <- lm(outcome ~ arm, data = cohort_alt)
same_altered <- digest(coef(m3), algo = 'sha256') == d1
same_altered
#> [1] FALSE
```

Refitting the identical model to identical data, in the same session, gives an identical digest; altering one outcome value changes it. The chunk above, run on R 4.6.0 when this book was rendered, returned TRUE and then FALSE.

Why the digest is sensitive to any change. The coefficients are a deterministic function of the data and the model specification. Nothing about the time of fitting, the machine's fonts, or the session's history enters them, so within one session and platform the digest changes exactly when a number changes and not otherwise.

Why it is still the wrong tool for verification. A digest compares serialized *bytes*, not numeric *values*. R's own documentation for `serialize()` (the mechanism `digest()` and `saveRDS()` both rely on) states that the default serialization format changed from version 2 to version 3 in R 3.6.0. It also warns explicitly that 'the serialization format may change in future versions of R, so this interface should not

Draft, November 13, 2026. Content may change.

be used for long-term storage of R objects' (`?serialize`, inspected). Nothing guarantees that two R versions, or two platforms, serialize an identical numeric vector to identical bytes. An added attribute, a changed internal representation, or a future format bump can flip the digest while every coefficient is still correct to the last bit. That makes a coefficient digest a plausible false-positive generator across upgrades. This is exactly the failure mode this book's baseline test is built to avoid, by comparing *values* with `expect_equal(..., tolerance = ...)` rather than comparing serialized bytes with `==`. The digests in Table 12.1 are defensible because they hash the bytes of a data file on disk, not a serialized R object, and byte-for-byte identity of a raw data file is the actual requirement. Reused on a fitted model's coefficients, it answers a different, stricter question ('are the bytes identical') than the one a reproducibility check needs answered ('is the value the same within tolerance'). It will occasionally answer it wrong in a way that looks like a false alarm.

One practical refinement the exercise invites. Digesting the whole `lm` object rather than `coef()` is fragile, because the object stores the call and a `terms` environment, which can carry incidental references. Hashing the coefficient vector, or a named list of the quantities actually reported, is more robust against incidental differences. However, it does not remove the cross-version risk above. A baseline test with a stated tolerance remains the right tool for verifying that an estimate has not drifted.

Why hashing a plot would not be sound. A rendered plot is an image file. Image files embed creation timestamps, are affected by device version and font availability, and can differ in their compression between runs. Two plots of the identical model can differ byte for byte while depicting exactly the same thing. The check would therefore fail on every rebuild and carry no information. The rule this generalizes to is that verification hashes computational outputs, not rendered artifacts; it is the same rule that governs the PDF in Chapter 9.

Draft, November 13, 2026. Content may change.

Exercise 4

```
# .gitlab-ci.yml
stages: [gates]

gates:
  stage: gates
  image: rocker/verse:4.6.1
  script:
    - Rscript -e 'renv::restore(prompt = FALSE)'
    - Rscript -e 'renv::install(c("rcmdcheck",
 ↪  "remotes"))'
    - Rscript -e
 ↪  'remotes::install_github("rgt47/zzrenvcheck",
        ref =
 ↪  "6f53235668cb4048c1ae793d64b10d6505bf69a2")'
    # Gate 1: fails on a WARNING, as error-on does on
 ↪  GitHub
    - Rscript -e 'rcmdcheck::rcmdcheck(args =
 ↪  "--no-manual",
        error_on = "warning")'
    # Gate 2: the target that modifies nothing
    - make check-renv-no-fix
    # Gate 3: render inside this image, without Docker
    - Rscript -e 'rmarkdown::render(
        "analysis/report/report.Rmd",
        output_format = "html_document",
        output_dir = tempdir())'
```

The two structural differences.
Jobs and steps versus stages and scripts. GitHub Actions composes a job from reusable `uses:` steps, each a versioned action maintained elsewhere. The workflow therefore reads as an assembly of components. GitLab CI defines stages containing jobs whose bodies are shell scripts, so the pipeline reads as commands in an image. The GitHub form is more declarative and depends on an ecosystem of actions; the GitLab form is more direct and depends on nothing but the image.

Draft, November 13, 2026. Content may change.

Per-file versus single-file configuration. GitHub reads every file under `.github/workflows/`, so a project naturally accumulates several workflows with separate triggers. GitLab reads a single `.gitlab-ci.yml` at the root, so everything is expressed in one file with rules selecting what runs when.

Why a Makefile shrinks both. The differences above are entirely about *orchestration*: what runs, in what order, on what trigger. The check logic itself, run `R CMD check`, render the report, verify the manifests, is identical in both and belongs in neither file. Putting it behind named targets leaves each CI file a thin shim naming the image and calling a short list of them. The logic is then single-sourced, testable locally with the same command CI runs, and portable to a third forge for the cost of one more shim.

A zzcollab-scaffolded compendium's `Makefile` already ships targets for this, but not every target is a gate, and the file above calls only one of them. Three cautions apply. First, `check-renv` runs the dependency check with `auto_fix = TRUE` and is meant to be run before a commit. On a runner it repairs `renv.lock` in place, so it reports success for the very omission the gate exists to catch. Its sibling `check-renv-no-fix` modifies nothing and exits nonzero on failure, and only that target is a gate. Second, `check` runs `R CMD check --as-cran`, which exits nonzero on an ERROR but not on a WARNING, so an undeclared import passes it; the file therefore calls `rcmdcheck` with `error_on = "warning"` directly. Third, the render targets (`docker-render`, `docker-render-qmd`, `docker-render-book`) start a container with `docker run`. A job whose `image:` is `rocker/verse` has no Docker daemon, so the file renders inside the image itself, after the `renv::restore()` that the body's GitLab file also performs. Rendering through the project's own image belongs in a job on a runner that can run Docker, as the `verify` job above does. If you adapt this exercise to your own compendium, name the targets that match it, and confirm that each one can fail.

Draft, November 13, 2026. Content may change.

Exercise 5

Data analysis: `R CMD check` and the manifest gate always; the render gate if a report is produced. **Manuscript:** all three, since the rendered document is the deliverable. **R package:** `R CMD check` and the manifest gate; the render gate only if a vignette exists. **Simulation:** `R CMD check` and the manifest gate; the render gate if a report is produced, plus a seed-determinism check. **Blog:** the render gate above all, since the deliverable is the site. `R CMD check` and the manifest gate apply to any code the posts run.

Arranging the simulation. CI runs a reduced pipeline: the same driver, the same functions, the same seed. The replicate count is set by a parameter to something small, perhaps a hundred replicates in place of a hundred thousand. The full pipeline runs elsewhere, on a cluster, and its outputs are recorded and deposited.

What the fixed seed contributes. It is what makes the reduced run a genuine reproducibility test rather than a smoke test. With the seed fixed, the first hundred replicates of the reduced run are *the same hundred replicates* the full run begins with. Their summary is therefore a deterministic value that can be recorded and compared exactly. Without the seed, the reduced run could only be checked against a tolerance derived from Monte Carlo error. That is a much weaker statement, and one that passes even when the generator has changed.

The limit is that the reduced run verifies that the machinery reproduces, not that the full study's estimates are correct, since a hundred replicates estimate the target quantity poorly.

Exercise 6

The shared role. Both are an independent re-execution of the analysis in an environment the author does not control, performed by a party who has no stake in it succeeding. A CI runner does this for every committed change, cheaply and mechanically; a codechecker does it once for a submitted manuscript, expensively and with judgment. In both cases the value comes from the same property. The analysis runs somewhere that carries none of the author's undeclared state, so

Draft, November 13, 2026. Content may change.

anything the author depended on without knowing it is exposed.

What a human codechecker catches that the author's own CI cannot. The class of failure in which the CI configuration itself encodes the author's assumptions. CI is written by the author, so it tests what the author thought to test, in an environment the author specified, against outputs the author chose to compare. Consider a workflow that installs a system library the Dockerfile omits, or that silently skips the expensive analysis and checks only the cheap one. Consider also one that compares against a baseline the author generated after the bug was introduced. Each passes indefinitely while the compendium is not reproducible at all.

A codechecker reads the paper, forms an independent view of what the analysis is supposed to produce, and checks that the artifacts produce it. That is a judgment about whether the right thing was verified. No amount of automation supplies it, because the automation is the thing in question.

The compact statement: CI answers 'does the analysis still do what it did', and a codechecker answers 'does the analysis do what the paper says'.

Exercise 7

The solution for this exercise is in the instructor guide, which is distributed separately to course instructors. Self-study readers can check both autogradable values against the chunk below, which runs the exercise's own lines against this repository's lockfile at render time.

```
script <- '../../R/cohort.R'
locked_real <- names(renv::lockfile_read('../../renv.l
 ↪  ock')$Packages)
base_pkgs <- rownames(installed.packages(priority =
 ↪  'base'))
used <- unique(renv::dependencies(script, quiet =
 ↪  TRUE)$Package)
missing <- setdiff(used, c(locked_real, base_pkgs))
length(missing)
#> [1] 0

script <- file.path(tempdir(), 'cohort_copy.R')
invisible(file.copy('../../R/cohort.R', script,
 ↪  overwrite = TRUE))
cat('library(sandwich)\n', file = script, append =
 ↪  TRUE)
used <- unique(renv::dependencies(script, quiet =
 ↪  TRUE)$Package)
missing <- setdiff(used, c(locked_real, base_pkgs))
missing
#> [1] "sandwich"
```

Exercise 8

A CI runner tests only what is checked out from version control, which is exactly what is committed, no more. An uncommitted change exists solely on the author's own machine, so from the runner's point of view it does not exist at all. The runner cannot fail on a problem it never sees, and it cannot confirm a fix that was never pushed either. 'Commit before you compute' is therefore what makes CI's test meaningful in the first place. Without it, a green build certifies a state of the repository that nobody's actual analysis ran against.

A concrete example. An analyst fixes a bug in `analysis/scripts/02_model.R`, reruns the report locally, confirms the figure now looks right, and pushes. However, the analyst forgets to `git add` a second file, a helper function in `R/model_utils.R` the fix also depended on.

Draft, November 13, 2026. Content may change.

> Locally, everything passes, because the uncommitted helper is still on disk. On push, the runner checks out only what was committed, and the helper function is missing or stale. The render gate fails at the first chunk that calls it. The author cannot reproduce this failure by looking at their own unchanged working directory until they notice the file was never staged.

Answers to the checks

Check 1. *from earlier chapters* (page 486)

Of the two discrepancies in this warm-up, colleague A's lies in session configuration. The R release fixes the default sampler at L2, but the `.Rprofile` overrode it, and an override is fixed only by an explicit `RNGkind()` call in the analysis code (Chapter 3). The seed belongs there too, set at the call that draws the resamples, not in `.Rprofile` (Chapter 11). Colleague B's lies in the numerical libraries, the BLAS and LAPACK beneath `crossprod()` and `solve()`. No rung guarantees that layer: an image determines which BLAS is installed, but the thread count and the processor still vary (Chapter 3). The remedy is a baseline test that compares at a stated tolerance, wide enough to absorb reordered floating-point sums and too narrow for a changed coefficient to pass (Chapter 11).

Check 2. *what CI does not know* (page 489)

No. The level is set by what is captured, not by who rebuilds it, and a CI run captures nothing. Whether the compendium is at L2 at all depends on what else is committed: the rungs are cumulative, so a `Dockerfile` reaches L2 only beside a lockfile, and without one the compendium is at L0. The green CI run adds assurance rather than capture. That is, it shows the same `Dockerfile` also builds on a machine that carries none of the author's undeclared state, which the laptop build alone cannot show. A build that only ever succeeds on the author's own machine is exactly the 'works on my machine' pattern this chapter opened with.

Check 3. *the analyst's irreplaceable judgments* (page 492)

Draft, November 13, 2026. Content may change.

Any of the three qualifies, but the clearest is the first: deciding what counts as a correct output and recording the baseline. A workflow can run a comparison and report its result. However, it has no way to decide which quantity matters, which outputs to compare, or what tolerance distinguishes a real change from floating-point noise. That is a scientific judgment about the analysis, and the CI configuration cannot make it.

Check 4. *why the runner is generous nowhere* (page 493)

The runner cannot supply undeclared state because it has none to supply: it was created moments ago from a clean image and has never run anything else. It is almost a tautology because the property that makes it useful, carrying nothing of the author's, is simply what 'fresh' means. No engineering insight is required to make a runner forget what it never knew.

Check 5. *capture versus validation* (page 500)

They are correct about the level and wrong about the loss. The reproducibility level is set by what is captured: the lockfile pins the packages (L1) and the container pins the environment (L2). Continuous integration captures nothing, so removing it leaves the compendium at exactly the level it held before, L2. What is lost is not level but assurance. Without CI, no one tests, on a clean machine, that the captured environment still rebuilds. The compendium becomes trusted rather than tested. The silent decay of an external reference (a package pulled from its repository, a base image whose digest drifts) will go unnoticed until someone tries a rebuild by hand.

Check 6. *what a version tag on an action actually promises* (page 502)

Visible is not the same as fixed. A tag is a label the action's maintainer, or anyone with write access to that repository, can repoint to a different commit at any time. The workflow records only the label it asked for, not which commit it ran last time. Checking what `v3` resolves to now answers only 'what would run if I checked right now'. It does not answer what ran last week, or what will run if the tag moves before the next trigger. A commit SHA fixes the content; a tag, however carefully maintained, is a pointer that can be moved. This is exactly the distinction this book already draws between a Docker tag and a Docker digest.

Check 7. *the presence-driven trap* (page 505)

Draft, November 13, 2026. Content may change.

A rename, for instance moving `report.Rmd` to `analysis-report.Rmd` during a reorganization, or a typo in the condition itself, makes `hashFiles()` return an empty string. The condition then evaluates to false and the step is skipped. The job still reports success, because a skipped step is not a failed one. Nothing in the build output distinguishes 'rendered successfully' from 'did not try to render'. The habit that catches it is having the workflow print, for every conditional step, whether it ran or was skipped. A silent skip is then visible in the log, even though it does not turn the build red.

Check 8. *trusting a stamp* (page 509)

No. The stamp certifies that a specific, now-superseded environment reproduced the baseline, not that the environment as it currently stands does. Because the lockfile changed after the last successful verification run, the claim is stale: the thing the stamp describes no longer exists. The honest statement is that the compendium was L3 as of the old lockfile and is unverified now. This is why a change to the environment should drop the stamp automatically.

Check 9. *what to hash* (page 512)

The flaw is that a rendered PDF is not a deterministic function of the analysis. It embeds the time of rendering and the identifiers of the fonts on the rendering machine, so two renders of an unchanged analysis produce PDFs with different bytes and therefore different checksums. The student's check would report a failure on every single rebuild, catching nothing real and quickly training them to ignore it. They should instead check the computational outputs that feed the report: the estimates and the key counts should be compared with a committed baseline within a stated tolerance by a baseline test. Where a derived file's bytes should not change, its checksum should be checked as well. The render gate, separately, confirms that the document still builds from those outputs. Byte-identical PDFs are attainable with timestamp normalization and font freezing, but that is a fragile route to a result the baseline test gives directly.

Check 10. *testing the claim, not just the tool* (page 513)

Because the claim is itself a factual statement about behavior, not a design intention. Design intentions drift from implementation the same way an undeclared dependency drifts from a lockfile. That is, a later change to the scaffolding code can silently break the L1 configuration while the

Draft, November 13, 2026. Content may change.

documentation still describes the old behavior. Asserting the claim in prose is exactly the kind of trust-without-test this whole chapter argues against. The capture test matrix applies the chapter's own standard to the framework that makes the claim.

Check 11. *when a package does need the render gate* (page 514)

The render gate applies whenever the package ships a vignette, which is itself a rendered document. The gate re-renders `vignettes/intro.Rmd` (or whichever vignette files exist) on every push. It confirms that the vignette's code still runs against the restored library, exactly the check a manuscript's report receives. Only a package with neither a report nor a vignette has nothing for the render gate to do.

Check 12. *what independence adds beyond CI* (page 515)

CI is configured by the author, so it tests what the author thought to test, against a baseline the author recorded, in an environment the author specified. If the author misunderstood what the analysis was supposed to produce, or recorded a baseline after a bug had already entered it, CI will faithfully and indefinitely confirm the misunderstanding. A codechecker reads the paper and forms an independent judgment of what the analysis should produce. The codechecker then checks the artifacts against that judgment rather than against the author's own configuration, which is a check CI structurally cannot perform on itself.

Check 13. *which gate, and why* (page 517)

It is not a flaw; it is a scope limit stated earlier in the chapter. `R CMD check` runs on the package `R CMD build` produces, and `.Rbuildignore` excludes `analysis/`, so the check never parses, runs, or scans the script that calls `library(sandwich)`. The gate is doing what it is specified to do, checking the package skin. The defect lives one directory outside that skin, which is why a compendium needs a second gate that does read `analysis/`.

Check 14. *complete the workflow file* (page 519)

The four completed parts of the condensed `compendium` workflow are these. Placeholder (1) is the trigger block:

Draft, November 13, 2026. Content may change.

```
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]
```

Without `pull_request` a change is tested only after it reaches `main`; without `push` a commit made directly to `main` is never tested. Placeholder (2) is the checkout:

```
- uses: actions/checkout@3d3c42e5aac5ba805825da76410c18127
  ↪ 3ba90b1 # v7.0.1
```

The condensed file prints `actions/checkout@v7`, a teaching simplification. A tag can be moved to another commit, whereas a full commit SHA fixes the code that runs, as a digest fixes a base image. Placeholder (3) is `- uses: r-lib/actions/setup-renv@v2`. A step that ran `install.packages()` instead would install whatever versions are current, so the gates would test a library the lockfile does not describe. Placeholder (4) is the dependency-manifest call:

```
zzrenvcheck::check_packages(strict = TRUE,
                            error_on_fail = TRUE)
```

This is the gate that went red on `sandwich` in this chapter's worked example, because it reads `analysis/`. Passing `error_on_fail = TRUE` explicitly keeps the failure from depending on how R was started; with `FALSE` the step would print the missing package and still pass.

Check 15. *bisecting an environment problem* (page 521)

`git bisect` localizes a failure to a commit by checking out different revisions and testing each one, which only works when the thing that changed is in the history it walks. Here the commit is fixed and identical on both days, so bisection has nothing to vary and nothing to find; every checkout it tries is the same code. The defect lies outside the commit, most likely in the restored environment: an unpinned dependency that resolved to a different

Draft, November 13, 2026. Content may change.

version between Friday and Tuesday, or a base image whose tag moved. The right comparison is `sessionInfo()` or `renv::status()` run on both days. If the lockfile itself is suspect, it is a diff of `renv.lock` against the copy captured on Friday. It is not a bisection across commits that never changed.

Check 16. *what this demonstration does and does not show* (page 523)

Running the logic locally confirms the gate's *logic* is correct: it finds the right packages and reports the right ones as missing. However, it says nothing about whether a real workflow file invokes that logic correctly, or restores the right library before running it. Nor does it show that the job actually stops on failure rather than printing a warning and continuing. Those are the cross-boundary and `error_on` pitfalls the chapter described earlier, and only an actual CI run, on an actual fresh runner, can confirm them.

12.19. Cheat sheet

Command or rule	What it does
`error-on: '"warning"'` (GitHub) / `error_on = "warning"` (`rcmdcheck`)	Fails `R CMD check` on any WARNING, including an undeclared import; `"error"` lets it through.
`zzrenvcheck::check_packages( strict = TRUE, error_on_fail = TRUE)`	Dependency-manifest gate; passing `error_on_fail = TRUE` explicitly keeps the step's failure from depending on how R was started.
`uses: actions/checkout@<full SHA> # v7.0.1`	Pins an action to a commit; Dependabot or Renovate can propose updates as pull requests.
`renv::dependencies( 'analysis')`	Scans source for packages actually used; the core of a hand-written dependency gate.
`rmarkdown::render(...)` in a CI step	The render gate: fails the job if any chunk errors on a clean restore.

Draft, November 13, 2026. Content may change.

Command or rule	What it does
`docker build -t compendium:ci .`	Rebuilds the project image from a clean machine, the L2 rebuild `zzc verify --full` requires.
`test-baseline-*.R` naming	What `zzc verify --full` and `ci/run-tests.R` look for to certify L3.
`digest(file = path, algo = 'sha256')`	Checksums a file's bytes that should not change (raw data); the wrong tool for a floating-point estimate.
`.Rbuildignore` entry for `analysis/`	Why `R CMD check` never sees analysis scripts or their dependencies.
Rule of thumb	Verify computational outputs within a tolerance; never hash rendered PDF or HTML bytes.
Habit 4	Make every gate able to fail, and watch it fail once.
Habit 6	Rebuild from clean, on a machine that is not yours.

12.20. Further reading

The distinction between capturing reproducibility and validating it organizes this chapter. It is drawn out across the introduction of this book (Chapter 1), and it rests on the general framing of computational reproducibility as a spectrum in Peng (2011).

On independent, external verification, the CODECHECK initiative described by Nust and Eglen (Nüst & Eglen, 2021) is the most concrete model of re-execution during peer review. It merits reading for its account of what a codechecker does and does not certify. The ACM's artifact review and badging policy (Association for Computing Machinery, 2020) documents the badge scheme now common at computing venues.

On the persistence of the problem that all this machinery addresses, Stodden and colleagues (Stodden et al., 2018) analyze the effectiveness of journal

Draft, November 13, 2026. Content may change.

reproducibility policies. They find, soberingly, that a policy on paper is not a reproduction in practice. Trisovic and colleagues (Trisovic et al., 2022), in re-running thousands of deposited R files, quantify how often re-execution simply fails, which is the empirical case for the standing checks this chapter describes.

For the practices that surround the tooling, the ten simple rules of Sandve and colleagues (Sandve et al., 2013) ask for automated steps and public access to scripts, runs, and results, the preconditions of independent re-execution. They also connect the mechanics of CI to the habits that make it worthwhile.

Draft, November 13, 2026. Content may change.

Part IV.

Collaboration and Stewardship

Draft, November 13, 2026. Content may change.

13. Collaboration, Archiving, and Stewardship

The intent is that these may act as a guideline for those wishing to enhance the reusability of their data holdings.

Wilkinson and colleagues, *The FAIR Guiding Principles for Scientific Data Management and Stewardship* (2016)

By the end of this chapter you will be able to decide what belongs in a team's slow-moving environment layer and what in its fast-moving package layer. You will also be able to state plainly why the committed Dockerfile rather than the pushed image is the reproducibility guarantee, and to deposit a compendium so that it outlives the laptop, the registry, and the forge it was built on.

Before you start

Reading time. About 60 minutes.
Prerequisites. Chapter 6 (Chapter 6) supplies the renv lockfile this chapter extends to a team. The rule for that extension, the union model, under which the project lockfile records every package any contributor has installed, is introduced in this chapter. Chapter 7 (Chapter 7) supplies the digest-pinned Docker image this chapter archives, whose guarantee it relocates from the image to the Dockerfile, and a first look at the two-layer team model this chapter develops. Chapter 8 (Chapter 8), which the core path marks optional, is the source of the single-file SIF image used in the deposit; the archiving section explains the one command the deposit needs, so that chapter is not required. Chapter 3 (Chapter 3) supplies the L0-L3 ladder every honest claim in this chapter is stated against. Chapter 4 (Chapter 4) supplies the commit rule this chapter's restricted-data section makes

stricter. No new statistical knowledge is assumed; reading a Dockerfile and a short shell session is.

By the end of this chapter you will be able to answer:

- Can you explain the two-layer team model and decide which layer a given package decision belongs to under the union model?
- Can you state why the committed Dockerfile, and not the pushed image, is the reproducibility guarantee, and choose a registry and a forge with vendor independence in view?
- Can you deposit a research compendium for the long term, minting a persistent identifier and archiving the environment as a single file, and relate the steps to the FAIR principles?
- Can you recover a decayed environment from the date an analysis was run, and state the sustainability limits of a young framework honestly rather than papering over them?
- Can you adapt the sharing-and-archiving workflow to a package, a blog, and a manuscript, naming what varies and what does not?

Outline.

- Where this sits on the stack: what archiving captures and what it only validates.
- Orientation: a laboratory whose environment nobody can reconstruct two years later.
- The analyst's contribution: three judgments no tool automates.
- Sharing with a team: the two-layer model.
- Sharing with the world: archiving, and what changes when the data are restricted.
- Stewardship over time: recovering a decayed environment.
- Across the archetypes: package, blog, manuscript.
- Worked example: one laboratory's quarter of collaboration and one deposit.
- The cohort, continued: checking this book's own compendium against a deposit checklist, today.
- Collaborating with an LLM.

Draft, November 13, 2026. Content may change.

Check 1: from earlier chapters

A laboratory is about to deposit a compendium whose README claims 'L3 verified'. The repository is under git. Its Dockerfile begins `FROM rocker/r-ver:4.4.1` and installs packages with `install.packages()` from the live repository, and there is no `renv.lock`. The CI workflow runs only `R CMD check`, and its badge is green. Which rung does the compendium actually reach, and which earlier remedy would raise it by each rung in turn?
The answer is on page 591.

13.1. Where this sits on the stack

This chapter fixes no new version of anything on the determinant stack, and neither of its halves, sharing with a team and archiving with the world, adds a rung. It still does two things to the stack, and they differ in kind.

Figure 13.1 marks both. Archiving pins the *input data* layer more tightly than any earlier chapter, which is the second sense of 'captures' in the Conventions. The compendium of Chapter 5 made the data locatable in a team's repository, which a forge outage or a departed contributor can erase. A deposit moves them to a persistent identifier that continues to resolve after the project ends. The layer is marked as captured here for that reason, although no new version or byte is fixed. Stewardship, by contrast, pins nothing new about the packages or the operating system; the lockfile of Chapter 6 and the digest-pinned image of Chapter 7 already did that capturing. What stewardship does is confirm, using `rang` or an archived image, that those pins still resolve after they have aged. It validates layers already captured elsewhere.

