Learn to Reason About History

A Seven-Volume Git Curriculum

Series Plan and Chapter Outlines

Flyxion
Independent Researcher

2026

Overview

This is a seven-volume Git curriculum built around a single governing sentence: *Don't memorize Git. Learn to reason about history.* Where most Git instruction treats the tool as a set of commands to be memorized against remembered scenarios, this series treats Git as an object model and a history — a content-addressed graph that can be understood well enough to predict before it is executed. Each volume opens the model further rather than adding another layer of commands: files become snapshots, snapshots become content-addressed objects, objects acquire parent relations and become a graph, the graph becomes evidence to be investigated, single graphs become many graphs coordinated by people, coordinated graphs become an architecture, and finally the graph becomes a substrate for delegated, partially autonomous work.

A second governing principle runs alongside the first: *the repository is not the companion to the textbook — the repository is part of the textbook.* All seven volumes share a single, deliberately engineered companion Git repository rather than seven independent teaching repositories. Its commit history is course material in the literal sense: exercises are encoded as immutable annotated tags into a real graph, and a number of exercises are answerable only by inspecting the shape of that graph — reachability, merge base, ancestry — rather than by reading a diagram in the text. The repository's history is intentionally not sanitized. Failed approaches, abandoned branches, regressions, and ugly intermediate states are preserved as pedagogical material rather than cleaned away before publication.

Each volume is built around a different running application, chosen so that the application's own real needs motivate each Git concept in the order it is introduced, rather than concepts being introduced and then illustrated with a contrived example. The applications accumulate: a personal Docker web service, a Stable Diffusion workstation, an Ollama/Granite research assistant, an essay-writing and LaTeX publishing pipeline, a small open-source scientific-computing library, and finally the five of these reconnected into one system that a final volume of software agents is delegated to help maintain.

Repository Architecture, in Brief

The seven volumes share one companion repository on a single trunk, with each volume's application occupying its own top-level directory (`v1-webservice/`, `v2-workstation/`, and so on). Annotated tags of the form `v<n>/ch<nn>/start` are the permanent, citable textbook interface; ordinary branches are disposable working surfaces students create from those tags and are free to destroy. A permanent `refs/textbook/*` namespace holds objects — abandoned commits, deleted-branch tips, pre-rewrite states — that specific exercises require to remain reachable without being reachable from any branch a student would casually merge into; these refs are exempt from garbage

collection at the published origin, though a plain `git clone` does not fetch them by default and the standard clone instructions add the necessary fetch refspec explicitly. Reference solutions live in a wholly separate `refs/solutions/*` namespace, distributed via a second bundle or remote and fetched only on demand, so that a student's ordinary clone cannot spoil any exercise by inspection. Every exercise carries a small machine-readable manifest (volume, chapter, exercise number, start tag, verification type, and either an expected solution tag or an interpretive rubric), keeping the exercise system independent of any one grading platform. Published instructional history is never rewritten; corrections arrive as new commits and, where needed, new tags rather than force-updates.

The specification governing all of this — the Series Repository Specification — has grown three further conventions directly out of building and reviewing the volumes against it: exercises that run `git gc`, `prune`, or `reflog expire` must operate on a disposable local clone rather than the shared origin, since an object that looks safely unreachable from one volume's vantage point may be load-bearing elsewhere in the series; exercises that repeat an operation differently against the same starting state (rebase, then compare against a merge of the same proposal) must explicitly preserve that starting state under `refs/textbook/*` before the first variant runs; and, more generally, a chapter's stated opening or closing state must be traceable to an actual sequence of prior operations rather than merely asserted in prose — the same check performed by building the real repository, applied by reasoning where the repository itself is not being built.

Volume I is, at the time of this writing, the only volume with an actually-built companion repository, used to validate the method before committing to it for the rest of the series. Building it caught two real design errors no amount of outline review would have found on its own — a planned merge that was silently a fast-forward, and a false assumption about what a plain `git clone` fetches by default. Volumes II through VII were reviewed by the same discipline applied at the outline level: tracing whether each chapter's stated premise is actually supported by what the preceding chapters would have produced, in place of building each repository outright. Each volume's chapter outline below is followed by a short status note recording exactly what that review found and what remains open — the intent being that this document alone, without needing to reconstruct the reasoning behind it, is enough to resume drafting from wherever the series was left off.

The document closes with an epilogue drawing the seven volumes back together: the single-line compression of the whole curriculum, the seven successive meanings a commit acquires across the series, two symmetries between volume pairs that only become visible once every outline exists side by side, and the closing argument that Volume VII's central claim about delegated software agents — that producing a change and authorizing it into history are different acts — was already true of every volume that came before agents entered the picture at all.

Contents

Volume 1

Git from First Principles

Running application: a Dockerized personal web service

Governing Idea

The reader begins with almost nothing: a directory whose state matters, and no instinct yet for why that should require any special tooling. Snapshots, commits, parents, branches, and eventually a repository are derived from that single problem rather than introduced as vocabulary. Commands appear only once the conceptual machinery already demands them, so that by the time `git commit` is used for the first time, the reader already understands what a commit is. The chapter-to-chapter progression is change → remember → compare → name → diverge → combine → recover → publish.

Chapter Outline

Ch. 1. A Directory That Changes. The reader loses the ability to answer "what did this say three versions ago" with no Git yet involved, establishing the problem historical state solves.

Ch. 2. Remember This State. `git init`, `add`, `commit` derived from the need to say "this state matters"; working tree vs. recorded state.

Ch. 3. What Changed? `diff`, `diff --staged`, `show`, `log`; working tree, index, and history as three potentially different states.

Ch. 4. A Commit Is Not a Save Button. Selective staging; a commit describes one meaningful transition, not the current contents of a directory.

Ch. 5. Naming History. Commit identifiers, `HEAD`, `HEAD~1`; history as a chain of named positions.

Ch. 6. Put It in a Container. A Dockerfile introduced broken, then fixed and committed as a known-good state.

Ch. 7. Things Git Should Not Remember. `.gitignore`; `.env` vs. `.env.example`; filesystem membership vs. historical membership.

Ch. 8. Go Back Without Going Back. `restore`, `checkout -- path`, `revert`; recovering old content without erasing subsequent history.

Ch. 9. Two Possible Futures. Branches introduced as names for divergent historical continuations, not copies of the project.

Ch. 10. Bringing Futures Together. A genuine two-parent merge; predicting the resulting tree before merging.

Ch. 11. When Both Futures Changed the Same Thing. A real semantic merge conflict; Git records disagreement but does not resolve meaning.

Ch. 12. The Fast-Forward That Barely Looks Like a Merge. Fast-forwarding motivated by asking what new commit is actually necessary.

Ch. 13. I Committed the Wrong Thing. `restore`, `reset`, `amend`, distinguishing changes to files, the index, and history.

Ch. 14. I Think I Lost My Work. A first, gentle encounter with the reflog after a branch disappears.

Ch. 15. Give This State a Name. Annotated tags; a moving branch name vs. a fixed release marker.

Ch. 16. Somewhere Else. Remotes, `fetch`, `push`, `pull`; Git $\neq$ GitHub.