```
plot_determinant_stack(
  captures = c('Input data'),
  validates = c('Package versions',
                'Operating system, system libraries'),
  title = 'Where archiving and stewardship sit on the
   ↪  stack'
)
```


Draft, November 13, 2026. Content may change.

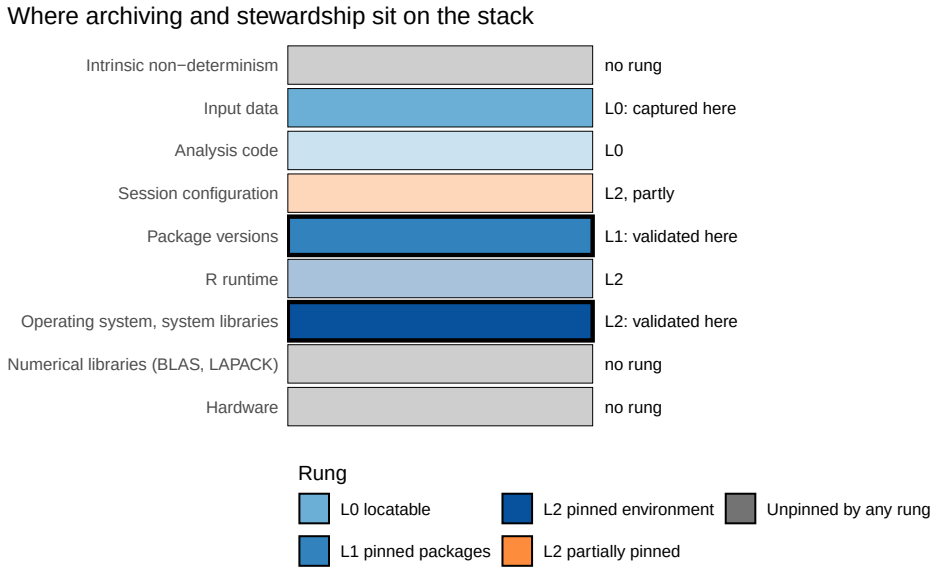


Figure 13.1.: The determinant stack of Chapter 3, with input data marked as captured here and the package-version and operating-system layers marked as validated. This chapter pins no new byte: the packages are captured in Chapter 6 and the operating system in Chapter 7. What it adds to the data layer is durability, lifting it from merely locatable on a forge to findable under a persistent identifier, which is why input data is captured rather than validated here; what it adds to the package and system layers is the stewardship check, using an archived image or a date-based reconstruction, that those earlier pins still resolve, which is why they are validated rather than captured again.

13.2. Orientation

The last question a reproducible analysis must answer is a social and temporal one. Every earlier chapter was concerned with a single machine and a single analyst. We saw how to pin the packages (Chapter 6), fix

Draft, November 13, 2026. Content may change.

the environment (Chapter 7), and render the report (Chapter 9). We then saw how to validate the whole with tests and continuous integration (Chapter 12). The present chapter concerns what happens when the analysis must leave that machine. It must first reach a collaborator down the hall, and eventually reach a stranger, a reviewer, or the analyst's own future self, years after the original laptop has been recycled.

Consider a concrete case. A five-person laboratory runs a multi-site observational cohort. The principal investigator fixes the computational environment at the project's start. Over the following year, one contributor adds survival-analysis packages, a second adds table-formatting packages for the manuscript, and a third adds spatial packages for a geographic sub-study. Each installs freely, on their own machine, without asking the others' permission. Two years after the paper appears, a journal requests the analysis for a reproducibility audit. The original environment must then be reconstructed from whatever survives in the repository. Nothing in the single-analyst workflow of the earlier chapters tells us how the laboratory should have coordinated during that year. Nor does it tell us how the compendium should have been deposited so that the audit could succeed.

These are two distinct problems, and we shall keep them apart. The first is collaboration: sharing a compendium with a team while it is under active development, so that contributors do not drift into mutually incompatible environments. The second is archiving and stewardship: depositing the finished compendium so that it remains findable and executable long after the project ends, and recovering it when the pins have decayed. Both sit at the top of the reproducibility ladder and add no new rung. Sharing a compendium changes nothing about its level: an L2 compendium shared with a team is still an L2 compendium, and a verified L3 compendium deposited in an archive is still L3. What sharing and archiving add is the assurance that the level already reached survives contact with other people and with the passage of time. That assurance is the subject of this chapter. Being the last chapter, it is also where we draw the threads of the whole book together.

We begin with the three judgments that remain the analyst's own. We then take up sharing with a team through the two-layer model, and sharing with the world through archiving, including what changes when the data are restricted. Stewardship over time, and the recovery of a decayed

Draft, November 13, 2026. Content may change.

environment, follow. Lastly, we consider the archetypes, a worked example, and a check of this book's own compendium against a deposit checklist.

Check 2: two problems, not one

The laboratory's L2 compendium is shared with a new contributor, who extends the package layer, and is later deposited in Zenodo. Does either event change the level the compendium has reached? What does each event add if not a level?
The answer is on page 592.

13.3. The analyst's contribution

Before turning to the mechanics, we note that three judgments in this chapter are the analyst's own, and no tool makes them.

1. **Where to draw the governance boundary, between what the team fixes and what each member may vary.** The framework offers the two-layer model as a default. However, only a person who understands the project can decide which layer a given decision belongs to. Fixing too much makes the team rigid and slows every contributor; fixing too little lets the environment fracture. The base image, the R version, the system libraries, and the packages every project in the lab uses are almost always the lead's to fix; the packages one project's analysis needs are almost always the members' to add. The awkward cases in between (a version of a system library that one sub-study needs and the others do not) are decided by the analyst, not the tool.

2. **When to deposit, at what depth, and at what level to claim.** An archival deposit is a public and permanent act, and the honest disclosure of the level reached is part of the deposit. Depositing a compendium and labeling it 'reproducible' without stating whether it has been verified, or whether the environment was pinned by digest or merely by tag, misrepresents the work. The analyst decides when the compendium is ready and which artifacts accompany it. The analyst also decides which data the consent and the data-use agreement

Draft, November 13, 2026. Content may change.

allow to accompany it, and, above all, what the accompanying claim honestly says.

3. **Where the work lives, and how dependent that choice makes it.** A registry and a forge are conveniences, but each is also a dependency on a commercial vendor whose terms may change. The analyst weighs the convenience of a popular platform against the vendor independence of an institutional or self-hosted alternative. The analyst does so knowing that a compendium tied to a service that disappears is a compendium at risk.

The vocabulary of this chapter

- **Registry.** A server that stores and distributes container images, from which collaborators pull a pre-built environment. GitHub Container Registry (GHCR) and Docker Hub are the two in common use.
- **Forge.** A platform that hosts a version-controlled repository and runs its continuous integration. GitHub and GitLab are the two principal forges; GitLab is frequently self-hosted by universities.
- **Two-layer team model.** A division of a project into a slow-moving environment layer, fixed once by a team lead, and a fast-moving package layer, extended by every contributor. The environment layer is the team image: the operating system, the R version, the system libraries, and the packages every project in the lab uses, pinned by digest. The package layer is the project lockfile.
- **Union model.** The rule that the project lockfile records the union of every package any contributor has installed, so that a rebuilt image carries what the whole team needs, not only what one member used.
- **Persistent identifier (DOI).** A durable, resolvable name for a deposited artifact, such as a Digital Object Identifier, that continues to resolve even if the artifact moves.
- **FAIR.** A set of principles holding that a research object should be Findable, Accessible, Interoperable, and Reusable, formulated for data and later adapted for research software.

- **OCI tarball or SIF file.** A container image exported as a single file, checksummable and depositable as an archival artifact. An OCI image is exported as a tar archive, whereas SIF is the self-contained single-file format used by the Apptainer runtime.
- **Zenodo.** A general-purpose research repository, operated at CERN, that mints a DOI for any deposited file set.
- **Controlled access.** A repository arrangement in which data are released only to approved users under an agreement, rather than to anyone who downloads.
- **Data-availability statement.** The statement in an article or README of what was deposited, where, and how any data not deposited may be obtained.

Check 3: whose judgment, again

A compendium is ready to deposit, by every mechanical check: the tests pass, the lockfile is current, the image builds. Name one judgment from this section that a checklist still cannot make, and say why it cannot.
The answer is on page 592.

13.4. Sharing with a team: the two-layer model

We begin with collaboration. The design starts from one observation: the two things a team shares change at very different rates. The computational environment (the operating system, the R version, the system libraries, and the packages every project in the lab uses) is chosen at most a handful of times over the life of a project. The set of packages one project needs, by contrast, changes almost weekly as the analysis grows. A model that forces both to be renegotiated on the same schedule will either freeze the packages to the cadence of the environment, which is too slow, or churn the environment to the cadence of the packages, which is too disruptive. The resolution, adopted by the framework this book uses and set out by its authors, is to separate the two into layers with different change rates and different governance.

Draft, November 13, 2026. Content may change.

Layer 1 is the environment, and it is lead-governed. A designated team lead selects a base image at the project's start and builds on it a team image carrying the packages every project in the lab uses, as Chapter 7 sketched. The lead resolves that image to an immutable content digest and fixes it for every collaborator. This is the slow-moving half. It corresponds to pillar one, the computational environment, and it is where the L1-to-L2 transition of the ladder is secured. The lead builds this environment once, and the whole team reuses it. The vignette that describes the pattern calls this 'build once, use everywhere'. It reports that reusing a pre-built team image reduces the setup of a new project from the fifteen or twenty minutes of a full build to a matter of seconds. For reproducibility, the larger gain is that every collaborator works from the same fixed floor.

Layer 2 is the project's package set, and it is contributor-governed under the union model. Each member installs, on top of the team image, the packages their part of the work requires, freely and without coordination. The project lockfile at any moment records the *union* of everything anyone has installed. The shared image is rebuilt from that lockfile periodically rather than on every change. When it is, it carries the full set every contributor needs, not merely the subset one member happened to use. A collaborator who pulls the latest main branch and rebuilds therefore inherits the survival packages, the table packages, and the spatial packages all at once. This holds whether or not their own work touches any of them. This is the fast-moving half, corresponding to pillar two, and it is governed by every member.

The union model buys the elimination of a common and maddening failure, the analysis that runs for its author and nowhere else. The price is a little environmental bloat. A package installed once for an exploratory notebook remains in the lockfile for everyone until it is deliberately removed. In our view this is a sound trade. On the one hand, the bloat is bounded and visible, a few extra megabytes in an image that is already a gigabyte or more, and it can be pruned when it matters. On the other, the failure it prevents is a contributor blocked because a colleague forgot to record a dependency. That is the precise 'works on my machine' pathology that the whole enterprise exists to abolish. We would rather carry an unused package than lose a day to a missing one.

Draft, November 13, 2026. Content may change.

13.4.1. The Dockerfile, not the image, is the guarantee

The pushed team image is a convenience, a cache that saves build time; it is *not* the reproducibility guarantee. The guarantee is the Dockerfile, together with the lockfile it restores from, both committed to version control. The reason is that a pushed image is opaque and impermanent. It is a binary blob on a registry that may be deleted, or pruned under a retention policy. It may be pushed for one processor architecture and silently emulated on another, and it cannot be inspected line by line to see how it was made. The Dockerfile is transparent and durable: it is a text recipe, readable and diffable, that reconstructs the image from named and pinned inputs. If the registry vanishes tomorrow, a compendium whose guarantee is the Dockerfile can be rebuilt; a compendium whose guarantee was the pushed image cannot.

The framework's vignette makes the same point in a note that merits restating in our terms. Pushing the image to a registry is optional, and if the image is not available, building from the Dockerfile reproduces the environment. We should qualify that claim, in keeping with this chapter's own discipline. A rebuild reproduces the same content (the same base image, R version, and package versions) only when the base image is pinned by content digest and the package snapshot is dated, as Chapter 7 urged. A recipe pinned by a mutable tag, or drawing on a rolling snapshot, reconstructs at best a functionally equivalent environment whose content may have drifted. Even a correctly digest-pinned, dated-snapshot rebuild is not guaranteed byte-for-byte identical at the image level, because per-layer timestamps ordinarily differ between builds. Matching content is the guarantee a pinned recipe gives, and bit-identical bytes require separate reproducible-build tooling (Chapter 7). A team that treats the pushed image as authoritative, and lets the Dockerfile drift out of step with it, has converted its compendium from something reconstructable into something that merely happens to exist. The committed lockfile still holds L1, but the environment layer is no longer captured by anything the compendium contains, so the compendium has regressed below L2. That regression is what honest disclosure is meant to catch.

Draft, November 13, 2026. Content may change.

13.4.2. Registries and the forge

Two distribution choices attend team sharing, and both are concerns about vendor independence. Neither changes the level a compendium has reached.

The **registry** is where the team image is published: GitHub Container Registry, Docker Hub, or an institutional registry such as one hosted inside GitLab. Docker Hub is the most familiar. However, its free tier rate-limits image pulls and permits only a single private repository, constraints that a small laboratory reaches quickly. GHCR integrates with the GitHub forge and has more generous terms for public research. An institution with confidentiality requirements may mandate a private or self-hosted registry. The choice is a trade between convenience and control, and, we would stress, it is reversible. Because the guarantee is the Dockerfile, a team can re-push its image to a different registry at any time without altering the compendium.

The **forge** is where the repository and its continuous integration live: GitHub, or, very often in the health sciences, an institutional GitLab. This choice matters because research compendia are disproportionately authored on self-hosted GitLab. Many universities and laboratories, particularly those subject to on-premises data-governance rules, run their own GitLab and cannot push to a public forge. A framework whose premise is reproducibility for that audience must not lock its users into one forge. The practical difficulty is concentrated entirely in continuous integration. GitHub Actions and GitLab CI use structurally different configuration dialects. Supporting both, without maintaining two parallel and divergent sets of workflow files, therefore requires that the CI logic be single-sourced, typically in the Makefile. Each forge's configuration is then reduced to a thin shim that calls it. The analyst should therefore treat the forge as a replaceable host and keep the reproducibility logic in artifacts that travel between forges unchanged.

Check 4: what survives a registry outage

A colleague argues that the team should push its image to Docker Hub after every package installation, 'so the image is always the source of truth'. What is wrong with treating the pushed image as the source

Draft, November 13, 2026. Content may change.

of truth, and what should be the source of truth instead?
The answer is on page 592.

13.5. Sharing with the world: archiving

Sharing with a team keeps a live project consistent. Archiving asks a different question: how does a finished compendium survive after the project ends, the collaborators disperse, and the laboratory's servers are decommissioned? A repository on a forge is not an archive. Forges delete inactive accounts, rename themselves, change ownership, and go out of business; a URL that resolves today may not resolve in a decade. An archive, by contrast, undertakes to preserve a deposit and to keep a name for it resolving indefinitely.

The instrument is a deposit in a research repository that mints a **persistent identifier**. Depositing the compendium in Zenodo, the general-purpose repository operated at CERN, produces a Digital Object Identifier. The DOI continues to resolve even as the underlying storage moves, and it can be cited in the article the compendium supports. The DOI is what lifts a compendium from merely locatable on a forge, which is fragile, to durably findable, which is the archival requirement.

A deposit should carry the environment as well as the source. Here the container runtimes of Chapter 8 supply the archival artifact needed. Apptainer, the high-performance-computing runtime, builds a single Singularity Image Format file from a digest-pinned Docker image. That SIF, or equivalently an OCI image exported as a tarball, is checksummable and depositable as one object. The deposit needs one command, `apptainer build env.sif docker://<image>@sha256:<digest>`, which pulls the image by its digest, flattens its layers, and writes them to a single read-only file. A reader who skipped Chapter 8 needs nothing more for this chapter, and a reader without Apptainer can write the tarball with Docker alone, using `docker save`. A compendium archived as its source tree, a pinned Dockerfile, and the built environment as a single file is reproducible by two independent routes. A reader may rebuild from the recipe or, if the recipe's inputs have decayed, may run the archived environment directly. Depositing both is belt and suspenders, and for a result that matters it is worth the disk. What a

Draft, November 13, 2026. Content may change.

deposit may carry of the *data* is a separate question. It is governed by the terms under which the data were obtained, rather than by the analyst's wish to be reproducible. For patient-level data the usual answer is that the data stay out of an open deposit, and Section 13.5.2 treats what is deposited in their place.

These practices realize a set of principles that the community has crystallized under the acronym FAIR. Wilkinson and colleagues proposed that a research object should be Findable, Accessible, Interoperable, and Reusable, and framed the principles for research data (Wilkinson et al., 2016). Barker and colleagues subsequently adapted them for research software, whose reuse involves execution as well as reading (Barker et al., 2022). A DOI serves findability. A repository with a stated access procedure serves accessibility, which for restricted data means a controlled route rather than an open download. A documented, standard container format serves interoperability, and a pinned, rebuildable environment serves part of reusability. Reusability also requires a clear usage license (R1.1 in the original principles), which a pinned environment does not by itself supply. In this light, the archetype-agnostic compendium of Chapter 5 is most of the way to a FAIR object by construction. This holds provided it is deposited rather than merely committed and carries an explicit license; licensing practice is treated separately.

13.5.1. Licensing, CITATION.cff, and Software Heritage

A pinned environment and a minted DOI satisfy three of FAIR's four letters without touching the fourth's hardest requirement. Reusability, principle R1.1 in the original statement, asks for 'a clear and accessible data usage license' (Wilkinson et al., 2016). Nothing about findability, accessibility, or interoperability supplies one. A compendium can be perfectly findable and still carry no statement of what a reader is permitted to do with it. In the absence of a license, that defaults, under copyright, to nothing. 'The cohort, continued' below returns to what license this book's own compendium carries.

Two further instruments round out what a deposit should carry, beyond the license file itself.

Draft, November 13, 2026. Content may change.

CITATION.cff. A `CITATION.cff` file is a plain-text, machine-readable citation record placed at a repository's root, using a small set of `key: value` fields (`cff-version`, `authors`, `title`, `version`, and `date-released` among them) (GitHub Docs, 2024). GitHub parses the file, when it sits on the default branch, into a 'Cite this repository' link on the repository page. The link generates an APA or BibTeX citation on request (GitHub Docs, 2024). Zenodo can read the same file when it archives a GitHub release and use it to pre-fill the deposit's own citation metadata, so a `CITATION.cff` written once serves both the forge and the archive. What it pre-fills is only what the file says, the `license` field included. A compendium with one license for its prose and another for its code cannot record in that field which license covers which files, so the license on the resulting deposit is checked against `LICENSE` before it is published. Without a `CITATION.cff`, a reader who wants to cite the compendium constructs a citation by hand from whatever metadata happens to be visible on the page. This is the same fragility that motivated the DOI in the first place.

Software Heritage. Zenodo is a deposit repository: an institution, CERN, operates it. Its continuation is a reasonable expectation, but it is an expectation and not a guarantee, since a deposit repository could in principle be discontinued by the institution that runs it. Software Heritage undertakes something narrower and more absolute for source code specifically. Its stated mission is a permanent, universal archive of publicly available source code. A depositor cannot withdraw what is archived; removal happens only through a takedown procedure that the project's staff carry out. The project's SWHID identifier scheme was formalized as an ISO/IEC standard in 2025 (OpenChain Project, 2025). Every archived object, whether a file, a directory, a commit, or a full repository snapshot, receives a SoftWare Hash IDentifier of the form `swh:1:<type>:<hash>`. Here `<type>` names the kind of object (`cnt` for a file's content, `dir` for a directory, among others) and `<hash>` is a 40-character value computed from the object's own bytes (Software Heritage, 2024). The identifier is computed from the content rather than assigned by a registry. Two copies of the same file anywhere in the archive therefore resolve to the same SWHID. This is a different kind of guarantee from a DOI: a DOI names a deposit, and a SWHID names the bytes.

This book's own repository illustrates the gap between having these instruments and using them well. A `CITATION.cff` is present at the repository

Draft, November 13, 2026. Content may change.

root, generated by zzcollab when the project was created. Its `license` field reads CC-BY-4.0, the license of the prose, while `LICENSE` places the code under MIT and GPL-3; the file's abstract states all three licenses in words, but a deposit service reading the `license` field sees only the one identifier. Its `orcid` field is still empty, so the record names its author without a persistent identifier. The file's presence is enough for GitHub to render the 'Cite this repository' link and for Zenodo to read it on deposit. Whether its content credits the right people under the right license is a question only a reading of the file answers.

Two broader efforts merit the analyst's attention. The Whole Tale platform, described by Brinckman and colleagues, bundled code, data, and computational environment into a single executable research object that a reader could run in the browser without local installation (Brinckman et al., 2019). Its public service did not respond when we checked in November 2026. It remains an ambition the compendium-plus-archived-image pattern approaches by other means. And the Turing Way, a community-authored handbook, collects the collaborative and stewardship practices of reproducible research into one continually updated reference (The Turing Way Community, 2022). Some readers will want the social side of collaboration, from code review to project governance to credit, in more depth than a single chapter allows. They should begin there.

13.5.2. When the data are restricted

Everything said so far about depositing assumes that every file in the compendium may be published. In health research that assumption is usually false. A compendium built on patient records contains, or was built from, such data. Their release is governed by law, by the consent the participants gave, and by the data-use agreement under which the analyst received the extract. A deposit in an open repository is a publication to anyone, indefinitely, and a file placed there cannot be recalled.

! Restricted data

The rule is the commit rule of Chapter 4, made stricter. Patient-level data do not go into an open deposit unless three conditions hold.

They must have been formally de-identified, and the consent and the agreement must both permit public release. The responsible data steward or IRB must also have confirmed it in writing. In their place, a deposit may take four routes:

- the code, the tests, the lockfile, the Dockerfile, and the archived image, always;
- de-identified data, only where the data's steward has made that determination;
- a synthetic stand-in, labeled as such;
- for data that may be shared under agreement but not opened to the world, a citation to a controlled-access repository such as dbGaP.

None of these substitutes removes the obligation to read the consent form and the data-use agreement first. The deposit's data-availability statement must say, exactly, which of these routes was taken.

The paragraphs below work through each of the four routes named above, and the two documents that govern the choice, in enough detail to act on.

The code and the environment, always. The analysis code, the tests, the lockfile, the Dockerfile, and the archived image carry no participant data. Depositing them openly lets anyone inspect the analysis, and lets anyone who obtains the data by the legitimate route re-run it. Two checks come first. The archived image must not contain the data. A `COPY . .` executed while the extract sat in the build context places it inside the image. A `.dockerignore` that excludes the data directory, and an inspection of the built image's file system, are therefore part of the deposit. And the other artifacts must not carry individual-level rows by another route. Rendered reports, cached chunk output, saved workspaces, logs, and test fixtures are all places where records leak. Tables with very small cell counts can disclose as surely as a row does.

De-identified data, where permitted. Under the HIPAA Privacy Rule, health information ceases to be individually identifiable by one of two routes. In the first, a qualified expert applies statistical methods and documents that the risk of identification is very small. In the second, the eighteen listed identifiers are removed and the covered entity has no actual knowledge that

Draft, November 13, 2026. Content may change.

the remainder could identify anyone (*45 CFR 164.514*, n.d.). A limited data set, which keeps dates and some geography, is not de-identified and may be shared only under a data-use agreement (*45 CFR 164.514*, n.d.). NIH guidance adds that data de-identified in the legal sense may still warrant controlled access (National Institutes of Health, 2026c). De-identification is a determination made by the data's steward, not something an analyst achieves by deleting the name column.

Synthetic data, as a stand-in. A synthetic file with the same columns, types, and plausible ranges as the analysis data lets a reader run the pipeline from end to end. The reader can confirm that the code executes, though not that the published estimates regenerate. It should be labeled synthetic in the file and in the README, with the script that generated it. A synthetic file produced by a model fitted to the real records needs the same disclosure review as the records themselves, since such a model can reproduce rare individuals. One simulated from stated parameters, as this book's running cohort is, does not.

Controlled-access repositories. When data may be shared but not publicly, a controlled-access repository holds them and releases them only to approved users under an agreement. NIH's dbGaP, for genotype and phenotype data, is an example: public study descriptions sit beside an authorized-access route with its own security procedures and code of conduct (National Center for Biotechnology Information, n.d.). The open deposit of code then cites the controlled-access accession, so that the two halves are linked by identifier. The NIH Data Management and Sharing Policy has been in force since 25 January 2023. It requires access controls for data with explicit limits on later use (National Institutes of Health, 2020). NIH's privacy guidance adds that those limits should travel with the data and be communicated to the repository (National Institutes of Health, 2026c).

Consent and agreement terms, read before the analysis begins. What may be deposited is settled by documents signed long before the deposit: the consent form, the IRB protocol, and the data-use agreement. The analyst reads them for redistribution, derived data, public posting, and retention. An agreement that requires the extract to be destroyed at the end of the project governs every copy of it, including those in backups, caches, and container images. NIH guidance asks that consent explain how data will be shared and that use agreements carry their restrictions with

Draft, November 13, 2026. Content may change.

the data (National Institutes of Health, 2020, 2026c).

The data-availability statement. The article and the README state what was deposited, where, and how the rest may be obtained, with the same exactness the honest-level claim demands. A statement of the following form is typical:

> The analysis code, package lockfile, container recipe, and archived container image are openly available at [DOI], with a synthetic dataset of the same structure on which the pipeline runs from end to end. The patient-level data were obtained from [registry] under a data-use agreement that does not permit redistribution; qualified researchers may apply to [data steward] for access. The synthetic data do not reproduce the reported estimates.

The last sentence bears on the level claim. A reader without access to the data can confirm that the pipeline executes. Only someone with access can confirm that it regenerates the published numbers, so an L3 claim must say who performed the comparison and on what data.

Check 5: which route, and why not the obvious one

A colleague proposes removing the name and medical record number columns from a patient-level extract and depositing the rest openly, calling the result 'de-identified.' What is wrong with this plan, and which of the four routes above should be used instead?
The answer is on page 593.

13.5.3. Working inside a trusted research environment

The four routes above assume that the analyst holds a copy of the data and must decide what to release. For data whose provider never releases a copy, that assumption fails. A trusted research environment, also called a secure data environment, a secure processing environment, or a data enclave, reverses the direction of travel. The data stay on the provider's infrastructure, and the analyst brings code to the data rather than data to the code. The NIH *All of Us* Researcher Workbench is a US example. Its

Draft, November 13, 2026. Content may change.

documentation states that cohort building and analysis take place solely on the Workbench, which supplies RStudio and Jupyter among its tools (National Institutes of Health, All of Us Research Program, n.d.-b). In the European Union, the European Health Data Space Regulation entered into force on 26 March 2025. Secondary use under it requires a permit from a health data access body. Processing may then take place only in a secure processing environment, from which users can download only non-personal data (Article 73). The rules on secondary use apply to most data categories from March 2029 (European Commission, n.d.; European Parliament and Council of the European Union, 2025). In the United Kingdom, the SATRE specification, an open community standard whose release 2.0.1 is dated July 2026, sets out what such an environment should provide (SATRE Community, 2026).

The Five Safes. A common way to describe the controls of such an environment is the Five Safes framework, which originated at the UK Office for National Statistics (UK Data Service, n.d.). Safe projects are approved by the data owner as serving the public good. Safe people are trained and authorized researchers. A safe setting is an environment that itself prevents unauthorized use. Safe data have been treated to protect confidentiality. Safe outputs are those screened and approved as non-disclosive before release.

Output checking. The last of the five is the one that shapes the analyst's daily work. Nothing leaves the environment, whether a table, a figure, a log, or a fitted model, until it has passed statistical disclosure control. SATRE makes two things mandatory for a provider: a documented process for disclosure control of outputs, and a process that assigns responsibility for output checking. It also recommends that each project's intended outputs be established at the outset (SATRE Community, 2026). The rules a checker applies are concrete. Under the *All of Us* dissemination policy, no participant count from 1 to 20 may be published, nor any figure, percentages included, from which such a count could be derived (National Institutes of Health, All of Us Research Program, n.d.-a). This is the small-cell hazard named under the first route above, now enforced by someone other than the analyst.

A reproducible pipeline is what makes this review feasible. A checker can judge an output only when its provenance is plain: which script produced it,

Draft, November 13, 2026. Content may change.

from which cohort, and with what suppression. A pipeline that writes every exportable table from a script, applies the threshold in code, and records what it suppressed turns each request into a small, checkable package. A revision the checker asks for is then a re-run, not a hand edit of a table whose origin nobody can reconstruct.

What reproducibility can mean inside. The environment belongs to the provider. The analyst cannot pin its operating system, may not be permitted to run Docker, and may reach only the package sources the provider chooses to offer. SATRE lists access to public package repositories, and tight control of which packages are available, as options for the provider rather than obligations (SATRE Community, 2026). The ladder still applies, with a change of owner at the environment layer. The lockfile records the package versions, restored against whatever mirror the environment offers. The operating system and R runtime are the provider's, so the compendium records the provider's image or environment identifier, read from inside the environment, in place of a Dockerfile of its own. A claim of L2 is honest only to the extent that this identifier names a fixed environment. What then enters the open deposit is the code, the lockfile or that identifier, a synthetic extract of the same structure, and the release record of what passed output checking and when. SATRE lists a machine-readable record of the disclosure checks applied to each output as an option (SATRE Community, 2026). L3 is the rung that cannot travel. The comparison with the published numbers can be run only inside the environment, by an approved user, so the level claim must say who ran it and where. The data-availability statement names the environment and its application route, as the controlled-access route above names an accession.

13.5.4. Funder and journal obligations

Everything the four routes above implement is a response to an obligation that exists before the analysis begins: a funder's data-sharing policy and, later, a journal's reporting requirement. Both predate the deposit and shape what the deposit must contain.

The funder side. The NIH Data Management and Sharing Policy applies to proposals due on or after 25 January 2023. It requires that an applicant submit a Data Management and Sharing Plan at the time of application,

Draft, November 13, 2026. Content may change.

not drafted after data collection is underway (National Institutes of Health, 2020). Applications submitted on or after 25 May 2026 use an updated Plan format (National Institutes of Health, 2026b), and career development (K) awards no longer need a Plan (National Institutes of Health, 2026a). The Plan goes in the 'Other Plan(s)' attachment of the research plan; only a brief summary and the associated costs go in the budget and its justification (National Institutes of Health, n.d.). NIH program staff assess the Plan. Peer reviewers may comment on its budget, but those comments do not affect the overall score. Once an award is made, the approved Plan becomes a term and condition of that award. Noncompliance can bring added special terms and conditions or termination of the award, and can affect future funding decisions (National Institutes of Health, 2020). For the analyst, the choice among the four routes above was therefore made, in outline, when the grant was written; the deposit executes the Plan.

The journal side. A code-availability statement is a narrower instrument than the data-availability statement of this section, above. It says where the *analysis code* lives, under what license, at what version, and what it does not include. It does so independently of whatever the data-availability statement says about the data. The two statements answer different questions, and a manuscript needs both. A reader who wants to re-run the pipeline reads the code statement, and a reader who wants to obtain the inputs reads the data statement.

Some reporting guidelines now name code availability as a checklist item rather than leaving it to a journal's general policy. TRIPOD+AI is the 2024 extension of the TRIPOD reporting guideline to prediction models built with regression or machine learning. Under its 'Open science' section it lists item 18f, 'Provide details of the availability of the analytical code,' and adds a footnote. The footnote specifies that this covers the analysis code, including any data cleaning, feature engineering, model building, and evaluation (Collins et al., 2024). A separate item, 18e, asks the same question of the data. Naming code availability as its own item, distinct from data availability, is what makes TRIPOD+AI concrete where a general journal policy is often vague: an author completing the checklist cannot satisfy 18f by describing only the data.

A code-availability statement that meets this bar states four things. The first is where the code is deposited, with a persistent identifier if one

Draft, November 13, 2026. Content may change.

was minted. The second is under what license it is released. The third is which version, commit, or tagged release corresponds to the reported results, since a repository's default branch moves on after publication. The fourth is what, if anything, the deposit withholds, and why (a proprietary dependency, institution-licensed software, or data referenced only by an accession number). 'Available on GitHub,' without a version and a license, answers none of the four.

For randomized trials, the two guidelines this audience completes most often took the same step in 2025. SPIRIT 2025, for trial protocols, and CONSORT 2025, for trial reports, each gained a new 'Open science' section (A.-W. Chan et al., 2025; Hopewell et al., 2025). In SPIRIT 2025, item 5 asks where the protocol and the statistical analysis plan can be accessed. Item 6 asks where and how the de-identified participant data, with their data dictionary, the statistical code, and any other materials will be made accessible. CONSORT 2025 asks the same of the finished report. Its item 3 now asks where the statistical analysis plan, as well as the protocol, can be accessed. Its new item 4 asks where and how the individual de-identified participant data, with their data dictionary, the statistical code, and any other materials can be accessed. Two consequences follow. The protocol promises, under SPIRIT item 6, what the report must deliver under CONSORT item 4, so the code-sharing route is fixed before enrollment, much as the NIH Plan fixes the deposit route. And the statistical analysis plan must itself be accessible, so the compendium should hold or cite it beside the code that implements it.

Journal policies vary more than the guidelines do. Table 13.1 summarizes three that biostatistics and epidemiology authors commonly meet, each read from the journal's own pages on 8 November 2026. Policies change, so the target journal's current instructions, not this table, govern a submission.

Draft, November 13, 2026. Content may change.

Table 13.1.: Code-availability policies of three journals, as stated on each journal's author pages on 8 November 2026.

Journal	Code and data	Review of the materials
Journal of the American Statistical Association	Code, data, and a workflow that reproduces the work are required at invited revision: for Applications and Case Studies since 2016, for Theory and Methods since 2021. An Author Contributions Checklist form is required at submission (Journal of the American Statistical Association, n.d.).	Usually carried out by an associate editor for reproducibility (Journal of the American Statistical Association, n.d.).
Biostatistics	Sharing of data and code is strongly encouraged where ethically feasible. Papers are marked D when the data are freely available, C when the code is, and R when both are (Biostatistics, n.d.).	None described on the author-guidelines page.
American Journal of Epidemiology	A data-availability statement is required. Sharing of data and code is strongly encouraged where ethically feasible (American Journal of Epidemiology, n.d.).	None described; editors may request a statistical review (American Journal of Epidemiology, n.d.).

Draft, November 13, 2026. Content may change.

13.6. Stewardship over time

Archiving preserves the artifacts; stewardship is the longer discipline of keeping them runnable as the world underneath them changes. Even a carefully pinned compendium decays. A base image tag may be re-pushed. A package version pinned in a lockfile may age out of the repository that once served it, and a system library may be dropped from a new operating-system release. The question stewardship answers is this. When the pins have decayed and the archived image was lost or never built, can the environment still be recovered?

For the R ecosystem a useful recovery tool is `rang`, described by Chan and colleagues (C. Chan & Schoch, 2023). (`rang` was archived on CRAN on 2026-08-22. The historical versions remain installable from the CRAN archive or from its GitHub repository. The technique it illustrates is worth knowing even if the package itself needs an alternative source or a successor tool by the time a reader needs it.) The ordinary workflow pins prospectively, by recording a lockfile at development time and restoring from it. `rang`, by contrast, reconstructs an environment retrospectively, from a *date*. Given the date an analysis was run, it resolves the package versions current on the archives at that date. It then generates a Dockerfile that can often reconstruct a period-correct environment even when no original lockfile survives. The two approaches are complementary on a time axis. The framework of this book enforces consistency prospectively over a project's active life, and `rang` provides a recovery path retrospectively when that consistency has lapsed. A prudent steward records the run date in the compendium so that this recovery path stays open.

> **!** The honest level
>
> Sharing and archiving themselves reach no rung, since preserving a level is not capturing or verifying one. The single failure this chapter exists to prevent is depositing a compendium no better than locatable, with a DOI and a tidy README, and calling it 'fully reproducible' in the article the deposit outlives.

We must extend the same honesty to the tools this book has taught, and to the framework it uses as its running example. Frameworks that compose

Draft, November 13, 2026. Content may change.

the whole determinant stack are young. Measured against the 9 layers of the determinant stack in Chapter 3, the framework firmly pins roughly four: the operating system and system libraries, the R runtime, the package versions, and the analysis code, with much of the session configuration besides. It leaves the rest (the hardware, the numerical libraries, the input data, and intrinsic non-determinism) to the base image, to the analyst, or to convention. Its own system review describes it candidly as a strong scaffolding tool rather than a finished guarantor of reproducibility. The review also notes the sustainability risk that attends a small maintainer base: a single principal author is a bus factor of one, and the tool's future depends on that one person. The appropriate response is to disclose its reach accurately, the same discipline the honest-level callout demands of any analysis. The evidence that disclosure cannot be taken for granted is empirical. Stodden and colleagues tested a journal policy requiring code and data on request. They obtained artifacts for 44 percent of 204 articles and reproduced the findings of 26 percent (Stodden et al., 2018). A gap between the reproducibility a project claims and the reproducibility it delivers is common.

Check 6: recovering a decayed environment

An analysis was run in 2024 and deposited as a source tree and an `renv.lock`, but no Docker image was ever built, and by 2029 several of the pinned package versions are no longer installable from CRAN. A reviewer asks you to reproduce it. Which tool addresses this situation, what single piece of recorded information does it most need, and why is this a stewardship failure rather than a capture failure?
The answer is on page 593.

13.7. Across the archetypes

We turn now to how the machinery varies, or does not, across the archetypes. The team and archiving machinery is archetype-agnostic: the two-layer model, the union lockfile, the registry and forge choices, the DOI deposit, and the `rang` recovery path apply to every archetype; Table 5.2 in Chapter 5 is the reference. What varies is the final act of publication. A **data-analysis** or **simulation** compendium is deposited as itself: the compendium with its

Draft, November 13, 2026. Content may change.

DOI *is* the published object. A **manuscript** compendium is deposited *beside* the article it supports, cited by DOI from its data-availability statement. A **package** compendium has a further, ecosystem-native channel, release to CRAN or an R-universe, whose incoming checks are themselves a validation gate. A **blog** compendium is deployed as a website, with tools such as `workflowr` (Blischak et al., 2019) binding each rendered page to the commit that produced it. The archival obligation still applies to the compendium behind the posts. In every case the underlying discipline is identical: fix the environment, govern it as a team, deposit with a persistent identifier, and state the level honestly. Only the last mile differs.

> Check 7: the archetype that changes nothing
>
> A simulation study, built entirely from its own generated data, is being prepared for deposit. Which part of this chapter's archiving machinery does it need, and which part, unlike the manuscript and package archetypes, does it not need to worry about?
> *The answer is on page 593.*

13.8. Worked example

To make the abstractions concrete, we follow the multi-site cohort laboratory of the Orientation through one quarter of collaboration and one archival deposit.

At the project's start the lead selected a base image, resolved it to a digest, and built the team environment once. Over the quarter, three contributors extended the package layer independently. The union model records the result: the project lockfile always holds the union of every contributor's additions. The following chunk reconstructs the running union over the quarter and tabulates it. It uses only base R together with `tibble` and `knitr`, and computes the cumulative union deterministically, so it regenerates identically on every render.

```
library(tibble)
library(knitr)
```


Draft, November 13, 2026. Content may change.

```
contributions <- tibble::tribble(
  ~contributor,          ~added,
  'Lead (initial base)', 'here, renv, rmarkdown',
  'Ana (survival)',      'survival, survminer, here',
  'Ben (tables)',        'gtsummary, gt',
  'Cai (spatial)',       'sf, tmap'
)

added_sets <- strsplit(contributions$added, ', ')
running_union <- Reduce(union, added_sets,
                        accumulate = TRUE)

contributions$union_n <- vapply(
  running_union, length, integer(1)
)

knitr::kable(
  contributions[, c('contributor', 'added', 'union_n')],
  col.names = c('Contributor', 'Packages added',
                'Lockfile union (n)')
)
```

Table 13.2.: The renv union model over one quarter of team collaboration. Each contributor installs packages independently; the project lockfile records the cumulative union, which the shared image is periodically rebuilt from. Because it is a union rather than a running total, a package a contributor installs that another already has does not enlarge it, so the union can grow by fewer than the number of packages added. The union size never shrinks, so every contributor who rebuilds inherits the full set.

Contributor	Packages added	Lockfile union (n)
Lead (initial base)	here, renv, rmarkdown	3
Ana (survival)	survival, survminer, here	5
Ben (tables)	gtsummary, gt	7
Cai (spatial)	sf, tmap	9

Draft, November 13, 2026. Content may change.

We see in the table the invariant that makes the model work. The union size is monotone, never shrinking as contributors are added, so a member who rebuilds the image at any point inherits everything the team needs up to that point. Because it is a union and not a running total, a package a contributor installs that the team already has does not enlarge it. Ana adds 3 packages, but one of them, `here`, was already in the lockfile from the initial base. The union therefore grows by only 2, from 3 to 5. Ana never installs `sf`, yet after the quarterly rebuild her environment carries it, because Cai did, and the lockfile records the union.

When the manuscript was accepted, the laboratory deposited the compendium. The cohort's patient-level extract came from the participating sites under a data-use agreement that forbids redistribution. The deposit therefore follows Section 13.5.2. It carries the code, the tests, the lockfile, the Dockerfile, the archived environment, and a synthetic extract with the structure of the real one. It also carries a README whose data-availability statement names the sites' data steward as the route to the real data. The extract itself is in none of these files. The session below is shown as a static illustration rather than executed. It checks that the image holds no study data, publishes it to a registry, and records its immutable digest. It then exports the environment as a single archival file and mints a DOI for the compendium. The `$` prompt is not part of the commands, and lines beginning `#>` are output.

```
# Rebuild the team image once, from the pinned
# Dockerfile and the union lockfile, then publish it.
$ make docker-build

# Before the image leaves the laboratory, confirm that
# no study data were copied into it; the expected
# result is an empty listing.
$ docker run --rm cohort-env:latest \
    find / -xdev -path '*raw_data*' -type f

$ docker tag cohort-env:latest \
    ghcr.io/mylab/cohort-env:2026-q3
$ docker push ghcr.io/mylab/cohort-env:2026-q3

# Record the immutable digest the team builds against.
```


Draft, November 13, 2026. Content may change.

```
# This, not the mutable tag, is the reproducible name.
$ docker inspect --format '{{index .RepoDigests 0}}' \
    ghcr.io/mylab/cohort-env:2026-q3
#> ghcr.io/mylab/cohort-env@sha256:4813816ad695...

# Export the environment as a single archival file for
# deposit, built from the digest-pinned image.
$ apptainer build cohort-env.sif \
    docker://ghcr.io/mylab/cohort-env@sha256:4813816ad695.
 ↪  ..

# Deposit the compendium (code, tests, lockfile,
# Dockerfile, synthetic extract, README; no patient
# data) plus the archived environment in Zenodo, which
# mints the persistent identifier the article's
# data-availability statement will cite. The command is
# schematic; real deposition uses the Zenodo REST API or
# the web upload interface, and the DOI is a placeholder.
$ zenodo-upload cohort-compendium-v1.0.zip cohort-env.sif
 ↪  # schematic
#> Deposited. DOI: 10.5281/zenodo.XXXXXXX
```

Three features of the session deserve note. First, the image is pushed by a dated tag but *recorded* by digest; the tag is convenient for humans, the digest is the immutable name a collaborator or a reviewer builds against. Second, the deposit carries both the compendium and the archived environment. Reproduction therefore has two independent routes, rebuild-from-recipe and run-the-archived-image, and does not depend on the registry surviving. Third, neither route includes the patient data. A reader of the deposit can run the pipeline on the synthetic extract and see that it executes. Regenerating the published estimates requires the real extract, obtained under the sites' agreement, and the article's level claim says so.

> Check 8: two routes, one reason
>
> The deposit above includes both a rebuildable Dockerfile and an archived SIF file of the same environment. Why both, rather than

whichever is more convenient?
The answer is on page 593.

13.9. The cohort, continued

The worked example above is a composite laboratory. We now apply a smaller, real question to this book's own running cohort compendium. If it had to be deposited today, what would already be true, and what would still need doing?

```
repo <- '../..'
in_repo <- function(path) file.exists(file.path(repo,
 ↪  path))

license_present <- in_repo('LICENSE')
description_present <- in_repo('DESCRIPTION')
lockfile_present <- in_repo('renv.lock')
dockerfile_present <- in_repo('Dockerfile')
dockerignore_present <- in_repo('.dockerignore')

# Ask git itself, so that directory patterns and `!`
# exceptions are applied exactly as `git add` applies them.
git_ignores <- function(path) {
  if (!nzchar(Sys.which('git'))) return(NA)
  status <- system2(
    'git',
    c('-C', repo, 'check-ignore', '--no-index', '--quiet',
     ↪  path),
    stdout = FALSE, stderr = FALSE
  )
  if (status %in% c(0L, 1L)) status == 0L else NA
}
cohort_ignored <-
 ↪  git_ignores('analysis/data/raw_data/cohort_raw.csv')
second_file_ignored <- git_ignores(
  'analysis/data/raw_data/registry_extract_2027.csv'
```


Draft, November 13, 2026. Content may change.