Ch. 17. Clone It and Rebuild It. Cloning into an empty directory and rebuilding the Docker image as a concrete test of reproducibility.

Ch. 18. A Complete Historical Investigation. No command sequence supplied; the reader must choose the operations to diagnose and repair a deliberately altered history.

Status and Open Items

Volume I is the only volume with an actually-built companion repository, constructed to validate the method before committing to it for the rest of the series. That build caught two real errors that outline review alone would not have found: the planned Chapter 10 merge was silently a fast-forward until main was given an independent commit during Chapter 9 to force a genuine two-parent merge, and a plain `git clone` was found not to fetch `refs/textbook/*` by default, which corrected an originally false claim in the specification and produced the explicit clone-time fetch-refspec instruction now required series-wide. Solution tags were also migrated from ordinary annotated tags into the separate `refs/solutions/*` namespace as peeled commit refs, and a real clone-and-fetch test confirmed a default clone exposes no solution. No open issues remain outstanding for this volume beyond eventually drafting prose against the built repository.

Volume 2

Inside Git

Running application: a versioned Stable Diffusion workstation

Governing Idea

Volume I treated Git from the outside, as working files becoming recorded history. Volume II opens the repository and asks what that recorded history actually consists of. A working Stable Diffusion installation immediately forces the question of which parts of a system constitute its reproducible identity and which are merely artifacts it produces — source, configuration, prompts, workflows, model files, caches, and generated images cannot all be treated the same way. The progression is file → content → object → tree → commit → reference → reachability → reproducible state.

Chapter Outline

Ch. 1. A Workstation Is More Than Its Files. Classifying workstation contents into source, configuration, recipe, dependency, model, cache, generated artifact, and metadata before touching Git internals.

Ch. 2. What Does Git Actually Store? `cat-file`, `rev-parse`; commit → tree → blob made concrete.

Ch. 3. Content Has an Identity. Two identically-worded files under different names; filename ≠ content identity via `hash-object`.

Ch. 4. The Hash Is Not the File. Object identity as a function of content and type; editing one character and examining the resulting blob.

Ch. 5. Where Did the Filename Go? Trees as relationships among objects; `ls-tree`.

Ch. 6. Build a Tree by Hand. Constructing blobs and a tree with plumbing commands, on its own detached exercise coordinate.

Ch. 7. Build a Commit by Hand. A hand-built commit object; state (tree) vs. history (commit) made explicit.

Ch. 8. Where Is the Branch? Branches located inside `.git/refs/`; a branch as a reference resolving to an object.

Ch. 9. The Index Is a Real Thing. Inspecting the index directly; predicting the next tree from a mixed staged/unstaged/unchanged scenario.

Ch. 10. The Seven-Gigabyte Question. Should a multi-gigabyte checkpoint become an ordinary Git object? Introduces a model manifest recording identity and acquisition recipe instead.

Ch. 11. Artifact Versus Recipe. An experiment record capturing model, workflow, prompt, seed, and parameters, explicitly referencing Chapter 10's model manifest, without requiring the generated image itself to enter history.

Ch. 12. Ignore Is Not Forget. A large artifact committed by accident, then `.gitignored`; the repository does not shrink.

Ch. 13. History Has Weight. `count-objects`, `verify-pack`; not present in HEAD $\neq$ not present in history, using Chapter 12's accidental commit.

Ch. 14. Loose Objects and Packed Objects. Packfiles introduced conceptually; logical identity $\neq$ physical storage layout.

Ch. 15. Reachability. An abandoned commit investigated for reachability from refs; connects to the series repository's own `refs/textbook/*`.

Ch. 16. Garbage Collection Is About Reachability. `git gc` explained via retention $\sim$ reachability $+$ recovery policy; run in a disposable clone per the specification's isolation rule, never against the shared origin.

Ch. 17. The Same State Can Have Different Histories. Two commits with equivalent trees but different parents or metadata.

Ch. 18. What Exactly Does a Commit Identify? Varying message, author, timestamp, or parent in isolation to see what changes identity.

Ch. 19. Large Files Without Pretending They Are Small. Git object database vs. external artifact store, with Git retaining only an identity or pointer.

Ch. 20. Reproducing the Workstation. Reconstructing the workstation elsewhere from the manifest; repository completeness $\neq$ containing every byte.

Ch. 21. Prove What Produced This. Given an output image and incomplete information, determining what the repository can and cannot establish about its provenance.

Ch. 22. Construct a Repository Without `git add`. The capstone: blob $\rightarrow$ tree $\rightarrow$ commit $\rightarrow$ ref built entirely from plumbing, closing the abstraction opened in Chapter 6.

Status and Open Items

Reviewed at the outline level rather than by building the repository. Open items to resolve before drafting prose: Chapters 6–7's hand-built tree and commit need to be explicitly placed on their own detached exercise coordinate rather than appearing to replace the chapter's real commit sequence; Chapter 9's three-file index scenario and Chapter 12's accidental large-artifact commit both need their originating chapter pinned down explicitly rather than assumed; Chapter 10's model manifest and Chapter 11's experiment record should explicitly reference each other so Chapter 21's

provenance question has an actual chain to follow rather than two parallel, unlinked patterns; and Chapter 15's abandoned commit needs an explicit statement of whether it is meant to survive Chapter 16's garbage-collection exercise via `refs/textbook/v2/` or be deliberately destroyed by it. The outline above already reflects the intended fixes (Chapter 11 references Chapter 10's manifest, Chapter 16 states the isolation rule) but these should be verified against the eventual repository build.

Volume 3

The Geometry of Git

Running application: an Ollama + Granite research assistant

Governing Idea

Volume II showed history as a content-addressed structure. Volume III asks what happens when history has more than one possible continuation. A local research assistant that must choose among several plausible summarization strategies — fixed chunking, semantic sectioning, hierarchical summary, recursive reflection — turns branches from an invented teaching device into the literal historical representation of genuine competing hypotheses. The progression is ancestry $\rightarrow$ divergence $\rightarrow$ comparison $\rightarrow$ merge base $\rightarrow$ integration $\rightarrow$ transplantation $\rightarrow$ reconstruction $\rightarrow$ history design.

Chapter Outline

Ch. 1. One Assistant, One History. A minimal working summarizer; reviews Volumes I–II material to keep the volume independently readable.

Ch. 2. A Branch Is a Possible Continuation. `experiment/fixed-chunks`; a branch as a movable name attached to one historical continuation.

Ch. 3. Two Experiments at Once. `experiment/semantic-sections` branched from the same ancestor; inspecting two legitimate divergent histories.

Ch. 4. Ancestry Is a Relation. The ancestor relation $\preceq$ tested experimentally with `merge-base --is-ancestor`.

Ch. 5. What Actually Changed Between Two Histories? Two- and three-dot comparison; change relative to a shared past rather than a raw file diff.

Ch. 6. The Merge Base. Naming and predicting the merge base of two histories before asking Git.

Ch. 7. Three-Way Merge. Fixed-chunking merged toward semantic-sectioning; reasoning about $\Delta(M, A)$ and $\Delta(M, B)$ before merging.

Ch. 8. A Conflict Is Historical Information. Both experiments alter the same function for different reasons; the resolution introduces information present in neither parent.

Ch. 9. Merge Commits Preserve a Fact. Inspecting the two-parent merge commit; equal trees do not imply equal commits.

Ch. 10. Fast-Forward Revisited. Fast-forwarding derived graph-theoretically from $A \preceq B$, contrasted with true divergence.

Ch. 11. A Third Hypothesis. `experiment/hierarchical-summary`, branched from an earlier point in history than the previous experiments.

Ch. 12. A Fourth Hypothesis: Recursive Reflection. `experiment/recursive-reflection`, built as several small commits for later chapters to manipulate.

Ch. 13. Commit Sets. Revision-set reasoning; Git's revision syntax as a language for selecting graph vertices.

Ch. 14. One Useful Commit, Not the Whole Branch. Cherry-picking a logging improvement out of a rejected algorithm.

Ch. 15. A Commit Is Not Its Patch. Comparing an original commit with its cherry-picked counterpart; same change, different object.

Ch. 16. Rebase Begins with a Question. Motivating rebase by asking what a branch would look like had it begun from today's main, after main has genuinely advanced with new commits in the intervening chapters.

Ch. 17. Rebase Reconstructs History. Predicting which object identities must change before running the rebase.

Ch. 18. Why Rebase Conflicts Feel Strange. A multi-commit rebase where one change no longer applies cleanly; conflict as replay against a different base.

Ch. 19. Merge or Rebase Is Not a Moral Question. Comparing the same development via merge and via rebase, using hierarchical-summary's pre-rebase tip explicitly preserved under `refs/textbook/v3/` before Chapters 16–18 rewrote it.

Ch. 20. Rebase –onto. An experiment that accidentally began from another experimental branch; selecting commits, old boundary, and new base.

Ch. 21. Cherry-Pick Versus Rebase. Choosing the operation from the intended graph transformation rather than habit.

Ch. 22. Revert on a Branched History. Reverting a regression after other work has followed it; which parent is mainline relative to the reversal.

Ch. 23. Criss-Cross and Ambiguous Ancestry. A preserved, deliberately complicated graph exposing multiple merge bases.

Ch. 24. Octopus Merge and Many Histories. Several independent improvements integrated with one multi-parent commit.

Ch. 25. First-Parent History. First-parent traversal as one particular projection of the graph.

Ch. 26. Topology Versus Time. Commits whose timestamps mislead; ancestral order, not the wall clock, determines dependence.

Ch. 27. Which Experiment Won? Running the summarization strategies against a small corpus; branches represent alternatives without requiring an advance decision.

Ch. 28. Preserve the Losing Experiment. A rejected algorithm's tip preserved under `refs/textbook/v3/` after its working branch is removed.

Ch. 29. Equivalent Result, Different History. Two paths (merge vs. rebase-and-cherry-pick) to the same final tree; what each history preserves and suppresses.

Ch. 30. Design the History Before Touching Git. Capstone: given a target graph and semantic requirements, decide which operations construct it before writing any commands.

Status and Open Items

Reviewed at the outline level. The outline above already incorporates the two fixes found necessary: Chapter 16 explicitly states that main advanced with new commits between Chapters 11 and 16, and Chapter 19's merge-vs-rebase comparison explicitly uses hierarchical-summary's pre-rebase tip preserved under `refs/textbook/v3/` before Chapters 16–18 rewrite it. Two items remain open: whether Chapter 21's rebase is a deliberate second pass on a different new base or risks duplicating Chapters 16–18's operation should be clarified in prose, and Chapter 22's reverted regression needs an explicit earlier chapter (not currently named) where some experiment is actually merged into main, since Chapters 7–9's merges are only between two experimental branches.

Volume 4

Git Archaeology

Running application: an essay-writing and LaTeX publishing pipeline

Governing Idea

Volume III ended by asking what history the reader wants to construct. Volume IV begins with the inverse problem: what history produced what is in front of me now. A substantial essay and its build pipeline, developed over a simulated months-long history of drafts, reorganizations, citations, regressions, and quiet corrections, gives the reader something closer to real intellectual history than a sequence of programming exercises. The central methodological principle is that Git can preserve evidence without automatically explaining what the evidence means; the progression is symptom → evidence → localization → attribution → causation → recovery → reconstruction.

Chapter Outline

Ch. 1. The PDF Is Wrong. A missing section with no explanation; observe before intervening.

Ch. 2. Ask History a Better Question. Path-narrowed `log`; the missing section discovered to have been renamed before it disappeared.

Ch. 3. What Did This Commit Actually Do? A commit event vs. its patch; commit messages are evidence, not ground truth.

Ch. 4. Follow the File. A fresh, hands-on rename-and-edit demonstration (distinct from Chapter 2's discovered rename) introducing rename detection.

Ch. 5. Where Did This Paragraph Come From? Pickaxe-style search; asking which transition changed the presence of content, not merely which commits touched a file.

Ch. 6. Blame Is Not Blame. `git blame` attacked as attribution under a traversal rule, not causation.

Ch. 7. The Formatting Commit That Changed Everything. A large formatting-only commit that dominates blame output; textual modification ≠ semantic modification.

Ch. 8. The Error Is Older Than the Broken Line. An early definition change breaks a later formula; manifestation time ≠ introduction time.

Ch. 9. The Build Used to Work. A known-good and a broken tag frame the natural problem for binary search.

Ch. 10. Bisect. Manual `git bisect` as search over ancestry rather than a memorized command.

Ch. 11. Automating the Investigation. The project's own build command used as bisect's oracle; the repository interrogated computationally.

Ch. 12. When Good and Bad Aren't Enough. Skipped commits and non-monotonic bugs; a tool inherits the assumptions of the question given to it.

Ch. 13. Find the Semantic Regression. Bisecting over a semantic property via a validation script, not mere build success.

Ch. 14. A Citation Changed Meaning. Surrounding prose drifts until a correct citation supports a stronger claim than its source warrants.

Ch. 15. Search Every Revision. Searching a remembered phrase across all history, including an abandoned branch.

Ch. 16. Recover a Deleted Section. Restoring the Chapters 1–2 section's content as a new historical event, not a return to the old state.

Ch. 17. Recover a Deleted Branch. Locating a removed branch's former tip via the reflog and other evidence.

Ch. 18. The Reflog Is Local Memory. Commit-graph history vs. reference-movement history as two different kinds of history.

Ch. 19. Recover from a Bad Reset. Reconstructing a branch tip after a deliberately destructive `reset --hard`.

Ch. 20. Recover from a Bad Rebase. A conflict resolved by accidentally deleting a paragraph, recovered via reflog evidence.

Ch. 21. ORIG_HEAD and Temporary Evidence. Git's own recovery breadcrumbs after disruptive operations.

Ch. 22. The Stash Is a Historical Object Too. Investigating, partially recovering, and deliberately dropping a stash.

Ch. 23. Dangling Does Not Mean Meaningless. A distinct minor incident: dangling objects investigated for meaning Git itself cannot supply.