```
)

cff_lines <- readLines(file.path(repo, 'CITATION.cff'))
cff_license <- sub('^license:\\s*', '',
                   grep('^license:', cff_lines, value =
                    ↪  TRUE))

level_now <- if (lockfile_present && dockerfile_present) {
  'L2'
} else if (lockfile_present) {
  'L1'
} else {
  'L0'
}
c(LICENSE = license_present,
  DESCRIPTION = description_present,
  renv.lock = lockfile_present,
  Dockerfile = dockerfile_present,
  .dockerignore = dockerignore_present,
  cohort_csv_ignored = cohort_ignored,
  second_file_ignored = second_file_ignored)
#>             LICENSE         DESCRIPTION
 ↪  renv.lock          Dockerfile
#>                TRUE                TRUE
 ↪  TRUE               FALSE
#>       .dockerignore  cohort_csv_ignored
 ↪  second_file_ignored
#>               FALSE                TRUE
 ↪  TRUE
```

The first two checks concern licensing. A `LICENSE` file is present, and it covers three kinds of material. The prose is CC BY 4.0, the code printed in the book is MIT, and the repository code, including every script that builds the cohort, is GPL-3. That is the explicit license FAIR's reusability principle asks for, and the one Section 13.5.2's deposit example above was silent about. A `DESCRIPTION` file is present and records `License: GPL-3`, matching the repository-code section of the `LICENSE` file, so a package repository or an automated check reading `DESCRIPTION` sees the same

answer for the code that a human reading `LICENSE` would. `CITATION.cff` does not: its single `license` field reads CC-BY-4.0, the prose license, and a deposit service that takes its license from that field would record it for the code as well. Before a deposit, the license recorded in the deposit's own metadata is checked by hand against `LICENSE`.

The last two checks ask git itself, through `git check-ignore`, whether a path would be ignored. The cohort extract `analysis/data/raw_data/cohort_raw.csv` is ignored. The more informative result is the second. The file `registry_extract_2027.csv` does not exist and is named nowhere in `.gitignore`, yet git reports that a file of that name placed in the same directory would be ignored as well. This is what protects real data. A rule that names one file protects that file and nothing else, and the next extract a teammate saves beside it under a new name would be staged by the next `git add .` without a warning. The directory pattern `analysis/data/raw_data/*`, with an exception for the directory's `README.md`, protects whatever is put in the directory, including files no one has yet thought to name. A text search of `.gitignore` for the string `raw_data` cannot tell these two cases apart, which is why the chunk asks git rather than reading the file, and why it tests a filename that appears nowhere in it. The CSV itself is generated from a committed script rather than tracked. This is the practice that the version-control chapter (Chapter 4) and Section 13.5.2 both prescribe for data with identifier-shaped columns. The chunk cannot certify whether the data are actually sensitive, only which paths git would exclude. That judgment is the analyst's, informed by what the data are.

On the capture ladder, the compendium is at L1. What it still lacks among the artifacts checked here is a `Dockerfile` and a `.dockerignore`. The `.dockerignore` matters as soon as a `Dockerfile` exists, because `.gitignore` keeps files out of git and not out of a Docker build context; Exercise 8 below follows a file through that gap. What the compendium still lacks for an archival deposit proper is the part no file-system check can see. It lacks a minted DOI, an archived container image, and a written statement of the level this compendium has reached at the time of deposit.

Draft, November 13, 2026. Content may change.

Check 9: what the checks did not check

Suppose every check above returned `TRUE` for this compendium, a `Dockerfile` and a `.dockerignore` included. Would that mean the compendium is ready to deposit as L3? Why or why not?
The answer is on page 594.

13.10. Collaborating with an LLM

A language model is a capable assistant for the mechanical parts of sharing and archiving. It is an unreliable one for the parts that require judgment about this specific project.

! Restricted data

One rule comes first (habit 3). Never paste or upload protected health information, identifiable records, or data held under a data use agreement, nor outputs derived from them, into a model or a coding agent unless three conditions all hold. The institution has approved that endpoint for that class of data. The IRB protocol and the consent permit the processing. The data use agreement, or the provider's terms, permits it. If any of the three is unclear, the answer is no.
The third condition is the one most often missed. Institutional approval of a tool does not override the terms under which a data provider released the data. NIH, for example, treats sharing controlled-access data through prompts with public generative AI tools, which it glosses as third-party tools, as a breach of the data use certification. It treats a model trained on such data as a data derivative, to be shared only with other approved users and not kept after the project closes (National Institutes of Health, 2025a).
Describe the structure instead: the column names and types, or a small synthetic example generated from the data dictionary, never edited from real rows. A coding agent sees whatever files are in its working tree, so run it in a checkout or container where the real data is not mounted.

Draft, November 13, 2026. Content may change.

> An agent also acts, so confine it to a branch and keep merges, pushes, and deposits for yourself (Section 2.10.1).

For a deposit, the derived outputs that fall under the rule include row-level logs and small-count tables. The NIH notice cited above names public generative AI tools, which it glosses as third-party tools, and makes no exception for a tool an institution has licensed. A prompt about a deposit names the file names and the directory layout rather than any file's contents. We give three exchanges, each with what to watch for and what the analyst must verify.

Prompt. 'Write the shell commands to tag our image, push it to GHCR, record its digest, and export it as a SIF for archival deposit.'

Watch for. Check whether the commands push and record the *mutable tag* rather than resolving and recording the *digest*. Recording the tag silently discards the reproducibility the exercise exists to secure. Check also every registry path and flag against the registry's own documentation and the analyst's setup.

Verification. Run each command and confirm the recorded reference is a `@sha256:...` digest, not a `:tag`. Pull the image fresh on a second machine using the recorded digest and confirm it resolves. Do not deposit until the digest round-trips.

With a coding agent, these commands are executed rather than read. A push to a public registry and an export bound for deposit are actions the analyst takes after reading the commands, not ones an agent runs on its own authority (Section 2.10.1).

Prompt. 'Draft the data-availability and reproducibility statement for our manuscript, citing the deposited compendium.'

Watch for. The model does not know what level the compendium actually reached, whether the analyst verified the outputs or merely pinned them, or what the data-use agreement permits. Check for confident language ('fully reproducible', 'all results can be regenerated') that the compendium has not earned. Check also for a sentence such as 'all data are available in the deposited compendium' attached to a deposit that holds none.

Draft, November 13, 2026. Content may change.

Verification. Rewrite the level claim by hand against the honest-level standard. Say verified only if `zzc verify` or its equivalent was run and its baseline test passed; say pinned if the environment was pinned but not re-run. Name the DOI actually minted, not a placeholder. Check every sentence about the data against what the deposit holds and against the agreement, and name the access route for anything withheld.

A better prompt. 'Draft a data-availability statement and a code-availability statement for our manuscript. Inputs: the attached deposit file listing, the DOI `<minted DOI>`, the level reached (L1: a lockfile, no baseline re-run), and the agreement's redistribution clause, pasted below. Claim no level above L1 and no data the listing does not show. Give two short paragraphs, then list each sentence with the file or clause that supports it, so that I can check them one by one.'

An agent given access to the deposit service could go on to publish the record and mint the DOI. A published deposit cannot be recalled (Section 13.5.2), so publishing it, like the level claim, stays with the analyst (Section 2.10.1).

Prompt. 'Our 2024 analysis has an renv.lock but several pinned versions no longer install. How do we reproduce it?'

Watch for. Check whether the answer edits the lockfile to newer versions, which changes the analysis rather than reproducing it. If the answer recommends `rang`, check it against the caveat in 'Stewardship over time': the package was archived on CRAN on 2026-08-22, and it is built for the case in which no lockfile survives. Here one does, so the versions to install are already known.

Verification. Confirm that the proposed route installs the *recorded* versions, from the CRAN archive or from a dated repository snapshot, with the recorded run date used only to choose the snapshot. Diff the installed versions against the lockfile. Test the reconstructed environment against a known output from the original analysis before trusting it.

Draft, November 13, 2026. Content may change.

13.11. Exercises

1. Take an existing single-analyst compendium of your own and identify, for each package it uses, whether the decision to include it belongs to the environment layer or the package layer of a hypothetical team version. Justify the two or three borderline cases.

2. Explain, in a paragraph a collaborator could act on, why the committed Dockerfile rather than the pushed image is the reproducibility guarantee. Then describe one concrete situation in which relying on the pushed image would cause a reproduction to fail.

3. Your institution runs a self-hosted GitLab and forbids pushing to public forges. List the components of a compendium's reproducibility workflow that must change when moving from GitHub to GitLab, and the components that do not, and identify which single place the CI logic should live so that the change is minimal.

4. Deposit a small practice compendium in a research repository that mints a DOI (Zenodo offers a sandbox for exactly this). The compendium must hold no restricted data: use public or synthetic data only, such as this book's simulated cohort. Record the DOI. Then write a data-availability statement that says what the deposit contains and names the level the compendium actually reached, defending your level claim against the honest-level standard. Then rewrite the statement for the case in which the analysis data were a registry extract held under a data-use agreement. Say what would have been deposited instead and how a reader could obtain the real data.

5. Take a compendium with an `renv.lock` and, using `rang`, generate a Dockerfile that reconstructs its environment from the date it was created. Compare the reconstructed package set against the lockfile and report any differences, explaining why a date-based reconstruction may not match a lockfile exactly.

6. For one archetype other than the manuscript (a package, a blog, or a simulation), write the specific 'sharing with the world' step it requires beyond the archival deposit, and explain what validation gate, if any, that step adds.

Draft, November 13, 2026. Content may change.

7. Run the checks from 'The cohort, continued' yourself, from the repository root rather than from `analysis/report/` (so set `repo <- '.'`). For the ignore checks, also run `git check-ignore --no-index -v` in a terminal on `analysis/data/raw_data/cohort_raw.csv` and on a second file name of your own invention in the same directory, and note which line of `.gitignore` git reports as the match. Report the booleans you get. Then explain, in two or three sentences, why the result for your invented file name, not the one for `cohort_raw.csv`, tells you whether real patient-level data saved in that directory would stay out of git, and name one check in the chunk that would still have to change before the compendium could hold such data safely.

8. This exercise returns to the commit rule of Chapter 4 (Chapter 4): never commit restricted data, because `git` keeps every version forever even after a file is deleted. The same rule applies one layer up. Suppose a `.dockerignore` lists `*.csv` but a teammate's branch adds `analysis/data/raw_data/extra.tsv` the week before a deposit, and nobody updates the ignore patterns. Walk through what happens to that file at each of three points: the `docker build`, the `docker push`, and the Zenodo deposit of the exported SIF. Then say which single artifact, if inspected before the deposit, would have caught the leak.

9. Write a code-availability statement, meeting TRIPOD+AI item 18f's standard, for this book's running cohort compendium as it stands today. Name the repository, the license (recall 'The cohort, continued' above), the commit or tagged release the statement should point to, and anything withheld. Then write the parallel data-availability statement for a hypothetical version of the same analysis built on a registry extract held under a data-use agreement. Say in one sentence how the code statement would differ from the data statement in that case.

13.12. Solutions

Attempt an exercise before expanding its solution. Most of these are performed on a compendium of the reader's own, so their outputs vary; the boxes record the criterion to apply and the reasoning behind it.

Draft, November 13, 2026. Content may change.

Exercise 1

No single correct answer. The criterion to apply is **how often the package changes and how many contributors need it**.

The **environment layer** is the team image. Beyond the operating system, R, and the system libraries, it holds the packages every project needs and nobody changes: the tidyverse, `knitr` and `rmarkdown`, the plotting stack, anything requiring system libraries that are expensive to install. Baking these into the image means each contributor pays the install cost zero times rather than once each. The layer moves only when the team lead rebuilds.

The **package layer** holds what one contributor needs for one piece of work: a specialized survival package, a mapping library used in a single figure, anything adopted mid-project. These arrive through the lockfile, which every contributor extends.

The borderline cases are the substance of the exercise, and three kinds recur.

A package that *started* in the package layer and is now used by everyone belongs in the environment layer at the next rebuild. The signal is the union model: when a package appears in the lockfile because several contributors installed it independently, it has outgrown the fast layer.

A package with heavy system dependencies used by only one contributor is ambiguous. Installing it from the lockfile is slow and may fail without the system libraries; baking it in imposes the size on everyone. The usual resolution is to put the *system libraries* in the image and leave the R package in the lockfile.

A package pinned to an unusual version for one analysis belongs in that project's package layer, however widely the package itself is used. The team image serves every project in the lab and should carry the version most of them use. One project's lockfile can record the unusual version and restore it into the project library, which takes precedence over the image's copy, without imposing it on the other projects. Within a single project the union lockfile still records one version per package, so that project's contributors must agree on it.

Draft, November 13, 2026. Content may change.

Exercise 2

A paragraph a collaborator could act on:

> The Dockerfile is a recipe and the image is a cake. The recipe says how the environment was made, in text, in version control, reviewable line by line, and it will still say so in ten years. The image is an opaque binary that someone must keep hosting. It can be deleted, pruned, rate-limited, or rebuilt under the same tag, and nothing in it explains how it was produced. So commit the Dockerfile and treat the pushed image as a cache: convenient, worth having, and not the record.

A concrete failure from relying on the pushed image. The registry removes the image under a retention policy for inactive repositories. Alternatively, the organization account lapses, or the tag is overwritten by a later push. An analysis whose only environment record was that image is now unreconstructable. The failure appears at exactly the moment someone tries to reproduce the work, which is usually years after anyone was watching.

A second and more insidious failure merits mention because it produces no error. The image is rebuilt and pushed under the same tag with a newer base. A reader pulls the tag, gets a different environment from the one the analysis ran in, and obtains slightly different numbers with nothing to indicate anything moved. This is the argument for pinning by digest as well as committing the recipe.

A third arises if the collaborator works on Apple Silicon. The pushed image may have been built for one architecture only. The pull then succeeds on the author's machine and fails, or runs under emulation with different numerical behavior, on someone else's.

Exercise 3

What must change. The CI configuration file, from `.github/workflows/*.yml` to a single `.gitlab-ci.yml` at the root, in the dialect of stages and scripts rather than jobs and reusable actions. The remote URL and the authentication method. The registry, from GHCR

Draft, November 13, 2026. Content may change.

to the GitLab container registry, and therefore the image reference in any pull. The secrets, which must be re-created as GitLab CI variables. Anything relying on a GitHub-specific feature: pull-request comment bots, `GITHUB_TOKEN`, Actions from the marketplace.

What does not change. Everything that constitutes the reproducibility of the compendium. The Dockerfile, the `renv.lock`, the directory structure, the `DESCRIPTION`, the tests, the analysis code, the data dictionary, and the checks themselves are all forge-agnostic. (Restricted data stay off both forges; moving forges does not change where they live.) The reproducibility machinery is independent of the hosting platform. A compendium that must be substantially rewritten to move forges was coupled to its forge in ways it should not have been.

Where the CI logic should live. In a `Makefile`, or an equivalent single entry point, so that both forges' CI files reduce to a shim naming an image and calling `make check`, `make check-renv`, and the render target that matches the archetype (`make docker-render` for an `.Rmd` report; there is no plain `make render`). The migration then touches one thin file rather than the check logic, and the same commands run locally. That incidental benefit makes the arrangement worth adopting even for a project that will never move.

Exercise 4

The deposit and DOI will differ for each reader. The substance is the statement and its defense.

For a compendium built on simulated data, with a lockfile and a Dockerfile but no verification record:

> The complete research compendium, comprising the analysis code, the simulated data and the script that generates them, the package lockfile, and the container recipe, is archived at https://doi.org/10.5281/zenodo.XXXXXXX. The data are synthetic and describe no real person. The computational environment is pinned by a digest-referenced container image and an `renv.lock` lockfile, so the analysis can be re-executed in the environment in

Draft, November 13, 2026. Content may change.

> which it was developed (L2 on the capture ladder used in the accompanying methods). We have not independently verified that re-execution regenerates the reported outputs, so no L3 claim is made.

For the registry case, the deposit loses the data and the statement gains an access route:

> The analysis code, package lockfile, container recipe, and archived container image are archived at https://doi.org/10.5281/zenodo.XXXXXXX, with a synthetic dataset of the same structure on which the pipeline runs from end to end. The patient-level data were obtained from [registry] under a data-use agreement that does not permit redistribution; qualified researchers may apply to [data steward] for access. The synthetic data do not reproduce the reported estimates. The environment is pinned (L2); re-execution on the registry data has not been independently verified.

A common wrong answer keeps 'the complete research compendium' in the second statement. It is not complete, and saying so is part of the statement's honesty.

Defending the claim against the honest-level standard means checking three things and being willing to have the answer come out lower than one would like. Is there a lockfile? Then L1. Is there, in addition, an artifact pinning the runtime and system libraries? Then L2. Is there *evidence* (a CI log or a verification report) that a baseline test passed within a stated tolerance after a clean rebuild? Only then L3. A recorded hash of an output file is not that evidence. A matching hash shows that the file has not changed, but L3 is defined by agreement with a committed baseline within a tolerance, which a hash cannot express.

The last clause of the statement above is the one that makes it honest, and it is the clause most authors omit. Omitting it invites the reader to assume verification that nobody performed.

Draft, November 13, 2026. Content may change.

Exercise 5

```
lockfile <- renv::lockfile_read('renv.lock')
pkg_names <- names(lockfile$Packages)