Ch. 24. fsck as Excavation. `git fsck` treated as archaeological survey rather than corruption checker.

Ch. 25. Before Garbage Collection. A deadline forces rescuing recoverable-but-fragile objects by establishing durable reachability.

Ch. 26. Who Changed This Definition? Reconstructing a definition's evolution across renames and rewordings, callback to Chapter 8's $\rho(x) = f(x)$, using log, show, blame, diff, and search together.

Ch. 27. What Happened Between These Releases? Comparing tagged editions `essay-v0.7` and `essay-v0.8` as historical explanation, not mere diff.

Ch. 28. The Innocent Commit. A regression whose visible failure follows an unrelated commit; resisting nearest-preceding-change as cause.

Ch. 29. Reconstruct the Writer's Intent Carefully. Evidential language (establishes / suggests / is consistent with / does not establish) applied to a confusing editing sequence.

Ch. 30. Provenance Without Authorship. What repository provenance can and cannot establish about intellectual or legal authorship.

Ch. 31. Generated Files as False Evidence. A staged accidental commit of `.aux/.log/.toc` files misleads a naive search for a phrase's origin.

Ch. 32. Reconstruct a Missing Build Environment. Distinguishing a broken source from a changed environment across Dockerfile, Makefile, and dependencies.

Ch. 33. Bisect the Pipeline, Not Just the Essay. Expanding the causal search boundary into styles, templates, and the bibliography processor.

Ch. 34. Build an Evidence Ledger. A structured claim/evidence/commit/interpretation/confidence record making an investigation auditable.

Ch. 35. Two Plausible Histories. Evidence compatible with two explanations; the correct answer is that the repository does not determine which.

Ch. 36. The Repository Crime Scene. Capstone: a badly damaged project and no supplied command sequence; the deliverable is a written archaeological report, staging a fresh fourth disappearance distinct from Chapters 2, 16, and 23.

Status and Open Items

Reviewed at the outline level. The recurring "lost content" premise originally appeared three times (Chapters 1–2, 16, and 23) with no stated relationship between them; the outline above resolves this by having Chapter 16 pay off Chapters 1–2's mystery, Chapter 23 stand as a distinct minor incident, and Chapter 36's capstone stage an entirely fresh fourth disappearance. Two items remain open: the `essay-v0.7/essay-v0.8` release tags used in Chapter 27 still need to be placed explicitly somewhere in the Chapters 1–26 timeline and added to the tag vocabulary, and Chapter 31's premise that generated files were historically committed needs an explicit staging chapter (its own, or an earlier one) rather than being merely supposed.

Volume 5

Git in the Open

Running application: a small open-source scientific-computing library

Governing Idea

Until now the reader has mostly reasoned about one graph. Collaboration introduces many partially overlapping graphs held by different people, machines, and hosting systems, and the book's question shifts from "what history exists" to "whose repository knows what, and how does history move between them." The progression is repository → remote → exchange → collaboration → review → integration → governance.

Chapter Outline

Part I — More Than One Repository

Ch. 1. There Is No "The Repository." Two clones (Alice's and Bob's) diverge the moment either commits; neither is metaphysically out of date.

Ch. 2. Clone Is Historical Replication. Replicated object IDs are identical across repositories, unlike Volume III's cherry-picked commits.

Ch. 3. What Is a Remote? A remote as configuration describing another repository, with neither side privileged.

Ch. 4. origin Is Just a Name. Renaming, adding, and deleting remotes changes nothing about the commit graph.

Ch. 5. Remote-Tracking References. `origin/main` as locally recorded knowledge of another repository at a past synchronization.

Ch. 6. Fetch Changes Knowledge, Not Work. `git fetch` updates what the repository knows without touching the working tree.

Ch. 7. Fetch Before Merge. Explicit `fetch` then `merge origin/main` before any convenience command is introduced.

Ch. 8. Pull Is Not Primitive. `git pull` as orchestration (fetch plus an integration policy), comparing pull-by-merge and pull-by-rebase.

Part II — Publishing History

Ch. 9. Push. Alice publishes `feature/numerical-integrator`; a push proposes objects and a reference update, not an upload.

Ch. 10. Why Push Can Be Rejected. Bob advances the shared branch first; rejection follows from Volume III's ancestry relation.

Ch. 11. Non-Fast-Forward Is Information. A rejection as a signal to investigate divergence, not an annoyance to bypass.

Ch. 12. Force Push. Deliberately rewriting a published experimental branch; permission to request a non-fast-forward update.

Ch. 13. Force-with-Lease. Conditional mutation: update the reference only if the remote still has the expected state.

Part III — Collaboration Begins

Ch. 14. The First Contributor. A second, distinct contributor implements a new numerical method; author vs. committer introduced.

Ch. 15. Forks. A separate repository whose history initially overlaps strongly with another, then diverges.

Ch. 16. Synchronizing a Fork. Fetching upstream, integrating locally, and republishing; local, fork, and upstream graphs held simultaneously.

Part IV — The Pull Request

Ch. 17. Git Does Not Have Pull Requests. A PR as a hosting abstraction proposing that history reachable from B be integrated into history reachable from A.

Ch. 18. The Pull Request as Proposed History. Examining a PR's commit sequence, diff, base, head, and merge base as complementary interpretations.

Ch. 19. Review the Commits. A five-commit PR reviewed for whether each commit is independently intelligible.

Ch. 20. Review the Diff. The same PR reviewed as one aggregate change relative to its base.

Ch. 21. Review Is Not Approval of Authorship. A reviewer verifies correctness and scope, not who originated every idea or line.

Part V — Changing a Proposal During Review

Ch. 22. Review Comments Become New History. A correction added mid-review; the reviewer distinguishes previously reviewed content from new content.

Ch. 23. Amend or Add Another Commit? Preserving the review conversation vs. preparing a clean eventual history, with no universal answer.

Ch. 24. Rebase During Review. Upstream changes while the PR is open; the proposal is rebased and its pre-rebase state preserved under `refs/textbook/v5/` before this happens, so the same starting proposal is available for Chapter 25.

Ch. 25. Merge Upstream Instead. Repeating the exercise using a merge against the preserved pre-rebase state; the Volume III question of which relationship to represent, now with social consequences.

Part VI — Integration Policies

Ch. 26. Merge Commit. Integrating a completed feature with an explicit two-parent merge that preserves its branch topology.

Ch. 27. Squash Merge. A many-commit PR squashed into one integration commit; proposal history vs. canonical integration history.

Ch. 28. Rebase Merge. The same feature integrated by replaying its commits onto main.

Ch. 29. Same Code, Three Histories. Three preserved copies of one starting branch integrated three different ways; identical trees, distinct histories, and the consequences for archaeology, bisectability, and attribution.

Part VII — Continuous Integration

Ch. 30. The Repository Gets a Robot. CI added; every proposed change now triggers format, lint, type-check, and test jobs.

Ch. 31. CI Tests a Commit, Not an Intention. A green check means specific predicates passed for a specific state, not that the change is correct.

Ch. 32. Why "Works on My Machine" Is Historical. Comparing environments; the project learns to version more of its execution recipe.

Ch. 33. Required Checks. Git capability $\neq$ project permission; the hosting layer can refuse what Git itself would allow.

Part VIII — Collaboration Under Conflict

Ch. 34. Two Contributors Change the Same Model. A genuine, non-pathological conflict resolved semantically.

Ch. 35. Integration Order Matters. Merging Alice then Bob, then resetting to the branches' preserved pre-merge tips and merging Bob then Alice; integration is not necessarily commutative.

Ch. 36. Dependent Pull Requests. Contributor B's feature depends on contributor A's unmerged feature; stacked branches.

Ch. 37. Stacked Changes. Reviewing several small, dependency-ordered proposals independently while preserving their relationship.

Part IX — Repository Governance

Ch. 38. Who May Change Main? Branch protection and required reviews; possible in Git $\neq$ permitted by the project.

Ch. 39. CODEOWNERS and Review Routing. Path-based specialist review as coordination, not territorial ownership.

Ch. 40. Maintainers Are Integrators. A maintainer curates which histories become reachable from important references.

Part X — Releases

Ch. 41. When Is a Commit a Release? An annotated tag for v1.0; branch, tag, and hosting-level release distinguished.

Ch. 42. Semantic Versioning as Project Policy. Git knows a tag's name, not what its numbering scheme is supposed to mean.

Ch. 43. Release Branches. A 1.x fix alongside unfinished 2.0 development on main.

Ch. 44. Backporting. Cherry-picking a main-line fix onto an older release branch in a realistic maintenance context.

Part XI — Trust

Ch. 45. Who Made This Commit? Author and committer fields as text claims, not proof of authorship.

Ch. 46. Signed Commits and Tags. What a signature actually establishes, and what it does not.

Ch. 47. Trusting Keys Is Another Graph. Identity, key, and policy relationships as a structure distinct from the commit DAG.

Part XII — Distributed Failure

Ch. 48. The Main Server Disappears. Every sufficiently complete clone contains the project; survival is not tied to one hosting account.

Ch. 49. Reconstruct the Project from a Clone. Restoring branches and tags from a surviving clone; what can and cannot be recovered.

Ch. 50. The Hosting Platform Is Not the Repository. Stating explicitly what the preceding forty-nine chapters demonstrated about Git vs. hosting infrastructure.

Part XIII — Capstone

Ch. 51. A Real Open-Source Release. The reader moves through contributor, reviewer, integrator, release maintainer, and archivist roles across a simulated multi-week collaborative development, ending by reconstructing the repository after the canonical remote disappears.

Status and Open Items

Reviewed at the outline level. This volume produced the series' first recurring structural finding rather than a one-off: the "repeat the exercise with a different operation on the same starting point" pattern appeared three times (Chapters 24–25, 26–29, and 35) with no stated preservation mechanism, which directly motivated the specification's new §7.2. The outline above already states the required preservation at each site. One item remains open: Chapter 9 (Alice pushes `feature/numerical-integrator`) and Chapter 14 ("The First Contributor" implements a new numerical method) still need their identity and timeline reconciled in prose, since Chapter 19's five-commit pull request under review needs an unambiguous author.

Volume 6

Git at Scale

Synthesis volume, reconnecting all five previous applications

Governing Idea

The reader now has five applications in hand and the problem is no longer any single Git operation but deciding how they should coexist: what belongs in the same history. All of this volume's identity-changing work — assembling the monorepo, splitting components out, and rewriting history — is understood to run against a derived working copy of the shared trunk rather than the canonical repository, so that Volumes I–V's published tags remain untouched throughout; the volume's capstone re-architecture is the one operation applied back to the real trunk, as a forward-moving commit rather than a rewrite. The progression is project → repository boundary → workspace → dependency → artifact → automation → scale → repository architecture.

Chapter Outline

Part I — Bring the Projects Together

Ch. 1. Five Histories Meet. The five applications' working copy considered as a system, asking whether it should be one history or several.

Ch. 2. The Repository Boundary. Criteria around atomic changes, release independence, permissions, and coupling; software boundary $\neq$ repository boundary.

Ch. 3. The Monorepo. Constructing the unified working tree; one commit legitimately spanning several applications.

Ch. 4. The Cost of Shared History. Making the monorepo hurt — irrelevant CI, unrelated tags, unnecessary clone weight — as an honest tradeoff, not a verdict.

Part II — Paths as Structure

Ch. 5. Path-Limited History at Scale. Volume IV's path-narrowed history revisited in a large repository; one graph, many projections.

Ch. 6. Cross-Cutting Commits. A single commit spanning three applications for one conceptual change; commit cohesion $\neq$ directory locality.

Ch. 7. CODEOWNERS in a Monorepo. Volume V's review routing at larger scale, and where directory structure alone stops being sufficient.

Part III — One Repository, Several Working Trees

Ch. 8. The Working Tree Was Never the Repository. `git worktree`; repository $\neq$ working tree, revisiting Volume II.

Ch. 9. Work on Two Branches Simultaneously. A genuine need (urgent release fix alongside a large refactor) motivating a second worktree.

Ch. 10. Worktrees and Branch Safety. Git's safeguards against checking out the same branch in incompatible worktrees.

Ch. 11. Detached Worktrees. Materializing a historical state for inspection without disturbing current work.

Part IV — You Don't Need Every File

Ch. 12. Sparse Checkout. Objects known to the repository $\neq$ paths materialized in the working tree.

Ch. 13. Sparse Does Not Mean Shallow. Limiting working-tree materialization vs. limiting available history, as distinct problems.

Ch. 14. Sparse Index. The performance motivation for scaling Git's own index representation with sparse workspaces.

Part V — You Don't Need Every Object Yet

Ch. 15. Shallow Clone. A Volume IV archaeological exercise fails against limited historical depth; known knowledge is smaller.

Ch. 16. Deepening History. Fetching additional depth on demand; Volume V's "fetch changes knowledge" generalized.

Ch. 17. Partial Clone. Known graph $\neq$ locally materialized object set; deferred object acquisition.

Ch. 18. Promisor Remotes. Missing objects as legitimate when another repository promises to supply them on demand.

Part VI — Large Files Return

Ch. 19. Stable Diffusion Comes Back. The large-model problem returns in this volume's working copy, now under collaborative constraints a local `.gitignore` recipe cannot solve.

Ch. 20. Git LFS. Pointer indirection compared against Volume II's model manifest.

Ch. 36. Rewrite an Entire History. Removing the artifact throughout history in a disposable laboratory; every descendant commit changes identity.

Ch. 37. The Blast Radius of Rewriting. Existing clones, tags, and external references disagreeing with the rewritten graph.

Ch. 38. Preserve the Old History Before Surgery. An archival reference created before rewriting; repair and destruction kept distinct.

Part XI — Hooks

Ch. 39. Git Can Trigger Programs. A harmless `pre-commit` formatting check as the first hook.

Ch. 40. Hooks Are Not Repository History by Default. `.git/hooks/` as local administrative state that must be deliberately versioned to be shared.

Ch. 41. Pre-Commit Checks. Local convenience vs. shared enforcement; a local hook can usually be bypassed.