graph <- rang::resolve(pkg_names, snapshot_date =
 ↪  '2026-06-29')
rang::dockerize(graph, output_dir = 'rang_build')
```

The comparison typically shows most packages matching and a minority differing, with the differing set growing as the gap widens between the snapshot date and the dates on which packages were actually installed.

Why a date-based reconstruction may not match a lockfile exactly. Four reasons, and it is worth separating them.

First and most important, a lockfile records what was *installed*, and a date reconstruction records what was *current on a date*. These differ whenever a package was installed on one date and the lockfile written on another. That is the normal case for a project of any duration.

Second, packages installed from sources other than CRAN (from GitHub at a particular commit, or from Bioconductor) have no CRAN-dated equivalent. The reconstruction therefore either misses them or substitutes something.

Third, archived packages present a gap. A package removed from CRAN is recoverable from the archive by version, but its dependency resolution at a past date may not be.

Fourth, the reconstruction resolves the transitive closure afresh, so a dependency that has since changed its own requirements can pull a different set than the lockfile recorded.

The conclusion the chapter draws: reconstruction by date is what one falls back on when no lockfile exists, and it is an approximation rather than a substitute. A project with a lockfile should restore it.

Exercise 6

Taking each in turn; the reader needs only one.

Draft, November 13, 2026. Content may change.

R package. The sharing step beyond the deposit is publication to a package repository, CRAN or Bioconductor, or at minimum a tagged release installable with `remotes::install_github()`. The validation gate this adds is substantial and externally imposed. CRAN runs `R CMD check` across multiple platforms and R versions, including development R, and rejects the package on failure. This is a stronger gate than any the author configured, precisely because the author did not configure it. It is the closest thing in the R ecosystem to the independent verification of Chapter 12.

Blog. The step is deployment to a public URL with a feed, and registration where readers will find it. The gate added is thin: a deploy check confirming the site builds and links resolve. Notably, no gate checks that the code in the posts still runs. That is why an old post is the most reliably broken artifact in this book's taxonomy.

Simulation study. The step is depositing the full results alongside the reduced pipeline, together with the seed and the replicate count, since the full run is too expensive to reproduce on demand. The gate added is the correspondence check: the reduced pipeline under the recorded seed must regenerate the leading replicates of the deposited full results. That holds only when a replicate's draws are fixed by its index: a seed or random-number stream per replicate (the per-task streams of Chapter 11), or one serial stream that the reduced pipeline consumes in exactly the same order. A full run made in parallel on streams assigned per worker does not qualify, because the stream a replicate drew from depends on the worker count. The Monte Carlo standard error of each reported summary belongs in the deposit as well, so that a reader can tell a changed result from Monte Carlo noise. The check is what lets a reader verify the machinery without paying for the study.

Exercise 7

The solution for this exercise is in the instructor guide, which is distributed separately to course instructors. Self-study readers can compare their booleans with the named vector the chunk in 'The cohort, continued' prints; it is computed at render time, so it, not any typed value, is the key. In outline, the argument runs as follows. A

Draft, November 13, 2026. Content may change.

rule that names `cohort_raw.csv` would report that file as ignored and say nothing about the next extract saved beside it, so only a file name that appears nowhere in `.gitignore` tests the protection real data depend on. When `git check-ignore -v` reports the directory pattern `analysis/data/raw_data/*` as the match for the invented name, any file saved there stays out of git. That is still not enough for real data: the `.dockerignore` check returns FALSE, and one that excludes the directory must exist before any `Dockerfile` copies the project into an image, because `.gitignore` does not govern a Docker build context.

Exercise 8

At `docker build`. The build context includes every file under the Dockerfile's directory not excluded by `.dockerignore`. Because `*.csv` does not match `extra.tsv`, the file is copied into the image by any `COPY . .` or equivalent instruction. It thereby becomes part of the image's file system layers.

At `docker push`. The image, `.tsv` included, is uploaded to the registry as-is. The registry has no knowledge of which files are data and which are code; it stores the layers it is given. The leak has now left the laboratory's own machines.

At the Zenodo deposit. `apptainer build` exports the pushed image, `.tsv` and all, into the SIF file, which is then deposited and assigned a DOI. The leak is now public and, per this chapter's own account of archival deposits, unrecallable.

What would have caught it. An inspection of the built image's file system before the push. This is the check this chapter's worked example performs with `docker run --rm ... find / -xdev -path '*raw_data*' -type f`, run against the actual data directory rather than assumed from the `.dockerignore` pattern. A pattern-matching ignore file is a plan; a listing of what the built image actually contains is a verification. Only the second one catches a file the plan did not anticipate. This is the Verification callout's own rule, 'how to see a claim actually holds,' applied to the narrower claim that no data entered the image.

Draft, November 13, 2026. Content may change.

Exercise 9

Code-availability statement. A statement meeting 18f's standard, as this book's repository stands:

> The analysis code for this compendium is available at [repository URL], at the commit tagged for this submission (not the default branch, which continues to change). The code is a `CITATION.cff`-bearing R research compendium; the code itself is released under GPL-3, the code printed in the accompanying book under MIT, and the prose and figures under CC BY 4.0. The deposit withholds nothing: the cohort analyzed is synthetic, generated by a committed, version-controlled script, and no restricted data were used.

Data-availability statement, registry-extract version.

> The patient-level data analyzed were obtained from [registry] under a data-use agreement that does not permit redistribution. Qualified researchers may apply to [data steward] for access under the same agreement terms. A synthetic dataset of the same structure, on which the pipeline runs end to end, is deposited alongside the code and does not reproduce the reported estimates.

How the two differ. The code statement answers 'can a reader obtain and run what produced these results,' and its answer here is simply yes, with a version and a license. The data statement answers a harder question, 'can a reader obtain what the code was run *on*,' and in the restricted case the honest answer is a controlled route rather than a download. The two statements can therefore diverge sharply even for the same manuscript. This is why a reporting guideline that names them as separate checklist items is doing real work.

Answers to the checks

Check 1. *from earlier chapters* (page 549)

The compendium in this warm-up, a Dockerfile with no lockfile, reaches only L0, locatable through git. The rungs are cumulative (Chapter 3), so without a lockfile it has not reached L1, and the Dockerfile cannot make it L2. Three steps raise it. A committed lockfile written by `renv::snapshot()` gives L1 (Chapter 6). Restoring that lockfile inside an image whose base is pinned by digest, with a dated package snapshot, gives L2 (Chapter 7); a tag such as `4.4.1` can be rebuilt to different bytes. L3 needs verification rather than more capture: a CI job that rebuilds the image and runs a baseline test within a stated tolerance (Chapter 12). A green `R CMD check` compares nothing with a recorded result.

Check 2. *two problems, not one* (page 552)

Neither event changes the level. Sharing the compendium with a new contributor, and depositing it with a DOI, are both social and temporal acts, not capture acts. The compendium was L2 before either happened and remains L2 afterward. What sharing adds is that the fixed environment survives contact with a second person without drifting. What archiving adds is that the fixed environment survives the passage of time and the disappearance of the original machine.

Check 3. *whose judgment, again* (page 554)

A checklist cannot decide what level to claim in the accompanying statement. It can confirm that a lockfile and a Dockerfile exist, which is mechanical. However, it cannot decide whether the analyst has actually re-run the pipeline from a clean rebuild and compared the outputs. Nor can it know what the consent and data-use agreement permit to accompany the deposit. Those are judgments about this specific project's history and its governing documents, and no tool has access to either.

Check 4. *what survives a registry outage* (page 558)

The pushed image is an opaque, impermanent binary that may be deleted, rate-limited, pruned, or built for the wrong architecture. It cannot be inspected to see how it was made. If Docker Hub removes it, an analysis that depended on it is lost. The source of truth should be the committed Dockerfile together with the lockfile it restores from. That is a transparent, diffable, durable recipe that reconstructs the same pinned environment, though not byte-identical image layers, from pinned inputs. The image is a convenience cache; the recipe is the guarantee. Pushing after every install

Draft, November 13, 2026. Content may change.

also wastes effort, since the image need only be rebuilt periodically from the union lockfile.

Check 5. *which route, and why not the obvious one* (page 564)

Deleting two columns is not de-identification in the regulatory sense. Dates, small geography, and rare combinations of remaining variables can still re-identify a person. De-identification needs either a qualified expert's documented determination that the risk is very small, or removal of all eighteen HIPAA identifiers with no actual knowledge that the rest could identify anyone. The analyst's own judgment is neither. Absent that determination in writing from the data steward or IRB, the extract stays out of an open deposit. The appropriate route is a synthetic stand-in with the same structure, deposited openly and labeled as synthetic. The real extract is made available, if the agreement allows it at all, only through a controlled-access repository or a direct request to the data steward.

Check 6. *recovering a decayed environment* (page 571)

The `rang` package (C. Chan & Schoch, 2023) addresses it by reconstructing a period-correct environment from a date. It resolves the package versions current at that date and generates a Dockerfile, even when the pinned versions are no longer installable directly. The single most useful recorded fact is the *date the analysis was run*, from which `rang` resolves the historical package set. It is a stewardship failure rather than a capture failure because the pins were captured correctly at the time (the compendium was L1). What lapsed was the preservation of a runnable environment over time, which an archived image or a recorded run date would have protected.

Check 7. *the archetype that changes nothing* (page 572)

It needs everything archetype-agnostic. It needs the two-layer model while the study was developed by a team, and the committed Dockerfile as the guarantee. It also needs a DOI-minting deposit carrying the code and the archived environment, and an honest statement of the level reached. It does not need Section 13.5.2's machinery at all. Data generated by the study's own code from stated parameters carries no participant, no consent, and no data-use agreement to honor.

Check 8. *two routes, one reason* (page 576)

Draft, November 13, 2026. Content may change.

Because each route can fail independently, and the deposit is meant to outlive both failures. The Dockerfile route fails if the base image's registry disappears or a pinned package is pulled from CRAN before `rang` or a successor tool can recover it. The archived-image route fails if the SIF or OCI file itself is lost or becomes unreadable by a future runtime. Depositing both means a reader needs only one of the two to still work. The archived image is often a gigabyte or more, so the second route is not free, but for a result that matters it is worth its size.

Check 9. *what the checks did not check* (page 579)

No. The checks are structural: a license file exists, a `DESCRIPTION` agrees with it, the capture files exist, and git would exclude the data directory. None of them is evidence that the analysis has been rebuilt from a clean environment and its outputs compared against a committed baseline within a stated tolerance. That is what an L3 claim requires. A compendium can pass every structural check above and still be no higher than L2, because files that capture the environment are not evidence that anyone rebuilt it. Structural readiness for deposit and the level honestly claimed in the deposit are two different questions, and this section answers only the first.

13.13. Cheat sheet

Command or rule	What it does or checks
`docker inspect --format '{{index .RepoDigests 0}}' <img>`	Reports the immutable digest to record, not the mutable tag
`docker run --rm <img> find / -xdev -path '*raw_data*'`	Confirms no restricted data were copied into the built image
`apptainer build out.sif docker://<img>@sha256:...`	Exports a digest-pinned image as a single archival file
`renv.lock` + `Dockerfile`, committed	The reproducibility guarantee; the pushed image is a cache, not the record

Draft, November 13, 2026. Content may change.

Command or rule	What it does or checks
Two-layer model	The team image (operating system, R, system libraries, lab-wide packages) is lead-governed and slow; the project lockfile is contributor-governed and fast, under the union model
Union lockfile	Records every contributor's packages cumulatively; never shrinks unless deliberately pruned; rebuild periodically, not on every install
`rang::resolve(pkgs, snapshot_date = ...)`	Reconstructs a period-correct environment from a date when no lockfile survives
Recorded run date	The single fact a retrospective recovery (`rang`) needs most
Restricted data (Section 13.5.2)	Code and environment, always; data only de-identified, synthetic, or controlled-access, per the agreement (habit 3)
DOI deposit	Lifts a compendium from fragile-locatable (a forge URL) to durably findable; changes no rung on the ladder
Honest claim at deposit	State the rung actually reached; the deposit outlives the memory of what was done (habit 7)
Trusted research environment (Section 13.5.3)	Bring code to the data; record the provider's environment identifier; L3 only inside
CONSORT 2025 item 4, SPIRIT 2025 item 6	Say where the statistical code (and de-identified data) can be accessed

Draft, November 13, 2026. Content may change.

13.14. Further reading

- Wilkinson and colleagues (Wilkinson et al., 2016) formulate the FAIR principles for research data; Barker and colleagues (Barker et al., 2022) adapt them for research software, whose reuse involves execution. Read both to understand what an archival deposit is trying to achieve, and why software reuse asks more than data reuse.
- Brinckman and colleagues (Brinckman et al., 2019) describe the Whole Tale platform, which bundles code, data, and environment into a single executable research object. It is the most ambitious realization of the compendium-as-runnable-object idea.
- The Turing Way (The Turing Way Community, 2022) is the community handbook for reproducible and collaborative research. For the social side of team collaboration, project governance, credit, and code review, in more depth than this chapter offers, it is the place to begin.
- Chan and colleagues (C. Chan & Schoch, 2023) present `rang`, the retrospective recovery tool for decayed R environments. Read it together with the pinning of Chapter 6 to see prospective capture and retrospective recovery as two halves of stewardship.
- Blischak and colleagues (Blischak et al., 2019) describe `workflowr`, which binds each rendered page of a versioned analysis website to its commit, the natural tool for the blog archetype's deployment.
- Stodden and colleagues (Stodden et al., 2018) provide the empirical case that a reproducibility policy is a weak predictor of an actually reproducible artifact, the evidence behind this book's insistence on honest disclosure over asserted reproducibility.
- The NIH Data Management and Sharing Policy (National Institutes of Health, 2020) and NIH's principles for protecting participant privacy (National Institutes of Health, 2026c) set out what a funded project must plan for when its data come from human participants; read them before writing a data-availability statement for such a project.

13.15. In conclusion

The book's argument comes down to four points. Reproducibility, we have maintained throughout, is not a badge a project either holds or lacks. It

Draft, November 13, 2026. Content may change.

is a level a project reaches. The discipline is to reach as high as the work warrants and to say plainly how high that was.

First, reproducibility comes in degrees, and the ladder names them: locatable (L0), pinned packages (L1), pinned environment (L2), and verified (L3). Every chapter of this book has been an ascent of that ladder. We began by finding the source and pinning the packages with a lockfile. We then fixed the environment with a container or a content-addressed alternative, and finally regenerated the outputs and confirmed they match. The ladder is the common scale on which any analysis, built with any toolchain, can be placed and compared.

Second, the rungs are reached by two different kinds of work, and conflating them is the source of much confusion. Capture fixes the inputs and sets the level from L0 to L2. Pinning the packages and the environment is capture, which carries a compendium as far as L2. The last rung, L3, is reached not by capture but by an act of verification: a clean rebuild after which a baseline test passes within a stated tolerance. Running `zzc verify` once, with such a baseline test in place, is therefore what lifts a pinned L2 compendium to a verified L3 one. Automated re-checking is distinct from that act. Continuous integration reruns the verification on every change and certifies that the L3 state still holds, but it adds no rung of its own. Removing it does not lower the level; it only stops the automated confirmation. Keeping these registers distinct is what lets a project report, honestly and mechanically, exactly where it stands.

Third, the unit of reproducible work is the research compendium, and it is not only a manuscript. The same machinery, the same pillars, and the same ladder serve a data analysis, a simulation study, an R package, a blog, and a scholarly manuscript alike. The archetypes differ only in their directory layout, in whether a report is rendered, and in the last mile of publication: a package submission, a deployed website, or a deposit beside an article.

Fourth, and above all, the goal is reproducibility matched honestly to the level actually reached. The tools this book has taught, and the framework it has used as its running example, are strong but partial. They firmly secure some determinants and leave others to convention, and an honest practitioner discloses the difference. A verified L3 compendium and a locatable L0 script are both legitimate outputs of a scientific process; what is not legitimate is to present the second as the first. The reader who

Draft, November 13, 2026. Content may change.

finishes this book with the judgment to name the level a given analysis has reached, across whatever archetype the work takes, and the honesty to claim no more, has learned the one thing the book most wished to teach.

The introduction named seven habits that recur across every chapter since. They are worth gathering here, once more, as the book's last word rather than its first.

Seven habits, and a motto

Claim only the level you have verified.

1. Commit before you compute.
2. Pin what you ran, not what is current.
3. Never commit, deposit, or paste restricted data.
4. Make every gate able to fail, and watch it fail once.
5. Compare against a recorded baseline, with a stated tolerance.
6. Rebuild from clean, on a machine that is not yours.
7. Write the honest level down.

Draft, November 13, 2026. Content may change.

References

21 CFR part 11: Electronic records; electronic signatures. (n.d.). Retrieved November 8, 2026, from https://www.govinfo.gov/app/details/CFR-2025-title21-vol1/CFR-2025-title21-vol1-part11

45 CFR 164.514: Other requirements relating to uses and disclosures of protected health information. (n.d.). Retrieved October 11, 2026, from https://www.law.cornell.edu/cfr/text/45/164.514

Allaire, J. J., Teague, C., Scheidegger, C., Xie, Y., & Dervieux, C. (2024). *Quarto: An open-source scientific and technical publishing system*. https://quarto.org/

American Journal of Epidemiology. (n.d.). *Author guidelines*. Oxford Academic. Retrieved November 8, 2026, from https://academic.oup.com/aje/pages/General_Instructions

Anaconda Inc. (2023). *Anaconda software distribution*. https://docs.anaconda.com/

Apptainer Project. (2024). *Bind paths and mounts*. https://apptainer.org/docs/user/1.3/bind_paths_and_mounts.html

Association for Computing Machinery. (2020). *Artifact review and badging — version 1.1*. ACM publications policy. https://www.acm.org/publications/policies/artifact-review-and-badging-current

Baker, M. (2016). 1,500 scientists lift the lid on reproducibility. *Nature*, *533*(7604), 452–454. https://doi.org/10.1038/533452a

Barker, M., Chue Hong, N. P., Katz, D. S., et al. (2022). Introducing the FAIR principles for research software. *Scientific Data*, *9*, 622. https://doi.org/10.1038/s41597-022-01710-x

Begley, C. G., & Ellis, L. M. (2012). Raise standards for preclinical cancer research. *Nature*, *483*(7391), 531–533. https://doi.org/10.1038/483531a

Biostatistics. (n.d.). *Author guidelines*. Oxford Academic. Retrieved November 8, 2026, from https://academic.oup.com/biostatistics/pages/General_Instructions

Blischak, J. D., Carbonetto, P., & Stephens, M. (2019). Creating and

sharing reproducible research code the workflowr way. *F1000Research*, *8*, 1749. https://doi.org/10.12688/f1000research.20843.1

Boettiger, C. (2015). An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, *49*(1), 71–79. https://doi.org/10.1145/2723872.2723882

Boettiger, C., & Eddelbuettel, D. (2017). An introduction to Rocker: Docker containers for R. *The R Journal*, *9*(2), 527–536. https://doi.org/10.32614/RJ-2017-065

Brinckman, A., Chard, K., Gaffney, N., et al. (2019). Computing environments for reproducibility: Capturing the Whole Tale. *Future Generation Computer Systems*, *94*, 854–867. https://doi.org/10.1016/j.future.2017.12.029

Bryan, J. (2018). Excuse me, do you have a moment to talk about version control? *The American Statistician*, *72*(1), 20–27. https://doi.org/10.1080/00031305.2017.1399928

Bryan, J., & Hester, J. (2024). *What they forgot to teach you about R: Practice safe paths*. https://rstats.wtf/practice-safe-paths.html

Bryan, J., the STAT 545 TAs, & Hester, J. (n.d.). *Happy git and GitHub for the useR*. Retrieved November 8, 2026, from https://happygitwithr.com

Buckheit, J. B., & Donoho, D. L. (1995). WaveLab and reproducible research. In *Wavelets and statistics* (Vol. 103, pp. 55–81). Springer. https://doi.org/10.1007/978-1-4612-2544-7_5

Chan, A.-W., Boutron, I., Hopewell, S., Moher, D., Schulz, K. F., Collins, G. S., et al. (2025). SPIRIT 2025 statement: Updated guideline for protocols of randomised trials. *BMJ*, *389*, e081477. https://doi.org/10.1136/bmj-2024-081477

Chan, C., & Schoch, D. (2023). Rang: Reconstructing reproducible R computational environments. *PLOS ONE*, *18*(6), e0286761. https://doi.org/10.1371/journal.pone.0286761

Claerbout, J. F., & Karrenbach, M. (1992). Electronic documents give reproducible research a new meaning. *SEG Technical Program Expanded Abstracts 1992*, 601–604. https://doi.org/10.1190/1.1822162

Collins, G. S., Moons, K. G. M., Dhiman, P., Riley, R. D., Beam, A. L., Van Calster, B., Ghassemi, M., Liu, X., Reitsma, J. B., Smeden, M. van, Boulesteix, A.-L., Camaradou, J. C., Celi, L. A., Denaxas, S., Denniston, A. K., Glocker, B., Golub, R. M., Harvey, H., Heinze, G., … Moher, D. (2024). TRIPOD+AI statement: Updated guidance for reporting clinical prediction models that use regression or machine learning methods. *BMJ*,

Draft, November 13, 2026. Content may change.

385, e078378. https://doi.org/10.1136/bmj-2023-078378

Development Containers Project. (2022). *Development containers specification*. https://containers.dev/

Di Tommaso, P., Chatzou, M., Floden, E. W., Prieto Barja, P., Palumbo, E., & Notredame, C. (2017). Nextflow enables reproducible computational workflows. *Nature Biotechnology*, *35*(4), 316–319. https://doi.org/10.1038/nbt.3820

Docker Inc. (2024a). *Docker desktop WSL 2 backend on Windows*. https://docs.docker.com/desktop/features/wsl/

Docker Inc. (2024b). *Multi-platform builds*. https://docs.docker.com/build/building/multi-platform/

Dolstra, E., Jonge, M. de, & Visser, E. (2004). Nix: A safe and policy-free system for software deployment. *Proceedings of the 18th USENIX Conference on System Administration (LISA '04)*, 79–92.

European Commission. (n.d.). *European health data space regulation (EHDS)*. Retrieved November 8, 2026, from https://health.ec.europa.eu/ehealth-digital-health-and-care/european-health-data-space-regulation-ehds_en

European Medicines Agency, GCP Inspectors Working Group. (2023). *Guideline on computerised systems and electronic data in clinical trials*. https://www.ema.europa.eu/en/documents/regulatory-procedural-guideline/guideline-computerised-systems-electronic-data-clinical-trials_en.pdf

European Parliament and Council of the European Union. (2025). *Regulation (EU) 2025/327 of 11 february 2025 on the european health data space and amending directive 2011/24/EU and regulation (EU) 2024/2847*. Official Journal of the European Union, L series. https://eur-lex.europa.eu/eli/reg/2025/327/oj/eng

Gentleman, R., & Temple Lang, D. (2007). Statistical analyses and reproducible research. *Journal of Computational and Graphical Statistics*, *16*(1), 1–23. https://doi.org/10.1198/106186007X178663

GitHub. (n.d.-a). *About large files on GitHub*. GitHub Docs. Retrieved November 8, 2026, from https://docs.github.com/en/repositories/working-with-files/managing-large-files/about-large-files-on-github

GitHub. (n.d.-b). *Configuring the retention period for GitHub actions artifacts and logs in your organization*. GitHub Docs. Retrieved November 8, 2026, from https://docs.github.com/en/organizations/managing-organization-settings/configuring-the-retention-period-for-github-

actions-artifacts-and-logs-in-your-organization

GitHub. (n.d.-c). *Removing sensitive data from a repository*. GitHub Docs. Retrieved October 11, 2026, from https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/removing-sensitive-data-from-a-repository

GitHub. (n.d.-d). *Security hardening for GitHub actions*. GitHub Docs. Retrieved October 12, 2026, from https://docs.github.com/en/actions/security-for-github-actions/security-guides/security-hardening-for-github-actions

GitHub Docs. (2024). *About CITATION files*. https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-citation-files

Goodman, S. N., Fanelli, D., & Ioannidis, J. P. A. (2016). What does research reproducibility mean? *Science Translational Medicine*, *8*(341), 341ps12. https://doi.org/10.1126/scitranslmed.aaf5027

Hopewell, S., Chan, A.-W., Collins, G. S., et al. (2025). CONSORT 2025 statement: Updated guideline for reporting randomised trials. *PLOS Medicine*, *22*(4), e1004587. https://doi.org/10.1371/journal.pmed.1004587

International Committee of Medical Journal Editors. (2026). *Use of AI by authors*. https://www.icmje.org/recommendations/browse/artificial-intelligence/ai-use-by-authors.html

Ioannidis, J. P. A. (2005). Why most published research findings are false. *PLOS Medicine*, *2*(8), e124. https://doi.org/10.1371/journal.pmed.0020124

Journal of the American Statistical Association. (n.d.). *JASA reproducibility guide*. Retrieved November 8, 2026, from https://jasa-acs.github.io/repro-guide/

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, *27*(2), 97–111. https://doi.org/10.1093/comjnl/27.2.97

Köster, J., & Rahmann, S. (2012). Snakemake — a scalable bioinformatics workflow engine. *Bioinformatics*, *28*(19), 2520–2522. https://doi.org/10.1093/bioinformatics/bts480

Kurtzer, G. M., Sochat, V., & Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. *PLOS ONE*, *12*(5), e0177459. https://doi.org/10.1371/journal.pone.0177459

Landau, W. M. (2018). The drake R package: A pipeline toolkit for reproducibility and high-performance computing. *Journal of Open*

Draft, November 13, 2026. Content may change.

Source Software, *3*(21), 550. https://doi.org/10.21105/joss.00550

Landau, W. M. (2021). The targets R package: A dynamic make-like function-oriented pipeline toolkit for reproducibility and high-performance computing. *Journal of Open Source Software*, *6*(57), 2959. https://doi.org/10.21105/joss.02959

Leek, J. T., & Peng, R. D. (2015). Reproducible research can still be wrong: Adopting a prevention approach. *Proceedings of the National Academy of Sciences*, *112*(6), 1645–1646. https://doi.org/10.1073/pnas.1421412111

Loo, M. van der. (2020). *Tinytest: Lightweight and feature complete unit testing framework*. https://CRAN.R-project.org/package=tinytest

Marwick, B., Boettiger, C., & Mullen, L. (2018). Packaging data analytical work reproducibly using R (and friends). *The American Statistician*, *72*(1), 80–88. https://doi.org/10.1080/00031305.2017.1375986

McBain, M. (2023). *Capsule: Reproducible project libraries via a lazy renv*. https://github.com/MilesMcBain/capsule

Microsoft. (2024a). *Install WSL*. https://learn.microsoft.com/en-us/windows/wsl/install

Microsoft. (2024b). *Working across file systems*. https://learn.microsoft.com/en-us/windows/wsl/filesystems

Morris, T. P., White, I. R., & Crowther, M. J. (2019). Using simulation studies to evaluate statistical methods. *Statistics in Medicine*, *38*(11), 2074–2102. https://doi.org/10.1002/sim.8086

National Academies of Sciences, Engineering, and Medicine. (2019). *Reproducibility and replicability in science*. The National Academies Press. https://doi.org/10.17226/25303

National Center for Biotechnology Information. (n.d.). *dbGaP: Database of genotypes and phenotypes*. Retrieved October 11, 2026, from https://dbgap.ncbi.nlm.nih.gov/home

National Institutes of Health. (n.d.). *Writing a data management and sharing plan*. Retrieved November 8, 2026, from https://grants.nih.gov/policy-and-compliance/policy-topics/sharing-policies/dms/writing-dms-plan

National Institutes of Health. (2020). *Final NIH policy for data management and sharing*. https://grants.nih.gov/grants/guide/notice-files/NOT-OD-21-013.html

National Institutes of Health. (2023). *The use of generative artificial intelligence technologies is prohibited for the NIH peer review process*. https://grants.nih.gov/grants/guide/notice-files/NOT-OD-23-149.html

National Institutes of Health. (2025a). *Protecting human genomic data when developing generative artificial intelligence tools and applications*. https://grants.nih.gov/grants/guide/notice-files/NOT-OD-25-081.html

National Institutes of Health. (2025b). *Supporting fairness and originality in NIH research applications*. https://grants.nih.gov/grants/guide/notice-files/NOT-OD-25-132.html

National Institutes of Health. (2026a). *Accelerated implementation of data management and sharing plan changes for NIH career development (K) awards*. https://grants.nih.gov/grants/guide/notice-files/NOT-OD-26-120.html

National Institutes of Health. (2026b). *Implementation update: NIH data management and sharing plan requirements*. https://grants.nih.gov/grants/guide/notice-files/NOT-OD-26-100.html

National Institutes of Health. (2026c). *Principles and best practices for protecting participant privacy*. https://grants.nih.gov/policy-and-compliance/policy-topics/sharing-policies/dms/privacy/best-practices

National Institutes of Health, All of Us Research Program. (n.d.-a). *Data and statistics dissemination policy*. All of Us Research Hub. Retrieved November 8, 2026, from https://www.researchallofus.org/?p=4698

National Institutes of Health, All of Us Research Program. (n.d.-b). *Researcher workbench*. All of Us Research Hub. Retrieved November 8, 2026, from https://www.researchallofus.org/data-tools/workbench/

NHS England RAP Community of Practice. (2024). *Levels of RAP*. https://nhsdigital.github.io/rap-community-of-practice/introduction_to_RAP/levels_of_RAP/

Nicholls, A., Bargo, P. R., & Sims, J. (2020). *A risk-based approach for assessing R package accuracy within a validated infrastructure*. R Validation Hub. https://www.pharmar.org/white-paper/

Nüst, D., Boettiger, C., & Marwick, B. (2018). How to read a research compendium. *arXiv Preprint arXiv:1806.09525*. https://arxiv.org/abs/1806.09525

Nüst, D., Eddelbuettel, D., Bennett, D., Cannoodt, R., Clark, D., Daroczi, G., Edmondson, M., Fay, C., Hughes, E., et al. (2020). The Rockerverse: Packages and applications for containerisation with R. *The R Journal*, *12*(1), 437–461. https://doi.org/10.32614/RJ-2020-007

Nüst, D., & Eglen, S. J. (2021). CODECHECK: An open science initiative for the independent execution of computations underlying research articles during peer review to improve reproducibility. *F1000Research*,

Draft, November 13, 2026. Content may change.

10, 253. https://doi.org/10.12688/f1000research.51738.2

Nüst, D., Sochat, V., Marwick, B., Eglen, S. J., Head, T., Hirst, T., & Evans, B. D. (2020). Ten simple rules for writing Dockerfiles for reproducible data science. *PLOS Computational Biology*, *16*(11), e1008316. https://doi.org/10.1371/journal.pcbi.1008316

Open Science Collaboration. (2015). Estimating the reproducibility of psychological science. *Science*, *349*(6251), aac4716. https://doi.org/10.1126/science.aac4716

OpenChain Project. (2025). *From heritage to hash: SWHID becomes a global ISO/IEC standard*. https://openchainproject.org/news/2025/04/29/from-heritage-to-hash-swhid-becomes-a-global-iso-iec-standard

OWASP GenAI Security Project. (2025a). *LLM01:2025 prompt injection*. https://genai.owasp.org/llmrisk/llm01-prompt-injection/

OWASP GenAI Security Project. (2025b). *LLM06:2025 excessive agency*. https://genai.owasp.org/llmrisk/llm062025-excessive-agency/

Patil, P., Peng, R. D., & Leek, J. T. (2016). A statistical definition for reproducibility and replicability. *bioRxiv*, 066803. https://doi.org/10.1101/066803

Peng, R. D. (2011). Reproducible research in computational science. *Science*, *334*(6060), 1226–1227. https://doi.org/10.1126/science.1213847

Peng, R. D. (2009). Reproducible research and Biostatistics. *Biostatistics*, *10*(3), 405–408. https://doi.org/10.1093/biostatistics/kxp014

Peng, R. D., Dominici, F., & Zeger, S. L. (2006). Reproducible epidemiologic research. *American Journal of Epidemiology*, *163*(9), 783–789. https://doi.org/10.1093/aje/kwj093

Posit Software, PBC. (n.d.). *Managing execution*. Quarto documentation. Retrieved November 8, 2026, from https://quarto.org/docs/projects/code-execution.html

prefix.dev. (2024). *Pixi: A fast software package manager*. https://pixi.sh/

Project Jupyter, Bussonnier, M., Forde, J., Freeman, J., Granger, B., Head, T., Holdgraf, C., Kelley, K., Nalvarte, G., Osheroff, A., Pacer, M., Panda, Y., Perez, F., Ragan-Kelley, B., & Willing, C. (2018). Binder 2.0 — reproducible, interactive, sharable environments for science at scale. *Proceedings of the 17th Python in Science Conference (SciPy 2018)*, 113–120. https://doi.org/10.25080/Majora-4af1f417-011

R Consortium. (2026). *Beyond feasibility: Learning from FDA's response to WebAssembly and container-based submissions*. https://r-consortium.org/posts/beyond-feasibility-learning-from-

fdas-response-to-webassembly-and-container-based-submissions/

R Core Team. (2019). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. https://www.R-project.org/

R Core Team. (2020). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. https://www.R-project.org/

Ram, K. (2013). Git can facilitate greater reproducibility and increased transparency in science. *Source Code for Biology and Medicine*, *8*(1), 7. https://doi.org/10.1186/1751-0473-8-7

Rodrigues, B., & Baumann, P. (2024). *Rix: Reproducible data science environments with Nix*. https://docs.ropensci.org/rix/

Saju, L. et al. (2025). Computational reproducibility of R code supplements on OSF. *arXiv Preprint arXiv:2505.21590*. https://arxiv.org/abs/2505.21590

Sandve, G. K., Nekrutenko, A., Taylor, J., & Hovig, E. (2013). Ten simple rules for reproducible computational research. *PLOS Computational Biology*, *9*(10), e1003285. https://doi.org/10.1371/journal.pcbi.1003285

SATRE Community. (2026). *Standard architecture for trusted research environments (SATRE), release 2.0.1*. https://satre-specification.readthedocs.io/en/stable/

Simonsohn, U. (2026). *Groundhog: Version-control for CRAN, GitHub, and GitLab packages*. https://groundhogr.com/

Software Carpentry. (2024). *The Unix shell*. The Carpentries. https://swcarpentry.github.io/shell-novice/

Software Heritage. (2024). *swh-model: Persistent identifiers (SWHID)*. https://raw.githubusercontent.com/SoftwareHeritage/swh-model/master/README.rst

Stodden, V., Seiler, J., & Ma, Z. (2018). An empirical analysis of journal policy effectiveness for computational reproducibility. *Proceedings of the National Academy of Sciences*, *115*(11), 2584–2589. https://doi.org/10.1073/pnas.1708290115

team, rgtlab curriculum. (2026). *Zzcollab: A flexible extension of rrtools for reproducible, collaborative research compendia in R* [Unpublished manuscript].

The Turing Way Community. (2022). *The Turing Way: A handbook for reproducible, ethical and collaborative research*. Zenodo. https://doi.org/10.5281/zenodo.3233853

Draft, November 13, 2026. Content may change.

Trisovic, A., Lau, M. K., Pasquier, T., & Crosas, M. (2022). A large-scale study on research code quality and execution. *Scientific Data*, *9*(1), 60. https://doi.org/10.1038/s41597-022-01143-6

UK Data Service. (n.d.). *What is the five safes framework?* Retrieved November 8, 2026, from https://ukdataservice.ac.uk/help/secure-lab/what-is-the-five-safes-framework/

U.S. Food and Drug Administration. (2002). *General principles of software validation; final guidance for industry and FDA staff*. https://www.fda.gov/regulatory-information/search-fda-guidance-documents/general-principles-software-validation

U.S. Food and Drug Administration. (2003). *Guidance for industry. Part 11, electronic records; electronic signatures — scope and application*. https://www.fda.gov/regulatory-information/search-fda-guidance-documents/part-11-electronic-records-electronic-signatures-scope-and-application

U.S. Food and Drug Administration, Center for Drug Evaluation and Research, Office of Biostatistics. (2026). *Statistical review and evaluation: R pilot 4 submission (BLA 111111)*. Review of the R Consortium R Submissions Working Group's Pilot 4. https://github.com/RConsortium/submissions-wg/blob/main/_Documents/Summary_R_Pilot4_Submission.pdf

Ushey, K., & Wickham, H. (2026). *Renv: Project environments*. https://CRAN.R-project.org/package=renv

Vandewalle, P., Kovacevic, J., & Vetterli, M. (2009). Reproducible research in signal processing. *IEEE Signal Processing Magazine*, *26*(3), 37–47. https://doi.org/10.1109/MSP.2009.932122

Vilhuber, L., Connolly, M., Koren, M., Llull, J., & Morrow, P. (2022). *A template README for social science replication packages* (Version v1.1). Zenodo. https://doi.org/10.5281/zenodo.7293838

Wickham, H. (2015). *R packages: Organize, test, document, and share your code*. O'Reilly Media.

Wickham, H., & Bryan, J. (2023). *R packages: Organize, test, document, and share your code* (2nd ed.). O'Reilly Media. https://r-pkgs.org

Wilkinson, M. D., Dumontier, M., Aalbersberg, Ij. J., Appleton, G., et al. (2016). The FAIR guiding principles for scientific data management and stewardship. *Scientific Data*, *3*, 160018. https://doi.org/10.1038/sdata.2016.18

Wilson, G., Aruliah, D. A., Brown, C. T., Chue Hong, N. P., Davis, M.,

Guy, R. T., Haddock, S. H. D., Huff, K. D., Mitchell, I. M., Plumbley, M. D., Waugh, B., White, E. P., & Wilson, P. (2014). Best practices for scientific computing. *PLOS Biology*, *12*(1), e1001745. https://doi.org/10.1371/journal.pbio.1001745

Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., & Teal, T. K. (2017). Good enough practices in scientific computing. *PLOS Computational Biology*, *13*(6), e1005510. https://doi.org/10.1371/journal.pcbi.1005510

Xie, Y. (2011). *Cache: Examples for the cache feature*. knitr documentation. https://yihui.org/knitr/demo/cache/

Xie, Y. (2015). *Dynamic documents with R and knitr* (2nd ed.). Chapman; Hall/CRC.

Xie, Y., Allaire, J. J., & Grolemund, G. (2018). *R markdown: The definitive guide*. Chapman; Hall/CRC. https://bookdown.org/yihui/rmarkdown/

Draft, November 13, 2026. Content may change.

Shell Primer

Every chapter in this book runs commands at a shell prompt: `docker build`, `git commit`, `Rscript`, `quarto render`. The book assumes a reader can already type these and read what comes back. This appendix supplies that prerequisite for a reader who has not yet acquired it. It covers enough of the Unix shell to follow every command shown elsewhere in the book, in the order those commands are needed, with nothing beyond that scope. The treatment below follows the scope and sequencing of a widely used open lesson covering the same ground (Software Carpentry, 2024).

Where you are: `pwd`, `ls`, `cd`

A shell session has a current working directory, the folder commands act on by default. `pwd` ('print working directory') reports it as an absolute path, one that starts from the filesystem root and names every folder down to the current one:

```
pwd
#> /Users/jane/project
```

`ls` lists the contents of the working directory, or of a path given as an argument:

```
ls
#> analysis  R  DESCRIPTION  renv.lock
ls analysis
#> data  report
```


Draft, November 13, 2026. Content may change.

Plain `ls` hides entries whose name starts with a dot, such as `.gitignore` or the `.here` marker Chapter 5 relies on. `ls -a` ('all') shows them, along with the two special entries every directory has, `.` for itself and `..` for its parent:

```
ls -a
#> .  ..  .gitignore  .here  analysis  R  DESCRIPTION
 ↪  renv.lock
```

`cd` ('change directory') moves the working directory. `cd analysis/report` moves into a subfolder named relative to where the shell already is, and `cd /Users/jane/project` moves to an absolute path regardless of the starting point. `cd ..` moves up one level, to the parent of the current directory, and `cd` with no argument returns to the home directory. `cd ../..` chains two steps up, and `cd ../report` moves up one level and back down into a sibling folder, without ever returning to the project root by name.

The distinction between a relative path (`analysis/report`, `../data`) and an absolute path (`/Users/jane/project`) matters throughout this book. A relative path resolves differently depending on the working directory the shell happens to be in when the command runs. This is exactly the fragility Chapter 5 examines in its section on project-oriented paths. A script that is correct when launched from the project root can be wrong when launched from `analysis/report/`, because a relative path has no fixed meaning on its own. `here::here()` fixes this inside R by computing an absolute path from a marker file. However, the underlying hazard (that a relative path depends on the starting directory) is a shell concept first and an R concept second.

File operations: `cp`, `mv`, `rm`

`cp source destination` copies a file, leaving the original in place. `mv source destination` moves or renames a file; the file at `source` no longer exists afterward, only the one at `destination` does. Both accept a directory as the destination to copy or move a file into it, keeping its name:

Draft, November 13, 2026. Content may change.

```
cp cohort_raw.csv backup/cohort_raw.csv
mv draft.qmd final.qmd
```

A single asterisk, *, is a wildcard the shell expands to every matching file name before the command ever runs it:

```
ls *.csv
#> cohort_raw.csv  cohort_clean.csv
```

`rm *.csv` therefore does not pass the literal text `*.csv` to `rm`; the shell expands it first, into the list of matching file names, and `rm` deletes each one named in that list. Because the expansion happens silently before the command runs, `*.csv` in the wrong directory deletes a different, unintended set of files, not nothing.

`rm file` deletes a file, and unlike `git`, the shell has no undo. A commit can be reverted, a branch can be recovered from the reflog, but a file removed with `rm` is simply gone. It does not pass through a trash folder, and no earlier chapter's recovery techniques apply to it. Running `rm *.csv` in the wrong directory, or `rm -rf` on the wrong path, destroys exactly what it is told to destroy, correctly and irreversibly. Treat `rm` as one-way, and prefer moving a file to an explicitly named `trash/` folder over deleting it outright when the cost of being wrong is high.

Redirection and pipes: >, >>, |

Every command in a shell produces output on a stream called standard output, which prints to the terminal by default. Redirection sends that stream somewhere else instead. `>` writes standard output to a file, replacing the file's entire contents if it already exists:

```
echo 'first line' > out.txt
cat out.txt
#> first line
echo 'second line' > out.txt
```


Draft, November 13, 2026. Content may change.

```
cat out.txt
#> second line
```

The second `>` above did not add a line; it overwrote the file, and `first line` is gone. `>>` appends instead of overwriting, adding its output after whatever is already in the file:

```
echo 'third line' >> out.txt
cat out.txt
#> second line
#> third line
```

This single character, `>` versus `>>`, is the difference between replacing a log file and growing one. It is worth checking deliberately, because the failure mode of using `>` by mistake is silent: the file exists, it has content, and nothing in the command's own output signals that the previous content was discarded.

A pipe, `|`, sends one command's standard output directly into the next command's standard input, without an intermediate file:

```
printf 'apple\nbanana\ncherry\n' | grep an | wc -l
#> 1
```

Here `printf` produces three lines, `grep an` keeps only the lines containing `an` (just `banana`), and `wc -l` counts the lines that survive, reporting 1. Chains like this appear throughout the book's CI configuration and `Makefile` targets, where a build step's output is filtered or counted without ever being written to disk.

A command has a second stream, standard error, for diagnostics and failures, kept separate from standard output so that redirecting one does not silently swallow the other. `2>` redirects standard error alone:

```
cat does_not_exist.txt 2> err.txt
cat err.txt
#> cat: does_not_exist.txt: No such file or directory
```


Draft, November 13, 2026. Content may change.

Without the `2>`, that same error message prints to the terminal as usual. A plain `>` on its own redirects only standard output and leaves error messages visible. This is why a redirected command can still fail loudly even though its output file is empty.

Environment variables and PATH

An environment variable is a named value the shell makes available to every program it runs, not just to the shell itself. `export NAME=value` sets one and marks it for export, meaning child processes, programs the shell launches, inherit it too:

```
export MYVAR=hello
echo "$MYVAR"
#> hello
bash -c 'echo "in subshell: $MYVAR"'
#> in subshell: hello
```

Without `export`, a variable assignment (`MYVAR=hello`, no `export`) is visible only to the current shell, not to programs it starts. This is the mechanism behind an instruction we give more than once in this book without naming it. Setting `RENV_CONFIG_AUTOLOADER_ENABLED=FALSE` before an R session, as Chapter 12's continuous-integration examples do, works because the shell exports it into the environment that `R` or `Rscript` then reads when it starts. The variable is not an R setting. It is a value the shell handed to the R process at launch, and any process the shell starts can read it with its own language's equivalent of `echo $VAR` (`Sys.getenv('RENV_CONFIG_AUTOLOADER_ENABLED')` in R).

`PATH` is a particular environment variable, a colon-separated list of directories the shell searches, in order, when it looks up a command name. Typing `Rscript` works because some directory in `PATH` contains a program named `Rscript`. `echo $PATH` shows the search list, and prepending a directory to it (`export PATH="$HOME/bin:$PATH"`) makes programs placed there runnable by name without typing their full path.

Draft, November 13, 2026. Content may change.

Permissions and the execute bit

`ls -l` shows a permissions string before each file name, such as `-rw-r--r--`. The first character marks the entry type (`-` for a plain file, `d` for a directory). The next nine are three groups of three, giving read, write, and execute permission for the file's owner, its group, and everyone else.

A script with no execute permission cannot be run directly, even if the file itself is readable text:

```
ls -l greet.sh
#> -rw-r--r--  1 jane  staff  37 Oct 12 07:48 greet.sh
./greet.sh
#> bash: ./greet.sh: Permission denied
```

(zsh words the same failure as `zsh: permission denied: ./greet.sh`.)

`chmod +x greet.sh` adds execute permission for the owner, the group, and others, after which the same command runs:

```
chmod +x greet.sh
ls -l greet.sh
#> -rwxr-xr-x  1 jane  staff  37 Oct 12 07:48 greet.sh
./greet.sh
#> hello from script
```

The `x` bit is what separates 'this file contains runnable instructions' from 'this file is runnable right now'. A shell script downloaded or generated by a template typically starts without it, and `chmod +x` is the usual fix.

`chmod` also accepts a three-digit numeric mode, one digit per group (owner, group, other), each digit the sum of read (4), write (2), and execute (1) for that group. `chmod 644 greet.sh` sets `-rw-r--r--`, read and write for the owner and read-only for everyone else, with no execute bit anywhere. `chmod 755 greet.sh` sets `-rwxr-xr-x`. This is the common mode for a script meant to be run by its owner and read and executed, but not modified, by anyone else. The symbolic form (`+x`, `-w`) changes one bit at a time

Draft, November 13, 2026. Content may change.

without needing to know the rest; the numeric form sets all nine bits in one command.

A note on bash versus zsh

The commands above work the same way in bash and in zsh, the two shells most readers will encounter, and this book's own examples assume bash-compatible syntax throughout. On macOS in particular, Apple switched the default shell from bash to zsh in macOS Catalina (2019). A reader following this book on a current Mac is therefore, by default, typing these commands into zsh, not bash. The everyday commands in this appendix, `pwd`, `ls`, `cd`, `cp`, `mv`, `rm`, redirection, pipes, `export`, `chmod`, behave the same in both, with one exception that a reader of this book will meet: a wildcard that matches nothing. In bash, an unmatched `*.csv` is passed to the command as the literal text `*.csv`, and the command itself then reports that no such file exists. In zsh, the same unmatched pattern is an error of the shell, `zsh: no matches found: *.csv`, and the command never runs at all. The same rule applies to a pattern meant for the command rather than the shell, such as `grep -r --include=*.qmd`, which zsh rejects unless the pattern is quoted (`--include='*.qmd'`). Quoting a wildcard that the command, not the shell, should expand is correct in both shells. Other differences surface only in more advanced scripting (array syntax, for example), which this book does not rely on. A reader who hits a syntax error on a book command that looks exactly as written should suspect a shell difference before suspecting a typo. The running shell can be checked with `echo $0` (`echo $SHELL` reports the login shell, which may differ).

Draft, November 13, 2026. Content may change.

Setup Appendix: Docker and Git on Your Own Machine

Chapter 7 teaches what a container is and why pinning a base image by digest matters. This appendix is narrower and more mundane. Its subject is getting Docker itself to run at all, on the machines most readers actually sit down at, before any of that chapter's content applies. We take up Docker on `arm64` laptops, `--platform linux/amd64` emulation and its consequences, Windows and WSL2, and locked-down institutional laptops. We add no reproducibility theory here. The appendix is a checklist for the moment before the first `docker build` succeeds. Its last two sections do the same for git, which Chapter 4 needs earlier in the course: a preflight to run once per machine before the first commit (Git preflight), and a table of the refusals a new user meets most often (When git refuses).

Apple Silicon (M-series Mac)

Chapter 7 already explains the mechanism. The Rocker images differ. `rocker/r-ver`, the base of the showcase Dockerfile, is published for both `linux/amd64` and `linux/arm64`, so on an Apple Silicon Mac (an `arm64` host) it builds natively without any flag. `rocker/tidyverse` and `rocker/verse` at 4.4.0, the bases of the tidyverse and publishing profiles, publish a `linux/amd64` entry only, so the Mac has nothing native to match, and the flag `--platform linux/amd64` is not an optimization but the only way to get a build at all. For either kind of base, the flag is what makes a Mac build the same image that a CI runner or an `x86` cluster builds, which is why this book passes it to every build. That section of Chapter 7 is worth reading before this one. What follows is the practical sequence for a reader sitting at such a machine for the first time.

Draft, November 13, 2026. Content may change.

1. **Confirm Docker Desktop is installed and running.** `docker version` should print both a `Client` and a `Server` block; a `Client`-only response, or a connection error, means the command is installed but Docker Desktop is not running.

2. **Add `--platform linux/amd64` to every build and run**: `docker build --platform linux/amd64 -t myimage .`. Against an `amd64`-only image such as `rocker/tidyverse:4.4.0`, omitting the flag does not silently succeed at native speed; it fails to resolve a matching entry at all. Against a multi-architecture image such as `rocker/r-ver`, omitting it does silently succeed, natively, and yields an `arm64` image that differs from the one the rest of the lab and the cluster run.

3. **Expect the build to be slower, not wrong.** Docker's own documentation on multi-platform builds warns that QEMU emulation 'can be much slower than native builds, especially for compute-heavy tasks' (Docker Inc., 2024b). A `renv::restore()` that compiles several packages from source is exactly that kind of task; budget minutes, not seconds, for a first build, and do not read slowness alone as a broken Dockerfile.

4. **Confirm emulation is actually what is happening**, rather than guessing from slowness alone. `docker image inspect myimage --format '{{.Architecture}}'` reports the architecture of the image that was actually pulled or built; `amd64` on an Apple Silicon host confirms QEMU emulation is in play. The build log itself also names the platform at the top of each stage (`[linux/amd64 ...]`) when `--platform` was passed explicitly.

Windows with WSL2

This book's images are Linux-based, and Docker Desktop on Windows runs them through the **WSL2 backend**. It does not run them through Hyper-V alone, nor through Windows containers, which target Windows-native images this book never builds. Docker's own current documentation describes the setup directly (opened 2026-10-12). WSL2 itself must be installed, and the Docker Desktop setting **Use WSL 2 based engine** must

Draft, November 13, 2026. Content may change.

be enabled under Settings > General (on by default on systems that support it). A specific Linux distribution is then enabled for Docker integration under Settings > Resources > WSL Integration, which is on by default for the default distribution (Docker Inc., 2024a). Installing WSL2 itself, if it is not already present, is a single elevated command from Windows: `wsl --install`, per Microsoft's own documentation (Microsoft, 2024a).

The problem that costs real time is where the project's files live. Microsoft's own guidance is explicit: for the fastest performance when working from a Linux command line, store files in the WSL Linux filesystem (`/home/<user>/project`), not on a mounted Windows drive accessed through `/mnt/c/...`. The same document calls out such a mount as measurably slower (Microsoft, 2024b). A repository cloned from inside Windows Explorer or PowerShell, then opened from a WSL2 terminal only by crossing the `/mnt/c` mount, pays this cost on every file Docker touches during a build. The fix is to run `git clone` from inside the WSL2 terminal itself, so the repository lands under the Linux filesystem from the start, not to move a Windows-cloned copy afterward.

A second, quieter hazard travels with the same mount boundary: line endings. Git on Windows commonly checks out text files with CRLF line endings by default (`core.autocrlf=true`), while this book's shell scripts, Dockerfiles, and R scripts assume LF, the Linux convention a container's shell expects. A script that looks identical in an editor can fail inside a container with `/bin/bash^M: bad interpreter: No such file or directory` when a shell runs it, or a bare `no such file or directory` when the container runtime executes it directly. The reason is that a trailing `\r` is part of the first line's interpreter path as far as the kernel is concerned, and no interpreter of that name exists. Cloning inside WSL2's own filesystem, where Git's Linux default is LF, avoids this entirely; cloning on the Windows side and later copying into WSL2 does not.

Locked-down institutional laptops

A reader with no administrator rights cannot install Docker Desktop at all, and nothing on the laptop itself will fix this. The options instead route around it through other infrastructure, in roughly increasing order of how much it is worth asking for:

Draft, November 13, 2026. Content may change.

- **A cloud-hosted development environment.** Chapter 8 already covers GitHub Codespaces, built on the Development Containers specification (Development Containers Project, 2022). There it is presented as a way to open a pinned environment in an editor without installing anything locally beyond a browser or a thin client. It requires no local Docker Desktop at all, because the container runs on GitHub's infrastructure, not the laptop.
- **A department-provided remote server or HPC login node,** reached over SSH. Chapter 8's HPC worked example already shows the companion half of this. Apptainer, not Docker, is the standard container runtime on a shared cluster, and it runs unprivileged. That is exactly the property that makes it acceptable where a locked-down laptop's Docker daemon is not. A reader who already has cluster access may find the institutional gap is already closed, just not with Docker.
- **A formal request to institutional IT**, when neither of the above is available. The request is answerable only if it is concrete. Ask specifically for Docker Desktop. If desktop software cannot be approved, ask instead for Docker Engine plus the ability to run containers under the requester's own account. Do not ask merely for 'Docker', which names a product family spanning both. An IT request phrased as 'I need Docker' is easy to deny or stall. On the other hand, one phrased as 'I need to run `docker build` and `docker run` as my own user, without root' is a specification someone can actually approve or substitute for.

Diagnosing a failed or unexpectedly slow build

When a build fails or crawls, work through this sequence before assuming the Dockerfile itself is wrong:

Check	What it tells you
`docker info`	Confirms Docker Desktop is actually running and reachable; also reports the server's architecture.

Draft, November 13, 2026. Content may change.

Check	What it tells you
`docker version`	Separates client-only output (Desktop not running) from a full client-and-server response.
Is `--platform` set, and does the base image publish an `arm64` entry?	An `amd64`-only base on an `arm64` host needs the flag explicitly, and a missing flag there is a pull failure, not a slow build. A multi-architecture base such as `rocker/r-ver` builds without it, natively, but then yields an `arm64` image rather than the `amd64` one CI and the cluster run.
Docker Desktop's allotted disk and memory (Settings > Resources)	A build that stalls rather than errors often has hit a memory or disk ceiling Docker Desktop itself enforces, independent of the host machine's total resources.
On WSL2, is the project on the Windows filesystem (reached as `/mnt/c/...`) rather than in WSL2's own Linux filesystem (`/home/<user>/...`)?	A Windows-side clone is the likely source of a build that is slow specifically on Windows and not elsewhere; cloning again from inside the WSL2 terminal, into the Linux filesystem, is the fix.

On the machine this book was written on (Apple Silicon, Docker Desktop with the `desktop-linux` context), `docker version` reports a `darwin/arm64` client against a `linux/arm64` server (verified, run 2026-10-12). A reader passing `--platform linux/amd64`, as this book does for every build, would see the pulled or built image itself report `amd64` under `docker image inspect`. The daemon underneath, however, is still `arm64`. This combination is the signature of emulation working as intended rather than a misconfiguration.

Draft, November 13, 2026. Content may change.

Git preflight

Chapter 4 assumes four things about git: it is installed, it knows who the analyst is, it names a new repository's first branch `main`, and the forge accepts the analyst's credentials. On a fresh installation none of the four can be taken for granted. The checks below are run once per machine, before the first commit. Each names the chapter of *Happy Git and GitHub for the useR* (Bryan et al., n.d.) that treats the step at length, with the RStudio detail this book omits.

1. **Git is installed.** `git --version` prints one line, such as `git version 2.56.0` on the machine this book was written on. A 'command not found' error means git must be installed first; see 'Install Git' (https://happygitwithr.com/install-git.html).

2. **Git knows who you are.** Every commit records a name and an e-mail address, so set both once:

   ```
   git config --global user.name 'A. Analyst'
   git config --global user.email 'analyst@example.org'
   ```

   Without them, git does one of two things. Where it cannot guess an address, it refuses the commit with 'Author identity unknown' and '*** Please tell me who you are.' Where it can, it builds an address from the login name and the host name, commits under it, and prints only a notice that the identity was 'configured automatically'. The second outcome is the worse one, because the commit succeeds under an address no collaborator would recognize. Use the address registered with the forge account. See 'Introduce yourself to Git' (https://happygitwithr.com/hello-git.html).

3. **New repositories start on `main`.** Unless told otherwise, git 2.56.0 names the first branch `master`, with a hint that the default will change in Git 3.0. The chapter's `git init -b main` handles this per repository; one setting handles it for all of them:

   ```
   git config --global init.defaultBranch main
   ```


Draft, November 13, 2026. Content may change.

Afterward, `git config --global --list` should show three lines, `user.name`, `user.email`, and `init.defaultbranch` (git prints the key in lowercase).

4. **The forge accepts your credentials.** For git operations over HTTPS, GitHub does not accept the website password. It needs a personal access token, which a credential helper keeps in the operating system's store; see 'Personal access token for HTTPS' (https://happygitwithr.com/https-pat.html). Alternatively, `gh auth login` (the GitHub command-line client) signs in through a browser and stores a token, and `gh auth setup-git` makes git use it. The other route is an SSH key registered with the forge; see 'Set up keys for SSH' (https://happygitwithr.com/ssh-keys.html). A token never goes in `.Renviron` or anywhere else inside the repository.

5. **Line endings, on Windows only.** Cloning inside WSL2 avoids the CRLF hazard described above. A repository that is also cloned on the Windows side should commit a `.gitattributes` file with the line `*.sh text eol=lf`, and the same for any other file a container executes; the attribute fixes LF on checkout whatever each user's `core.autocrlf` is. Failing that, set `git config --global core.autocrlf input`, under which git performs no conversion on checkout.

6. **A first push.** Create the repository on the forge *empty*, with no README, license, or `.gitignore`. Each of those makes a first commit that the local repository does not share. The push is then refused, and so is the pull meant to repair it (the 'fetch first' and 'unrelated histories' rows of Table 1). Then, in the local repository:

```
git remote add origin
 ↪  https://github.com/lab/diabetes-cohort.git
git push -u origin main
```

Success ends with the line `branch 'main' set up to track 'origin/main'.` See 'Connect to GitHub' (https://happygitwithr.com/push-pull-github.html) and 'Existing project, GitHub last' (https://happygitwithr.com/existing-github-last.html).

Draft, November 13, 2026. Content may change.

When git refuses

The messages in Table 1 are quoted as git 2.56.0 printed them, on macOS, in scratch repositories built for this appendix on 2026-11-08, shortened where an ellipsis appears. Two rows are the exception and say so: their messages come from the forge, not from git, and were not reproduced. The table follows the model of the Docker table above: find the symptom, then read across.

Table 1.: Git refusals a new user meets, indexed by the message git or the forge prints. Messages from git 2.56.0, run 2026-11-08; the two forge rows were not reproduced.

Symptom (what git prints)	Cause	Remedy
`Author identity unknown … *** Please tell me who you are.`	No `user.name` or `user.email` is set, and git cannot guess one.	Set both (Git preflight, step 2) and repeat the commit. If git guessed instead, fix the last commit with `git commit --amend --reset-author`, which git itself suggests.
A refused HTTPS push reporting that password authentication was removed (the forge's message, quoted in the Happy Git chapter on tokens; not reproduced)	The account password was given where the forge requires a token.	Store a personal access token through a credential helper or `gh auth login`, or switch the remote to SSH (step 4).

Draft, November 13, 2026. Content may change.

Symptom (what git prints)	Cause	Remedy
`! [rejected] main -> main (fetch first)` and `Updates were rejected because the remote contains work that you do not have locally.`	The remote branch holds commits the local one lacks, because a collaborator pushed first or the forge made a first commit, and git will not overwrite them.	`git pull`, then `git push`. If the pull stops with `fatal: Need to specify how to reconcile divergent branches.`, run `git pull --no-rebase` to merge (or set `pull.rebase false` once). Never force-push a shared branch.
`fatal: refusing to merge unrelated histories`	The local and forge repositories were started separately, typically with a README made on the forge.	If the forge's commit holds only template files, `git pull --no-rebase --allow-unrelated-histories origin main` joins the two once. Next time, create the forge repository empty or start by cloning it.
`You are in 'detached HEAD' state.`; `git status` reports `HEAD detached at` and a hash	A commit or tag was checked out rather than a branch.	Look around freely; `git switch main` returns. To keep commits made there, `git switch -c <new-branch-name>` first, as git's own advice says.

Draft, November 13, 2026. Content may change.

Symptom (what git prints)	Cause	Remedy
`CONFLICT ( content): Merge conflict in analysis.R` and `Automatic merge failed; fix conflicts and then commit the result.`	Both branches changed the same lines.	Edit the file down to one version and delete the markers, then `git add` and `git commit`; or `git merge --abort` (Chapter 4).
`error: src refspec main does not match any`	There is no local branch named `main`: either nothing has been committed yet, or the branch is `master`.	Make the first commit; check the name with `git branch --show-current`; rename with `git branch -m main`; push again.
A push refused for a file that exceeds the forge's size limit, 100 MiB on GitHub (GitHub, n.d.-a) (the forge's message; not reproduced)	A data file or a large output was committed.	First ask whether it holds PHI; if so, the exposure procedure of Chapter 4 comes before any git command. If the file is only in the last unpushed commit, `git rm --cached <file>`, ignore it, and `git commit --amend` (or `git reset HEAD~1` if it was the commit's only file). If it is deeper in the history, rewrite with `git filter-repo`.

Draft, November 13, 2026. Content may change.

Cheat-Sheet Index

Each content chapter ends with a two-column reference table of the commands, rules, and terms introduced in that chapter. This appendix lists what each table covers, part by part, with a reference to the full table, where the surrounding context explains any term that is not self-explanatory.

The entries below are read from the chapters' own tables when the book is built, so the index cannot fall out of step with them.

Foundations

Chapter 2, The Reproducibility Crisis (Section 2.13). Reproducibility; Replicability; Robustness; Computational failure; Statistical failure; `RNGkind()`; `stringsAsFactors`; Determinant stack; Floor, not ceiling; Honest level; Record model use (Section 2.10.2).

Chapter 3, Levels of Reproducibility (Section 3.14). L0 locatable; L1 pinned packages; L2 pinned environment; L3 verified; Capture vs. validation; Reproduced (this book); `zzc verify --full`; Honest statement.

Chapter 4, Version Control with Git and GitHub (Section 4.16). `git init -b main`; `git add <file>`; `git commit -m '...'`; `git status`; `git log --oneline`; `git show <hash>`; `git switch -c <name>`; `git remote add origin <url>`; `git push -u origin main`; `git tag -a v1.0 -m '...'`; `!pattern` in `.gitignore`; `git check-ignore -v <path>`; `git rm --cached <file>`; Git refuses; `git filter-repo --sensitive-data-removal`; Rule of thumb; Habit 1.

Chapter 5, The Research Compendium (Section 5.19). `DESCRIPTION`, `NAMESPACE`, `R/`; `.Rbuildignore` entry `^analysis$`; The Five Pillars; 'If I deleted this file, could the code regenerate it?'; Five archetypes; Progressive disclosure; Rule of thumb; Habit 7; `usethis::create_`

Draft, November 13, 2026. Content may change.

`package()`, `use_r()`, `use_package()`; `tinytest::setup_tinytest('.')`; `devtools::load_all()`, `document()`, `check()`; README computational requirements.

Capturing the Environment

Chapter 6, Pinning Packages with renv (Section 6.14). `renv::init()`; `renv::snapshot()`; `renv::restore()`; `renv::dependencies(files)`; `make check-renv` / `zzrenvcheck::check_packages(auto_fix = TRUE)`; `zzrenvcheck::check_packages(auto_fix = FALSE, strict = TRUE, error_on_fail = TRUE)`; `options(repos = c(CRAN = '.../cran/YYYY-MM-DD'))`; Rule of thumb; Habit 3.

Chapter 7, Pinning the Environment with Docker (Section 7.17). `FROM image@sha256:...`; `ARG PPM_SNAPSHOT=.../jammy/YYYY-MM-DD`; `ENV RENV_CONFIG_REPOS_OVERRIDE=...`; `ENV LANG=... LC_ALL=...`; `ENV TZ=UTC`; `ENV OMP_NUM_THREADS=1`, `OPENBLAS_NUM_THREADS=1`; `WORKDIR /project`; `COPY renv.lock renv.lock` + `renv::restore()`; `docker build --platform linux/amd64`; `docker buildx imagetools inspect image:tag`; `docker image inspect --format '{{index .RepoDigests 0}}' image:tag`; Rule of thumb.

Chapter 8, Backends and Runtimes: Nix, Podman, and Apptainer (Section 8.14). `podman run ...`; `apptainer build x.sif docker://img@sha256:...`; `apptainer exec --cleanenv --no-home --bind "$PWD" x.sif Rscript --vanilla ...`; `rix::rix(r_ver=, r_pkgs=, project_path=)`; `nix-shell default.nix`; `.binder/Dockerfile`; `.devcontainer/devcontainer.json`; Rule of thumb; Habit 2; Habit 7.

Validation

Chapter 9, Literate Programming and the Rendered Report (Section 9.18). `library(pkg)`, one per package; An inline R expression in prose; `knitr::opts_chunk$set(...)`; `SOURCE_DATE_EPOCH`; `make docker-render`; Five landmines; Rule of thumb; Habit 6; Habit 7; `freeze: auto`.

Draft, November 13, 2026. Content may change.

Chapter 10, Automating the Analysis: Make and targets (Section 10.16). Rule: `target: prerequisites`; Make recipe line; `make`; `touch file`; `tar_target(x, expr)`; `tar_target(x, path, format = 'file')`; `tar_make()`; `tar_outdated()`; `tar_visnetwork()`; `tar_render()` / `tar_quarto()`; Capture vs. validation; Honest claim; LLM response cache.

Chapter 11, Testing and Code Quality (Section 11.19). `expect_equal(a, e)`; `expect_equal(a, e, tolerance = 1e-8)`; `expect_true()`, `expect_false()`, `expect_identical()`, `expect_error()`, `expect_inherits()`; `tinytest::test_package('pkg')`; `exit_file('message')`; `make lint` / `make style`; `test-baseline*.R` naming; Rule of thumb; Habit 4; Habit 5; Code review, six questions (Section 11.12).

Chapter 12, Continuous Integration and Verification (Section 12.19). `error-on: '"warning"'` (GitHub) / `error_on = "warning"` (`rcmdcheck`); `zzrenvcheck::check_packages(strict = TRUE, error_on_fail = TRUE)`; `uses: actions/checkout@<full SHA> # v7.0.1`; `renv::dependencies('analysis')`; `rmarkdown::render(...)` in a CI step; `docker build -t compendium:ci .`; `test-baseline-*.R` naming; `digest(file = path, algo = 'sha256')`; `.Rbuildignore` entry for `analysis/`; Rule of thumb; Habit 4; Habit 6.

Collaboration and Stewardship

Chapter 13, Collaboration, Archiving, and Stewardship (Section 13.13). `docker inspect --format '{{index .RepoDigests 0}}' <img>`; `docker run --rm <img> find / -xdev -path '*raw_data*'`; `apptainer build out.sif docker://<img>@sha256:...`; `renv.lock` + `Dockerfile`, committed; Two-layer model; Union lockfile; `rang::resolve(pkgs, snapshot_date = ...)`; Recorded run date; Restricted data (Section 13.5.2); DOI deposit; Honest claim at deposit; Trusted research environment (Section 13.5.3); CONSORT 2025 item 4, SPIRIT 2025 item 6.

Draft, November 13, 2026. Content may change.

Glossary

This appendix collects, in one alphabetical list, the concepts defined in each chapter's 'vocabulary of this chapter' section together with the commands, functions, tools, and services the book relies on. The per-chapter vocabulary sections remain in place; this is a second view of them and not a replacement.

Each entry names the chapter that introduces or develops it, and a term developed in more than one chapter names each. Entries are alphabetized by the term itself, so a function is filed under its own name rather than under the package that supplies it: look for `tar_make()` under T and `renv::restore()` under R.

A

`apptainer build`. Converts an OCI image into a single SIF file that runs without root, which is what makes it acceptable on a shared cluster and depositable as an archival artifact. (Chapter 8)

Apptainer (formerly Singularity). A container runtime built for shared high-performance computing, where users are not administrators. Its SIF format is a single checksummable file. (Chapter 8)

Archetype. The kind of research object a compendium is (data analysis, manuscript, package, simulation study, or blog), which varies the layout and the gates but not the underlying machinery. (Chapter 5)

Audit trail. In this book, the commit history read as a dated, attributed record of what changed in an analysis and, through the messages, why. (Chapter 4) *Regulated sense:* 21 CFR 11.10(e) requires secure, computer-generated, time-stamped audit trails of the actions that create, modify, or delete electronic records, and record changes may not obscure previously

Draft, November 13, 2026. Content may change.

recorded information (*21 CFR Part 11*, n.d.). A git history can be rewritten, so it does not meet that requirement unless the hosting system and its procedures prevent rewriting. (Chapter 4)

B

Backend. The mechanism that supplies the environment. Docker is the container backend; Nix is a container-free backend that declares the environment directly. (Chapter 8)

Base image. The starting image a recipe builds on, named on the `FROM` line. Here, a Rocker image providing a fixed R version on a fixed operating system. (Chapter 7)

Baseline test. A test, in a file named `test-baseline-*.R`, that recomputes key outputs and compares them with values recorded in an earlier session and committed, using `expect_equal()` with a stated tolerance. It is the evidence L3 requires. (Chapter 11, Chapter 12)

Binder. A service that builds an environment from a public repository and serves it in a browser, so a reviewer can run an analysis with nothing installed locally. (Chapter 8)

Branch. A movable name for a line of development. The default branch is conventionally called `main`. (Chapter 4)

C

Capture. Fixing an input so that it cannot drift: writing a lockfile, building a container image. Capture sets levels L0 to L2; L3 is reached by validation, which pins nothing new. (Chapter 3)

Checksum. A short fingerprint, computed from a file's or an object's bytes by a hash function such as SHA-256, with the property that any change to the input, however small, changes the fingerprint. A stored checksum lets a later run detect that an input has been altered. (Chapter 12)

Draft, November 13, 2026. Content may change.

Code chunk. A block of executable code embedded in a literate document, delimited from the prose, whose output is inserted into the rendered document at the chunk's position. (Chapter 9)

CODECHECK. A scheme in which an independent person re-runs a submitted analysis and certifies the result, catching the class of failure where the author's own checks encode the author's own assumptions. (Chapter 12)

Codespaces and devcontainers. A specification, and a hosted implementation of it, for describing the development environment an editor should open a project inside. (Chapter 8)

Commit. A recorded snapshot of the tracked files at one moment, carrying an author, a timestamp, a message, and an immutable identifier (a *hash*). The atomic unit of history. (Chapter 4)

Computational reproducibility. Obtaining the same numerical results from the same data and the same code, on another machine or at a later time. (Chapter 2)

conda and pixi. Cross-language environment managers that pin R and Python packages without a container. Both run natively on Windows, for the packages conda-forge builds there. (Chapter 8)

Container. A running instance of an image. One image can start many containers; each begins from the same frozen filesystem. (Chapter 7)

Container image. A packaged environment, built from a recipe, that carries the operating system, the system libraries, and the language runtime. It fixes the files a run starts from, not the machine: thread counts and the CPU still vary, and an image built for one processor architecture runs on another only under emulation, where numerical results can differ. (Chapter 3)

Controlled access. A deposit arrangement in which a repository holds restricted data and releases them only to approved users under an agreement, rather than to anyone who downloads. Its purpose is to honor a data-use agreement and the terms a participant consented to while the record stays discoverable and citable. (Chapter 13)

Continuous integration (CI). A hosted service that, on every committed change to a repository, checks out the code onto a fresh virtual machine,

Draft, November 13, 2026. Content may change.

rebuilds the project, and runs a defined set of checks. It reports success or failure automatically. (Chapter 12)

D

Daemon. A background system service that runs continuously with elevated privilege. Docker relies on one; Podman and Apptainer do not, which is why they run without administrative rights. (Chapter 8)

Data-availability statement. A sentence in a manuscript, or in its compendium, naming where the data behind its results can be obtained. It gives an open deposit's DOI, the name and access route of a controlled-access repository, or the terms under which the data cannot be shared at all. (Chapter 13)

Data-validation test. A check on the data itself rather than the code: that a value lies in a plausible range, that no impossible ages appear, that the expected columns are present. (Chapter 11)

Dated snapshot. A view of a repository frozen as it stood on a particular calendar day, so that an install run today and one run next year fetch the same versions. (Chapter 6, Chapter 7)

Dependency graph. The graph whose nodes are the steps and whose edges record that one step consumes what another produces. A valid pipeline's graph is a *directed acyclic graph* (DAG): directed because inputs flow to outputs, and acyclic because a step cannot, directly or transitively, depend on its own output. (Chapter 10)

Dependency-manifest gate. A CI check that enforces the dependency invariant of Chapter 6. In its strict, R-package form it requires every package the code uses to be declared in `DESCRIPTION` and pinned in `renv.lock`. In a research compendium, `DESCRIPTION` declares only the API exported from `R/`, while analysis scripts legitimately draw on packages present in `renv.lock` but not in `DESCRIPTION`. There the gate enforces the weaker inclusion that every package the code uses is pinned in `renv.lock`. (Chapter 12)

Draft, November 13, 2026. Content may change.

`DESCRIPTION`. The human-authored file that declares a project's direct dependencies by role (`Imports`, `Suggests`, `Depends`), with optional version floors. (Chapter 6)

Determinant stack. The layered set of things a computational result depends upon, from the hardware at the base through the environment and the packages to the analysis code and the data at the top. A layer is 'pinned' when it has been fixed to a recorded value. (Chapter 2, Chapter 3)

Determinism (of a render). The property that rendering the same source under the same environment produces a byte-for-byte identical output file. A rendered PDF is usually not deterministic in this strict sense, because it embeds timestamps and other build-time metadata. (Chapter 9)

`digest()`. Computes a content hash of an R object. Used to verify that a regenerated computational output matches a recorded one, on objects rather than on rendered documents. (Chapter 12)

Docker. The most widely used container platform, and one instrument for reaching a pinned environment rather than a synonym for reproducibility. (Chapter 7)

`docker build` and `docker run`. Build an image from a recipe and start a container from an image. The image is durable; the container is ephemeral. (Chapter 7)

Docker Hub and GHCR. Container registries. A pushed image is a cache and not the record: the committed recipe is the record, because a registry can delete, rate-limit, or repoint what it holds. (Chapter 13)

`docker pull` and `docker inspect`. Fetch an image and read its metadata, including the content digest that a reproducible recipe pins instead of a tag. (Chapter 7)

E

Expectation. A single assertion, such as `expect_equal(actual, expected)`, that passes or fails. (Chapter 11)

`expect_equal()` and the other expectations. The assertions of `tinytest`. A file of bare, top-level expectation calls is a complete test file,

Draft, November 13, 2026. Content may change.

and `expect_inherits()`, `expect_true()`, and `expect_error()` cover most of the rest. (Chapter 11)

F

FAIR. A set of principles holding that a research object should be Findable, Accessible, Interoperable, and Reusable, formulated for data and later adapted for research software. (Chapter 13)

Five Safes. A framework for controlled data access: safe projects, safe people, safe settings, safe data, and safe outputs. (Section 13.5.3)

Forge. A platform that hosts a version-controlled repository and runs its continuous integration. GitHub and GitLab are the two principal forges; GitLab is frequently self-hosted by universities. (Chapter 12, Chapter 13)

Freeze (Quarto). A project option under which a render reuses a document's stored results from `_freeze/` instead of executing its code. With `freeze: auto`, a document re-executes only when its source changes, not when its data, packages, or R change. (Section 9.8)

G

`git add`. Stages changes for the next commit. Staging is what lets a commit be one idea rather than a snapshot of whatever is on disk. (Chapter 4)

`git clone`, `push`, `pull`, `fetch`. Move history between a local repository and a remote. A remote serves backup, distribution, and collaboration, which are three purposes worth keeping separate. (Chapter 4)

`git commit`. Records the staged changes with a message and an immutable identifier. The atomic unit of the history. (Chapter 4)

`git diff`. Shows what changed, between the working tree and the index, or between any two commits. (Chapter 4)

`git filter-repo`. Rewrites history to remove content, producing new hashes for every commit from the offending one forward. The tool of last

Draft, November 13, 2026. Content may change.

resort after protected data has been committed, and no substitute for notifying whoever is responsible for the data. (Chapter 4)

`git init`. Creates a repository in a directory. The `.gitignore` should be written before the first commit, not after. (Chapter 4)

`git log` and `git show`. Read the history: `log` lists commits, `show` prints one commit's message and its diff together. (Chapter 4)

`git switch` and `git merge`. Move between branches and rejoin them. A merge commit is a commit with more than one parent (usually two), which is what records that lines of work came back together. (Chapter 4)

`git tag`. Marks a particular commit permanently, such as the state of an analysis at submission. Cheap to place and expensive to omit. (Chapter 4)

GitHub Actions. GitHub's continuous-integration service, configured by YAML workflow files under `.github/workflows/`. (Chapter 12)

GitHub and GitLab. The two principal forges. GitLab is frequently self-hosted by institutions that forbid pushing to a public forge, and a compendium's reproducibility machinery should be portable between them. (Chapter 13)

`.gitignore`. A file listing path patterns that git should not track, used to keep derived outputs, large data, and, critically, sensitive files out of the history. (Chapter 4)

`groundhog.library()`. Loads a package as it stood on a given date, a lighter alternative to a lockfile that reconstructs by date rather than by record. (Chapter 6)

I

Image. A frozen, read-only filesystem together with metadata, bundling an operating system, R, a set of packages, and configuration. It is analogous to a snapshot of a fully installed computer, packaged so it can be shipped. (Chapter 7)

Draft, November 13, 2026. Content may change.

Input-addressed. Identified by a hash of the build inputs rather than by a mutable name, so that a reference cannot silently point at different content later. (Chapter 8)

`install.packages()`. Base R's installer. Pointed at a live repository it resolves 'the current version' to whatever is current on the day it runs. For this reason a reproducible recipe points it at a dated snapshot or replaces it with a lockfile restore. (Chapter 6)

Integration test. A check that several components behave correctly when composed into a pipeline, from raw input to final result. (Chapter 11)

Invalidation. The act of marking a target out of date because something it depends on, a file, an upstream target, a function, or a package, has changed. (Chapter 10)

K

`knitr` and Sweave. The engines that execute code inside a literate document. Sweave came first; `knitr` superseded it and underlies both R Markdown and Quarto's R support. (Chapter 9)

L

Landmine. A pattern in a document that makes the render succeed for its author and fail for a later reader, by reaching outside the captured and committed material. (Chapter 9)

LaTeX and TinyTeX. The typesetting system a PDF render requires, and a small R-managed distribution of it. Belongs in the image rather than assumed on the host. (Chapter 9)

Layer. Each instruction in an image's recipe produces one filesystem layer. An image is the stack of its layers, and identical layers are shared and cached across images. (Chapter 7)

`library()` and `require()`. The two calls that attach a package to the current session. Both are scanned when a dependency check derives what

Draft, November 13, 2026. Content may change.

the code uses, so a package loaded this way must be declared and pinned. (Chapter 6)