Ch. 42. Commit Message Hooks. Structured commit conventions motivated by what downstream automation actually needs.

Ch. 43. Server-Side Policy. Client hooks vs. server-side acceptance; extends Volume V's governance model.

Part XII — Configuration as a Layered System

Ch. 44. Which Git Configuration Won? System, global, local, and worktree scopes resolved rather than treated as a flat bag of settings.

Ch. 45. Conditional Includes. Context-sensitive configuration across professional, personal, and textbook repositories.

Ch. 46. Aliases Without Hiding Git. An alias compresses understood reasoning; it does not replace an unexplained model.

Part XIII — Repository Performance

Ch. 47. What Makes a Repository Slow? Scale as a vector over commits, files, blob size, refs, and working-tree size, not one condition.

Ch. 48. Measure Before Optimizing. Benchmarking clone, fetch, status, checkout, log, and diff before enabling any optimization.

Ch. 49. Commit-Graph and Auxiliary Indexes. Derived performance structures that accelerate queries without changing logical history.

Ch. 50. Maintenance. Routine maintenance as preserving efficient physical representation, connecting back to Volume II's loose objects and packs.

Part XIV — Reproducible Environments

Ch. 51. One Commit, Five Environments. Identical commit, potentially different behavior; a commit identifies repository state, not the whole execution universe.

Ch. 52. Version the Recipe. Docker returns from Volume I; the repository progressively captures more of the environmental specification.

Ch. 53. Reproducibility Has a Boundary. Naming what remains outside preserved state rather than asking whether reproducibility is perfect.

Part XV — The Unified Pipeline

Ch. 54. Granite Writes an Intermediate Draft. The research assistant's output consumed by the publishing pipeline; source vs. intermediate vs. final artifact.

Ch. 55. When Generated Output Becomes Source. A generated draft substantially edited by a human; the artifact/recipe boundary as a historical transition, not a filesystem rule.

Ch. 56. Stable Diffusion Produces Publication Art. Versioning the prompt, model, seed, and workflow for a requested illustration; Volume II revisited under real publication needs.

Ch. 57. Scientific Results Enter the Essay. Whether a generated table should be committed alongside its generation recipe; policy by function, not dogma.

Ch. 58. Docker Builds the Whole System. An orchestrated environment running the full corpus-to-PDF pipeline.

Part XVI — Release the System

Ch. 59. One Version Number Is Not Enough. A single monorepo version vs. independent component versions.

Ch. 60. Coordinated Releases. A release as a compatibility constraint set across several component versions.

Ch. 61. Build Once, Publish Many. Separating source history from derived release artifacts; the mature form of Volume II's artifact/recipe distinction.

Part XVII — Architecture Exercises

Ch. 62. Monorepo or Multirepo? Choosing and defending repository boundaries for hypothetical structures; deliberately no universal answer.

Ch. 63. Submodule, Subtree, Package, or Same Tree? Beginning from desired historical semantics rather than feature preference.

Ch. 64. What Should Be Tracked? Designing a tracking and artifact policy for a realistic, thousands-of-files workspace — the grown-up version of Volume I's first `.gitignore` exercise.

Ch. 65. What Must Survive? A repository survivability audit classifying each dependency as Git-preserved, artifact-preserved, externally recoverable, ephemeral, or lost.

Part XVIII — Capstone

Ch. 66. Re-Architect the Entire Series. An intentionally awkward combined repository redesigned by the reader with no canonical command sequence; the resulting decisions are then applied back to the real trunk as the volume's one forward-moving restructuring commit, and must prove themselves through the repository's own subsequent history.

Status and Open Items

Reviewed at the outline level. This volume produced the most significant structural finding in the series so far: its entire subject — submodule extraction, subtree splitting, history rewriting — performs identity-changing operations on material that is the literal shared companion trunk Volumes I–V's tags depend on remaining immutable. The outline above resolves this by treating everything upstream of the capstone as confined to a disposable working copy, with only the capstone's final decision applied back to the real trunk as an ordinary forward-moving commit. One item remains open: Chapter 34 (splitting history back out) currently uses history-filtering machinery that is not formally introduced until Chapters 35–36 in Part X, which comes after it; this needs either reordering or the concept pulled forward.

Volume 7

Git for Agents

Running application: the integrated system inherited from Volume VI

Governing Idea

Distributed collaboration assumed many repositories, but participants who still behaved at roughly human speed. Software agents break that assumption: concurrent, faster-than-human, and only sometimes correct. The book is not primarily about getting agents to write code; it is about maintaining historical integrity under delegated action, formalized as a delegation tuple $D = (B, T, A, \Delta, V, P)$ with the crucial separation $A(\Delta) \not\Rightarrow P(\Delta)$ — producing a change and authorizing it into canonical history are different operations. Volume VI's capstone re-architecture is understood as the one operation actually applied to the real trunk, giving this volume solid ground to inherit.

Chapter Outline

Part I — The Agent Enters the Repository

Ch. 1. Give the Machine a Task. Agent completion is not repository acceptance; `status` and `diff` before any commit.

Ch. 2. The Working Tree Is the Evidence. The agent's self-report vs. the observed working-tree change, generally not equivalent.

Ch. 3. Task Scope. A task formalized as $T = (G, S, C, V)$ — goal, scope, constraints, validation — against a concrete file in the inherited application tree.

Ch. 4. Correct but Out of Scope. A correct fix bundled with several defensible but unrequested changes; correct $\not\Rightarrow$ in-scope.

Ch. 5. Scope Is Historical. A commit containing unrelated improvements asserts a historical cohesion that did not exist in the task.

Part II — Preserve the Human

Ch. 6. The Human Already Has Changes. Uncommitted human work coexists with an agent's separate task; distinguishing Δ_H from Δ_A.

Ch. 7. Never Assume a Dirty Tree Is Yours. An agent must not "clean up" an unfamiliar dirty tree.

Ch. 8. Accidental Destruction. A deliberately bad automation's `reset --hard`, run in isolation, destroys and then recovers unfinished human work.

Ch. 9. Establish a Boundary Before Delegation. Recording repository state before delegation via a branch, worktree, or checkpoint.

Part III — Worktrees for Agents

Ch. 10. Give the Agent Its Own Working Tree. Volume VI's worktrees repurposed as an agent-isolation primitive.

Ch. 11. One Task, One Branch. A branch coordinate for a task, without implying the work is good.

Ch. 12. One Task, One Worktree, One Branch. Combining the previous two into the default isolation model for substantial delegated work.

Ch. 13. Isolation Is Not Acceptance. Isolated $\not\Rightarrow$ valid; sandboxing is not review.

Part IV — The Agent Makes Too Much Change

Ch. 14. The 5,000-Line Diff. Recovering a twenty-line semantic change buried in a mostly-formatting patch.

Ch. 15. Diff Size Is Not Change Size. Lines changed $\not\propto$ importance, in both directions.

Ch. 16. Separate Mechanical and Semantic Changes. Reconstructing the agent's work into independently reviewable commits.

Ch. 17. Preserve or Discard Incidental Improvements? A judgment exercise on an unrelated bug the agent happens to notice.

Part V — Steering

Ch. 18. The Task Changes While It Is Running. An active objective interrupted and redirected mid-task.

Ch. 19. Steering Is Not Restarting. Some prior work remains valid, some irrelevant, some directly conflicting with the new instruction.

Ch. 20. Prompt History Is Operational History. Git history alone does not show why the agent changed direction; instruction history as a second evidentiary layer.

Ch. 21. Don't Commit the Conversation. Useful provenance $\neq$ total surveillance; not every prompt belongs in Git.

Part VI — Interruptions and Continuations

Ch. 22. Stop Here. Inspecting a halted agent's working state for coherence.

Ch. 23. Checkpointing Long Tasks. Intermediate checkpoints intelligible enough to actually continue from.

Ch. 24. Another Agent Continues the Work. Testing whether relevant state survives in durable artifacts rather than one agent's transient context.

Ch. 25. The Handoff Commit. A commit message as an interface constraining safe continuation, not merely describing the past.

Part VII — Multiple Agents

Ch. 26. Two Agents, Different Tasks. Non-overlapping scopes integrated cleanly, the easy case.

Ch. 27. Two Agents Touch the Same File. A parser task and a typing task on the same file within the inherited application tree; ordinary distributed development accelerated.

Ch. 28. Non-Overlapping Diffs Can Still Conflict. A clean merge that nonetheless breaks the program; merge-clean $\not\Rightarrow$ integration-correct.

Ch. 29. Agent Scheduling. A task dependency graph; branches provide isolation but not scheduling.

Ch. 30. Speculative Parallelism. Three agents given the same problem, evaluated rather than all merged; branches as alternative computational trajectories.

Part VIII — Agent Reports

Ch. 31. "All Tests Pass." Reported validation $\neq$ independent validation.

Ch. 32. What Exactly Was Tested? A validation claim is incomplete without its scope and environment.

Ch. 33. Read the Actual CI Result. A green result belongs to an earlier commit than the one the branch now points to.

Part IX — Review Machine-Generated Work

Ch. 34. Review the Patch, Not the Personality. Confidence and tone are not repository evidence.

Ch. 35. Plausible Explanations. A persuasive report paired with a subtly incorrect implementation.

Ch. 36. Review by Invariant. Constructing properties the resulting state must satisfy for changes too large to review line by line.

Ch. 37. Differential Review. Treating the agent as a producer of a transformation $F : S_0 \to S_1$ and reviewing F itself.

Part X — Tests Written by the Same Agent

Ch. 38. The Self-Confirming Patch. An agent's misunderstanding encoded consistently in both implementation and tests.

Ch. 39. Independent Oracles. Existing invariants, golden fixtures, and independently written tests as uncorrelated validation sources.

Ch. 40. Mutation as a Question. Whether agent-written tests actually notice a deliberately altered implementation.

Part XI — CI as a Second Agent

Ch. 41. The Agent–CI Loop. A repeated propose-fail-revise cycle embedding the repository in a feedback system.

Ch. 42. Fix the Failure or Silence the Test? A CI failure "fixed" by weakening the test rather than the implementation.

Ch. 43. Reward Hacking in Miniature. Optimizing for the validator vs. satisfying its purpose, kept concrete rather than sensational.

Part XII — Agent Commits

Ch. 44. Who Is the Author? Separating Git metadata, project attribution convention, intellectual provenance, and legal authorship; Volume IV's distinctions reused.

Ch. 45. Machine Attribution Without Theatre. A minimal provenance convention that helps future investigation without overwhelming the history.

Ch. 46. The Human Commits the Agent's Work. A human-authored commit of reviewed agent output; metadata $\neq$ complete genealogy of ideas.

Part XIII — Agent Branches and Pull Requests

Ch. 47. The Agent Opens a Proposal. A candidate history whose existence does not imply acceptance.

Ch. 48. The Agent Reviews Its Own Work. Correlated self-review compared against an independent review process.

Ch. 49. Review Comments as Steering. A review comment as a new task constraint, unifying Volume V's collaboration with this volume's steering.

Part XIV — Dangerous Git Operations

Ch. 50. Should an Agent Be Allowed to Force Push? Analyzed by scope rather than answered categorically; capability $\neq$ authorization.

Ch. 51. Least Historical Authority. Granting only the repository powers a task actually requires.

Ch. 52. Read-Only Agents. An agent asked only to diagnose, producing analysis rather than a patch.

Ch. 67. Smaller Tasks Produce Reviewable Histories. Task decomposition as a direct determinant of history quality.

Ch. 68. Review Compression. Tests, invariants, and focused diffs compress what a human must inspect directly, without eliminating review.

Ch. 69. The Cost of Saying Yes. Every merged change increases future maintenance obligation; optimizing net useful verified change, not accepted volume.

Part XX — Repository Instructions

Ch. 70. AGENTS.md. Formalizing and expanding the repository-local agent instructions already present in the inherited tree; an instruction file can be incomplete or wrong, and repository reality still wins.

Ch. 71. Instructions Have Scope. Different guidance for different paths in the monorepo, paralleling Volume VI's configuration inheritance.

Ch. 72. Test the Instructions. A fresh agent given only repository-contained guidance, observed where it becomes confused.

Part XXI — The Agent as Maintainer

Ch. 73. Dependency Maintenance. A routine, bounded, autonomous maintenance task with an explicit scope.

Ch. 74. Routine Does Not Mean Risk-Free. A minor update with a dramatic lockfile consequence; frequency must not collapse review quality.

Ch. 75. Release Preparation. Notes, version changes, and manifests prepared by an agent, with the release event itself kept separate.

Ch. 76. The Release Gate. Automation may prepare an irreversible action without automatically executing it.

Part XXII — When Agents Disagree

Ch. 77. Two Reviews, Two Conclusions. Conflicting agent assessments resolved by inspecting evidence, not by voting.

Ch. 78. Consensus Can Be Correlated Error. Agreement among similarly-built evaluators is not independent confirmation.

Ch. 79. Diversity of Verification. Tests, types, static analysis, and human inspection as genuinely different kinds of evidence.

Part XXIII — Autonomous Repository Maintenance

Ch. 80. Give the Agent a Queue. Several approved tasks, each with its own branch and worktree; no invented objectives.

Ch. 81. Candidate Histories. An autonomous agent proposing history without autonomously controlling canonical history.

Ch. 82. Automatic Integration. Low-risk changes meeting a strong predicate integrated automatically; autonomy as graduated, not binary.

Ch. 83. Escalation. Stopping to request human judgment as a successful control path, not a failure.

Part XXIV — The Repository Under Continuous Change

Ch. 84. There Is No Quiet Moment. Continuous concurrent activity ends the idea of inspecting the repository once everyone is finished.

Ch. 85. Optimistic Concurrency. Read, compute, validate, commit — with Git as the coordination substrate.

Ch. 86. Stale Work Is Not Failed Work. A well-implemented patch discarded because another change solved the problem first.