Linting. Static analysis of source code for stylistic and likely-error patterns, performed by `lintr`, without running the code. (Chapter 11)

`lintr` and `styler`. Static analysis and automatic reformatting for R source. Legibility is a validation feature: code nobody can read is code nobody can check. (Chapter 11)

Literate programming. Writing narrative prose and executable code together in one source document, so the document is at once the program and its explanation. The term is Knuth's. (Chapter 9)

Lockfile. A machine-generated file, `renv.lock`, recording the R version, the package repositories, and the exact version and source of every package the project uses, with metadata copied from each package's `DESCRIPTION`. (Chapter 3, Chapter 6)

M

`make`. Rebuilds targets whose modification time is older than a prerequisite's. Conservative in one direction and unreliable in the other: it will rebuild what did not need it, and miss a change that leaves timestamps untouched. (Chapter 10)

N

Nix and Guix. Declarative package managers that pin an entire dependency graph, from the system libraries upward, to a single revision. They reach a pinned environment from a recipe rather than from a built artifact. (Chapter 8)

Draft, November 13, 2026. Content may change.

O

OCI image. An image built to the Open Container Initiative specification, the common format that Docker, Podman, and Apptainer can all consume. A Docker image is an OCI image. (Chapter 8)

OCI tarball or SIF file. A container image exported as a single file, checksummable and depositable as an archival artifact. An OCI image is exported as a tar archive, whereas SIF is the self-contained single-file format used by the Apptainer runtime. (Chapter 13)

Oracle. The source of independent truth a test compares against; the known correct answer. (Chapter 11)

P

Output checking. Review of every result leaving a trusted research environment against statistical disclosure control rules, such as minimum cell counts. (Section 13.5.3)

Package churn (code rot). The gradual breakage of once-working code as the packages beneath it are updated over time. (Chapter 2)

Package library. The directory into which R installs packages. A project-local library belongs to one project rather than being shared across all of a user's work. (Chapter 6)

Package record. The `renv.lock` entry for one package: its name, exact version, and source, plus fields such as `Imports` and `License` copied from its `DESCRIPTION`. It contains no checksum of the package's files; the `Hash` field written by `renv` before 1.1.0 was a fingerprint of `DESCRIPTION` fields used to key the package cache. (Chapter 6)

Package repository. A server, such as CRAN or the Posit Package Manager, from which packages are installed. (Chapter 6)

`pacman::p_load()`. Loads packages and installs any that are missing. Convenient and a landmine in a sealed document: it reaches the network at render time and pins nothing. (Chapter 9)

Draft, November 13, 2026. Content may change.

pandoc. The document converter a render invokes after the code has run, turning the executed markdown into the requested output format. (Chapter 9)

Persistent identifier (DOI). A durable, resolvable name for a deposited artifact, such as a Digital Object Identifier, that continues to resolve even if the artifact moves. (Chapter 13)

Pipeline. An analysis expressed as a set of steps, each declaring its inputs and outputs, so that the whole can be re-executed by a single command that runs the steps in dependency order. (Chapter 10)

`pkg::fun()`. A namespaced call, which uses a function without attaching its package. It still creates a dependency, and a scanner that looked only for `library()` would miss it. (Chapter 6)

Podman. A daemonless, rootless container runtime that runs the same images as Docker. The usual answer where Docker Desktop licensing is a constraint. (Chapter 8)

`podman run`. Starts a container without a root daemon, consuming the same image format Docker produces. Swapping the runtime changes the launcher and not the reproducibility level. (Chapter 8)

Posit Package Manager. A package repository that serves dated snapshots and prebuilt binaries, so that an install run today and one run next year fetch the same versions. (Chapter 6)

`precommit`. Runs checks before a commit is recorded, keeping trivia out of the history. It can be skipped, which is why it complements rather than replaces continuous integration. (Chapter 11)

Pre-commit hook. A script that git runs automatically before a commit is recorded, used here to run the linter and the styler as a gate. (Chapter 11)

Prerequisite. In Make, an input a target depends on. If any prerequisite is newer than the target, the target is out of date. (Chapter 10)

Profile. The base-image bundle a project is built from, such as `minimal`, `tidyverse`, `rstudio`, or `publishing`. It selects the `FROM` image. (Chapter 7)

Draft, November 13, 2026. Content may change.

Progressive disclosure. The property that one layout supports a project's evolution from one archetype to another with no reorganization. (Chapter 5)

Prompt injection. Instructions embedded in content a model or agent reads (a README, an issue, a CI log, a data file) that it then follows as if the user had given them. (Section 2.10.1)

Protected path. A file or directory, such as `inst/tinytest/`, `inst/baseline/`, `renv.lock`, or `.github/workflows/`, whose every change the analyst reviews line by line before merging, whoever or whatever made it. (Section 2.10.1)

Pull request (merge request). A proposal, made on the remote, to merge one branch or fork into another, reviewed before it is accepted. (Chapter 4)

Purely functional package manager. A package manager, such as Nix, in which the installed result is a deterministic function of the declared inputs, so that the same declaration yields a deterministic, reproducible environment. (Chapter 8)

Q

Quarto and R Markdown. The literate-document systems this book uses: prose and executable code in one source, rendered to HTML, PDF, and other formats. (Chapter 9)

`quarto render`. Executes a document's code and produces its output formats. With `freeze` configured it may replay a cache instead of executing, which is why a gate that must exercise the code has to clear the cache first. (Chapter 9)

R

`R CMD check`. The standard R command that builds a package, runs its tests, and applies dozens of standardized checks; here it is an integrity gate on the compendium's package skin (`R/`, the tests, `DESCRIPTION`). It

Draft, November 13, 2026. Content may change.

does not check `analysis/`, which `.Rbuildignore` excludes from the build. (Chapter 5, Chapter 12)

`rang`, `groundhog`, and `capsule`. Alternatives and complements to a lockfile workflow. `rang` reconstructs a package set from a date and needs no prior lockfile; `groundhog` loads packages as of a date written into the script in advance; `capsule` layers a lazier discipline over `renv`, materializing the pinned library only when the work is reproduced or checked. Only `rang` helps when no lockfile survives, and its result is an approximation rather than a record. (Chapter 6)

Raw versus derived data. Immutable original inputs versus the processed datasets regenerated from them by code. (Chapter 5)

Recipe (or command). The shell command Make runs to bring a target up to date from its prerequisites. (Chapter 10)

Registry. A server that stores and distributes container images, from which collaborators pull a pre-built environment. GitHub Container Registry (GHCR) and Docker Hub are the two in common use. (Chapter 13)

Remote. A copy of the repository hosted elsewhere, typically on GitHub or GitLab, used for backup, distribution, and collaboration. `origin` is the conventional name of the primary remote. (Chapter 4)

`remotes::install_github()`. Installs a package from a git repository rather than a package repository. Reproducible only if pinned to a commit or a tag. (Chapter 13)

Render gate. A CI check that re-renders the compendium's report (an R Markdown or Quarto document) from source and fails the build if the render errors. (Chapter 5, Chapter 12)

Render (verb and noun). To execute a literate source document and produce a finished output file (HTML, PDF, Word); and the resulting file itself. (Chapter 9)

Rendering toolchain. The programs a render invokes behind the scenes: `knitr` or the Quarto engine to run the code, `pandoc` to convert the result, and, for PDF, a LaTeX distribution to typeset it. (Chapter 9)

Draft, November 13, 2026. Content may change.

`renv` and `packrat`. The current and superseded R dependency managers. Both pair a project-local library with a lockfile; `renv` rebuilt the interface around a small set of verbs and a shared cache. (Chapter 6)

`renv::init()`. Creates a project-local library and lockfile, taking a project from unpinned to pinned packages. (Chapter 6)

`renv::restore()`. Reads the lockfile and installs the recorded versions into the library. The command a collaborator runs on a fresh clone. (Chapter 6)

`renv::snapshot()`. Reads the library and writes what it finds into the lockfile. Snapshot writes; restore reads. (Chapter 6)

`renv::status()`. Reports disagreement between the library and the lockfile, and is the first diagnostic when a restore appears not to have worked. (Chapter 6)

Replicability. Obtaining consistent findings from a new study, with new data, aimed at the same question. (Chapter 2)

Repository (repo). A directory whose contents and full history git tracks, held in a hidden `.git` subdirectory. A *clone* is a complete copy of a repository, history and all. (Chapter 4)

Reproduced. Obtaining the same computational result by re-executing the same code on the same data. Contrast **replicated**, obtaining a consistent finding from a new study with new data. (Chapter 3)

Reproducibility test. A check that a recorded computational result regenerates. The form that can fail is the baseline test; two runs in one session do not qualify. (Chapter 11)

Research compendium. A single self-describing directory bundling the code, data, prose, and environment record of an analysis, organized by convention so that it can be located and re-run. (Chapter 2, Chapter 5)

Restore. The reverse of a snapshot: reading the lockfile and reinstalling exactly the packages it records. (Chapter 6)

`rix::rix()`. Generates a `default.nix` pinning an R version, a package set, and the `nixpkgs` revision that determines everything beneath them. (Chapter 8)

Draft, November 13, 2026. Content may change.

RNG kind. The random-number algorithm in force, set by `RNGkind()`. Two runs with the same seed but a different kind draw different numbers. (Chapter 3)

`RNGkind()`. Selects the random-number generator algorithm. Its `sample.kind` default changed in R 3.6.0, so the same seed draws different numbers under different R versions unless the kind is recorded. (Chapter 2)

Robustness. Reaching the same qualitative conclusion from the same data under different but defensible analytical choices. (Chapter 2)

Rocker. The community project that publishes the standard R Docker images. (Chapter 7)

Rootless. Able to run under an ordinary user account, without the `root` superuser privilege. A requirement on shared clusters, where users are not administrators. (Chapter 8)

R-package skin. The minimal set of files (`DESCRIPTION`, `NAMESPACE`, an `R/` directory) that makes a directory a valid R package, and therefore checkable by the R build tools. (Chapter 5)

Runner. The fresh, ephemeral machine (usually a virtual machine or a container) on which the CI service executes the workflow. It is discarded after the run, so nothing persists between runs except what the workflow explicitly restores. (Chapter 12)

Runtime. The program that runs a container image: Docker, Podman, or Apptainer. The runtime is interchangeable when the image is a standard OCI image. (Chapter 8)

S

`saveRDS()` and `readRDS()`. Write and read a single R object to a file. The ordinary way to record the baseline that a reproducibility test compares against. (Chapter 11)

Sealed render. A document in which every reported number and figure is regenerated from the code that produces it on each build, so the prose cannot drift from the analysis. (Chapter 9)

Draft, November 13, 2026. Content may change.

Seed and RNG kind. The seed (`set.seed`) is the fixed starting point that makes a random draw repeatable. The RNG kind is the generator algorithm that turns that seed into numbers, and its default has changed across R versions. (Chapter 2)

`sessionInfo()`. Reports the R version, platform, and attached packages of the current session. A record of what ran, not a mechanism for making it run again. (Chapter 3)

`set.seed()`. Fixes the starting point of the random-number stream so that a stochastic result repeats. Necessary but not sufficient: the generator algorithm must be fixed as well. (Chapter 2)

SIF. The Singularity Image Format, a single-file archive that Apptainer produces from an image, suitable for depositing alongside a compendium. (Chapter 8)

Silent default. A behavior that software applies unless told otherwise, and that can change between software versions without any error or warning to the analyst. (Chapter 2)

Snakemake and Nextflow. Pipeline tools from the bioinformatics community, language-agnostic and built for cluster execution. (Chapter 10)

Snapshot. The act of scanning a project, resolving the versions currently installed, and writing them into the lockfile. (Chapter 6)

`SOURCE_DATE_EPOCH`. A conventional environment variable, honored by many document tools, that fixes the timestamp a build embeds, so that two builds do not differ merely because they ran at different moments. (Chapter 9)

Staging area (index). The set of changes marked, with `git add`, to go into the next commit. Staging lets a commit hold a chosen subset of the current changes. (Chapter 4)

Staleness (out-of-date). The condition of a target whose inputs have changed since it was last built, so that its stored output no longer reflects its inputs. (Chapter 10)

Styling. Automatic reformatting of source code to a consistent layout, performed by `styler`. (Chapter 11)

Draft, November 13, 2026. Content may change.

T

Tag. A permanent, human-readable name for one specific commit, used to mark a milestone such as a manuscript submission. (Chapter 4)

Tag and digest. A tag, such as `rocker/tidyverse:4.4.0`, is a human-readable pointer that may later be repointed. A digest, `@sha256:...`, is an immutable hash of the image's exact content. The same digest always denotes the same bytes. (Chapter 7)

Target. A single step in the pipeline, together with the output it produces. The word is shared by Make and `targets`, with a shift of emphasis: for Make a target is a file, for `targets` it is more often an R object. (Chapter 10)

`targets`. A function-oriented pipeline tool for R that hashes function bodies and R objects rather than file timestamps, so it detects changes a Makefile can miss. (Chapter 10)

`tar_make()`. Runs a `targets` pipeline, rebuilding only the targets that are out of date and everything downstream of them. (Chapter 10)

`tar_outdated()`. Reports which targets `tar_make()` would rebuild, without rebuilding them. The way to check a prediction about the dependency graph before paying for it. (Chapter 10)

`tar_target()`. Declares one node of a `targets` pipeline: a name, the expression that computes it, and optionally a format such as `file`. (Chapter 10)

`tar_visnetwork()`. Draws the pipeline's dependency graph, derived from the code rather than from a hand-maintained declaration. (Chapter 10)

The Five Pillars. The five artifacts a compendium manages, one per capture point: the `Dockerfile`, `renv.lock`, `.Rprofile`, the source code, and the data. (Chapter 5)

The level ladder (L0 to L3). The four-rung scale, L0 locatable, L1 pinned packages, L2 pinned environment, L3 verified, used throughout this book to grade how far down the stack an analysis has been secured. (Chapter 3)

Draft, November 13, 2026. Content may change.

`tinytest`. A lightweight, zero-dependency R testing framework in which a test file is a plain script of bare, top-level expectation calls (Loo, 2020). (Chapter 11)

`tinytest` and `testthat`. R testing frameworks. `tinytest` has no dependencies and its test files are plain scripts of bare expectations, which is why this book uses it. (Chapter 11)

Trusted research environment. A secure computing environment, operated by a data provider, in which approved researchers analyze data that never leave it; outputs are released only after output checking. Also called a secure data environment or a data enclave. (Section 13.5.3)

Two-layer team model. A division of a project into a slow-moving environment layer, fixed once by a team lead, and a fast-moving package layer, extended by every contributor. The environment layer is the team image (the operating system, the R version, the system libraries, and the packages every project in the lab uses), pinned by digest; the package layer is the project lockfile. (Chapter 13)

U

Union model. The rule that the project lockfile records the union of every package any contributor has installed, so that a rebuilt image carries what the whole team needs, not only what one member used. (Chapter 13)

Unit test. A focused check that one small piece of code, typically a single function, behaves as intended on known inputs. (Chapter 11)

Up-to-date (skippable). The condition of a target whose inputs have not changed since it was last built, so that rebuilding it would reproduce the stored output and can be skipped. (Chapter 10)

V

Validation. Confirming that the captured inputs actually suffice, by rebuilding and re-running. Validation does not pin anything new; it checks.

Draft, November 13, 2026. Content may change.

(Chapter 3) *Regulated sense:* in FDA software guidance, validation is confirmation, by examination and objective evidence, that software specifications conform to user needs and intended uses (U.S. Food and Drug Administration, 2002). That is a broader and more formal claim than a rebuild and re-run, and passing this book's checks is not a validation record in that sense. (Chapter 11)

Verification. Rebuilding the compendium's environment from its recipe, re-running the analysis in it, and comparing the regenerated outputs with a committed baseline, within a stated tolerance, by means of a baseline test. Passing it is what the ladder calls level L3. (Chapter 12) *Regulated sense:* in FDA software guidance, verification is objective evidence that the outputs of a development phase meet the requirements specified for that phase (U.S. Food and Drug Administration, 2002). A baseline test can contribute such evidence only if the baseline is traced to a stated requirement. (Chapter 11)

W

Workflow. The file, written in YAML, that tells the CI service what to do: which events trigger it, what environment to run in, and what commands to execute. On GitHub these files live under `.github/workflows/`; on GitLab the single file is `.gitlab-ci.yml`. (Chapter 12)

Working tree. The ordinary files in the repository directory as they currently exist on disk, as opposed to the versions recorded in the history. (Chapter 4)

Z

Zenodo. A general-purpose research repository, operated at CERN, that mints a DOI for any deposited file set. (Chapter 13)

zzcollab. The framework this book uses to compose the reproducibility tools into one system, organized around the levels ladder. (Chapter 5)

Draft, November 13, 2026. Content may change.

`zzrenvcheck::check_packages()`. Checks the dependency triad and can repair it, writing a missing package into `DESCRIPTION` and injecting a resolved entry into the lockfile. (Chapter 6)

Draft, November 13, 2026. Content may change.

Credits

The framework at the center of this book, `zzcollab`, and the argument that motivates it are developed in the accompanying methodological paper, *zzcollab: A Flexible Extension of rrtools for Reproducible, Collaborative Research Compendia in R*. The book draws its worked apparatus, its reproducibility-levels ladder, and its capture-versus-validation framing from that work and from the framework's documentation.

The book also draws on the reproducibility literature it surveys, in particular the research-compendium lineage of Gentleman and Temple Lang and of Marwick, Boettiger, and Mullen; the containerization work of the Rocker project; and the many rules-and-practices papers cited in the chapters.

Cover artwork palette anchored on a verification teal `#14746f`. Typography follows the book's house style: body text in a serif face, code in a monospace face.

Colophon

This book was produced with Quarto. Its examples use R, the `zzcollab` framework, Docker, and `renv`, each introduced in the chapter that first requires it.

The book is built as a Quarto project, and the sources render to HTML and PDF with `quarto render`. Wherever we present a computed figure or table, that result was produced by executing the code shown beside it, not typed by hand. It was not necessarily produced by the build that made this copy. The book renders with Quarto's `freeze: auto` setting, under which a page's code is re-executed only when that page's source file changes; otherwise the results stored in the freeze cache at its last execution are replayed. A change to a package version, a data file, or a function under `R/` does not by itself trigger re-execution. A printed result therefore agrees with the code beside it as of that page's last execution. The table below follows the same discipline: every cell in the 'Actual' column is read from the R session that last executed this page, and every cell in the 'Locked' column from the committed `renv.lock`.

Table 1.: The rendering environment of the last execution of this page, read from the live session, beside the versions the committed lockfile records.

Field	Actual	Locked
R version	4.6.0	4.6.1
Platform	aarch64-apple-darwin23	not recorded
Quarto version	1.10.18	not recorded
Package: tibble	3.3.1	3.3.1
Package: ggplot2	4.0.3	4.0.3
Package: withr	3.0.3	3.0.3
Package: knitr	1.51	1.51
Package: rmarkdown	2.31	2.31

Draft, November 13, 2026. Content may change.

Table 1.: The rendering environment of the last execution of this page, read from the live session, beside the versions the committed lockfile records.

Field	Actual	Locked
renv.lock present	yes	

The last row is the book's own honest reproducibility disclosure, made the same way the book asks every analysis to make it: by naming the artifact rather than asserting the level. When this page last executed, an `renv.lock` for the book's own source was present, so the table reports 'yes'. This sentence and the table are computed in the same execution, so they cannot disagree with each other. A lockfile is the L1 artifact. The L2 step, a `Dockerfile` that pins the operating system and the R runtime, is not yet in place.

A lockfile records versions; it does not make the session use them. In the session that executed this page, 1 row differed from the lockfile (R version). Restoring the lockfile would reinstall the locked package versions but not R itself, which only a pinned environment, the L2 rung, fixes.

The HTML edition is the accessible one. Its pages are checked with an automated accessibility tool, its figures carry alt text, and its folded answers open from the keyboard. The PDF is not yet a tagged PDF: we tried PDF/UA tagging and found that it is not yet compatible with the line breaking that keeps long code lines on the printed page, so a screen-reader user should prefer the HTML.

This page last executed on 2026-11-08.

Draft, November 13, 2026. Content may change.