Part XXV — Capstone

Ch. 87. The Autonomous Maintenance Week. A simulated week combining every prior mechanism — steering, handoff, conflicting agents, a subtle regression caught by a second verifier, a scheduled release with a distinct authorization boundary — graded on eight preserved invariants (spanning surviving human work, separable task histories, independent validation, non-stale validation, non-absorbed unrelated changes, respected authorization boundaries, sufficient provenance, and a reconstructable release) rather than volume of accepted change. The reader deletes every agent workspace at the end; the canonical repository must still make sense.

Status and Open Items

Reviewed at the outline level. Three items remain open: the Volume VI/VII boundary needs an explicit statement, given above, that Volume VI's capstone re-architecture is the one operation actually applied to the real trunk, so this volume has real ground to inherit rather than a hypothetical one; the example file paths in Chapters 3 and 27 (`spherepop/validation.py`, `spherepop/parser.py`) do not match the established Volume VI application tree and need correcting to an actual path within it; and Chapter 8's deliberate `git reset --hard` destruction demo needs the same disposable-clone isolation note the specification requires elsewhere (§7.1, §7.3). A minor wording issue also remains: Chapter 70's "AGENTS.md" chapter treats the file as newly introduced, though it already appears in this volume's opening file tree, and should be reframed as formalizing or expanding an existing file rather than introducing one.

Epilogue: What the Commit Became

The Compression

Set beside one another, the seven volumes compress to a single line:

$$\textsc{Record} \to \textsc{Represent} \to \textsc{Transform} \to \textsc{Infer} \to \textsc{Coordinate} \to \textsc{Architect} \to \textsc{Delegate}$$

Volume I asks how a state becomes history. Volume II asks what that history physically consists of. Volume III turns history into a graph that can be deliberately transformed. Volume IV reverses the direction of reasoning and treats the graph as evidence to be interpreted rather than material to be shaped. Volume V makes history distributed across many repositories held by many participants. Volume VI makes the boundary between one history and another an explicit architectural decision rather than an accident of how a project happened to start. Volume VII finally asks who, or what, may propose a change to history, and under what conditions that proposal deserves to become permanent.

This is not merely a convenient mnemonic. It names what actually changes from volume to volume: not the difficulty of the material, but the unit the reader is asked to reason about. A useful way to see this is to watch what a single Git concept — the commit — is asked to be at each stage of the series.

The Commit's Seven Meanings

In Volume I a commit is barely more than a named transition:

$$S_i \longrightarrow S_{i+1}.$$

It is the answer to "what did I want to remember." By Volume II it has become an object with internal structure,

$$C = (T, P, M),$$

a tree, a set of parents, and metadata, addressed by the content it represents rather than by where it happens to sit. By Volume III it is a vertex in a graph,

$$C_i \in V(G),$$

whose meaning depends as much on its edges — its ancestry, its merge relationships — as on its own content. By Volume IV a commit becomes evidence: something to be read for what it can and cannot establish about a past event, with its message treated as testimony rather than fact. By Volume V a commit becomes a unit of exchange, something whose identity must survive intact as

it moves between independently governed repositories, replicated rather than reconstructed. By Volume VI a commit becomes implicated in an architectural boundary: whether it belongs in this history or another is no longer a question the commit itself can answer, only one the repository's design can. And by Volume VII, in a repository where machine activity can produce thousands of internal operations behind a single accepted change, a commit becomes a compression boundary — the deliberately chosen record of a transformation that humans and future machines actually need, out of everything that technically occurred.

None of these seven meanings displaces the one before it. A Volume VII commit is still exactly the small, addressed, graph-situated, evidentiary, exchangeable object it was in Volumes I through VI. The series does not revise the concept; it keeps discovering more of what was already implied by treating a commit as a claim about history in the first place.

Two Symmetries

Two pairs of volumes turn out, in hindsight, to be asking the same question from opposite directions.

Volumes III and IV:

$$\text{Volume III: desired history} \longrightarrow \text{operations}$$

$$\text{Volume IV: observed history} \longrightarrow \text{possible causes.}$$

Volume III gives the reader a target graph and asks which Git operations would construct it. Volume IV gives the reader a damaged, unexplained graph and asks which operations could have produced it. The direction of inference reverses — forward design versus backward reconstruction — but the underlying skill, reasoning correctly about what a given Git operation does to a graph, is identical. A reader who has genuinely learned Volume III's lesson has, without knowing it yet, already learned most of what Volume IV requires; Volume IV mainly asks them to run that reasoning in reverse and to tolerate the cases where it does not fully resolve.

Volumes VI and VII stand in an analogous relationship, one level up:

$$\text{Volume VI: where should the boundaries of history lie?}$$

$$\text{Volume VII: who may alter history within those boundaries?}$$

Volume VI treats repository boundaries as a designed artifact — which projects share history, where a submodule reference is more honest than a merged tree, what must survive if every working machine disappeared tomorrow. Volume VII takes those boundaries as given and asks who, human or machine, is authorized to propose a change inside them, and under what evidence that proposal is allowed to become permanent. Architecture answers where history's walls stand; delegation answers who is allowed to build inside them. Neither question is prior to the other in any deep sense, which is part of why the series can end with Volume VII without implying that repository architecture is a solved problem by that point — it simply becomes the stable ground against which delegation is finally addressed.

The Ending Was Always True

Volume VII's formal model of delegation rests on one separation:

$$A(\Delta) \not\Rightarrow P(\Delta).$$

An acting process producing a transformation does not, by itself, authorize that transformation to enter canonical history. This is presented in Volume VII as a fact about software agents, and it is one. But by the time a reader reaches Volume VII, they have already seen this exact separation under six other names. A contributor's pushed branch in Volume V is $A(\Delta)$ awaiting a maintainer's P. A rejected experimental branch in Volume III is $A(\Delta)$ that never received P at all, and remains preserved precisely because producing it was still worth something. A CI job's green check in Volume V evaluates a narrow, historically indexed predicate — it is evidence offered toward P, never P itself. Even Volume I's very first exercises, asking the reader to inspect `git status` and `git diff` before committing anything, are already teaching this separation in miniature: the working tree can contain a proposed change that the reader has not yet decided is worth keeping.

Agents do not introduce $A(\Delta) \not\Rightarrow P(\Delta)$ into Git. They make it impossible to keep ignoring, because they can produce candidate transformations faster than any human process of P can evaluate them. Volume VII's real subject, stated plainly in its closing chapter, is that once generation becomes cheap, the scarce resources become attention, verification, judgment, and coherent history — and a repository that indiscriminately records every produced transformation has not necessarily become richer for it. It may only have become noisier.

Which returns the series, in its last sentence, to its first lesson. Volume I spends its second chapter teaching that a commit is not a save button — that `git commit` means something closer to *this state matters, remember it* than to an ordinary autosave. Seven volumes and roughly three hundred chapters later, once the reader has watched that same idea survive contact with objects, graphs, evidence, distribution, architecture, and delegated machine labor without ever needing to be revised, only enlarged, the sentence can finally be given its full weight:

A commit is a commitment.

Generation may be autonomous. Commitment should remain deliberate. That is the sentence the whole series was building toward, and it was already true on page one.