MOSTLY DONE.

OR,

A BRIEF TREATISE

ON THE SEVERAL CONDITIONS

UNDER WHICH

SOFTWARE MAY BE DECLARED FINISHED;

OR,

WHY THE AGENT, THE TEST SUITE, THE PULL REQUEST, THE RELEASE WORKFLOW, AND THE ACTUAL APPLICATION ARE RARELY SPEAKING OF THE SAME THING

CLINT ECKER

MOSTLY DONE.

OR,

A BRIEF TREATISE
ON THE SEVERAL CONDITIONS
UNDER WHICH

SOFTWARE MAY BE
DECLARED FINISHED;

OR,

WHY THE AGENT, THE TEST SUITE,
THE PULL REQUEST, THE RELEASE
WORKFLOW, AND THE ACTUAL
APPLICATION ARE RARELY
SPEAKING OF THE SAME THING

BY

CLINT ECKER

CHICAGO, ILL.
MMXXVI

FIRST EDITION

Published in the United States of America
by 2389 Research Press

2000 W. Fulton St., Unit F-314
Chicago, Ill. 60612
+1 (814) 666-2389

hello@2389.ai clint@2389.ai https://2389.ai

Library of Congress Control Number: [LCCN pending]
ISBN: [ISBN pending]

Printed on paper, electrons, or both.
May your tests be thorough and your merges clean.

It is well known that the solution to too much code is 'more code'.

An Anonymous Engineer,
Probably

Contents

List of plates

Preface: The several conditions

Ask a coding agent whether the work is done and it will say yes. Ask the test suite and it will say the assertions passed. Ask the pull request and it will say the branch was green. Ask the release workflow and it will say the artifacts published. Ask the actual application, running on an actual machine in front of an actual person, and you will sometimes get a different answer than all four.

None of these witnesses is lying. Each is answering a different question. The agent means "I completed the unit I was given." The tests mean "the fields I compare agreed." The pull request means "this branch was compatible with the base it last saw." The release workflow means "my own steps succeeded." The application means whatever it does when a human opens it. Software built with coding agents lives in the gap between these answers, and most of what goes wrong in agentic development is some version of mistaking one witness's testimony for the whole verdict.

Mostly done.

Hence the title. “Mostly done” is the honest, permanent condition of software in motion, not a complaint about laziness. Whether the work is done in general is unanswerable. Which specific conditions of doneness have been established, by which evidence, and which remain open: that you can answer, and this book is a treatise, brief on purpose, about how. How to state the conditions, how to make them executable, how to catch the witnesses disagreeing early, and how to keep a fallible author, human or machine, from declaring the verdict alone.

The argument in one paragraph

Good software built with coding agents does not come primarily from better prompting, a larger context window, or a more obedient model. It comes from

arranging the work so that generation is cheap, mistakes are local, claims are challengeable, and verification is closer than self-deception. The agent can write a great deal of code. The engineering problem is to build a system in which no single witness, machine or human, is allowed to declare the software finished on its own testimony.

What "finished" means here

"Done" is intentionally not reduced to test coverage or release speed. It decomposes into conditions that can each be established or refuted separately:

1. **Semantic correctness.** The software does what its contracts say, including awkward edge cases.
2. **Preservation.** User data, source text, and intended meaning are not silently lost.
3. **Perceptual quality.** Rendered output and interactions satisfy named visual or behavioral rules that type checks cannot cover.
4. **Operational truth.** The shipped artifact, release assets, website, and documentation match the code that was supposedly completed.
5. **Maintainability.** Changes remain local enough that another agent can reason about them without reconstructing the entire system.
6. **Epistemic honesty.** Unknown causes remain unknown until evidence resolves them. Bad news is surfaced rather than narrated away.

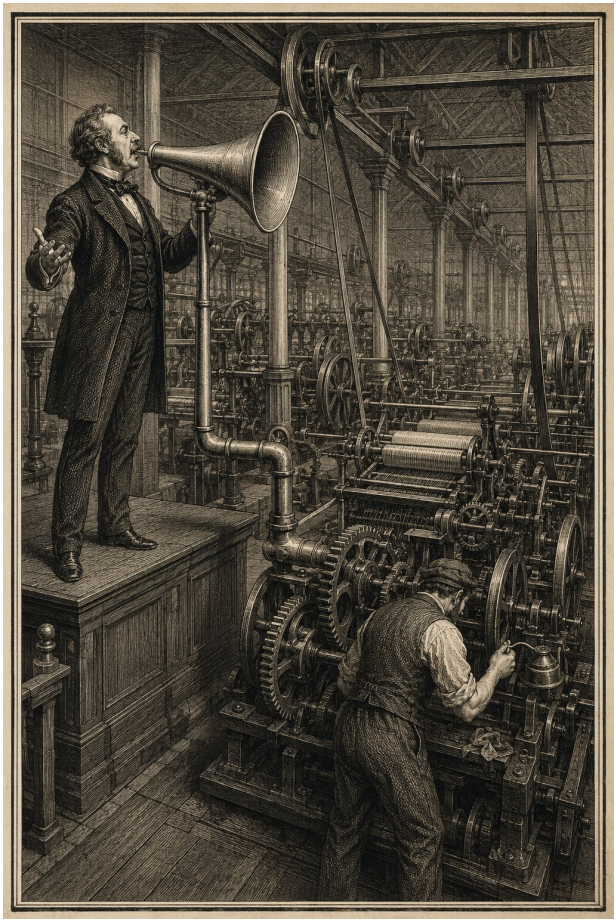
These conditions sometimes conflict. A hard gate can slow a small change. A beautifully factored system can still render something ugly. A reviewer can produce plausible nonsense. The method in this book reserves human judgment for the places that require it and turns everything else into machinery.

How to read it

The early chapters describe the shared engineering loop: boundaries, executable contracts, tested tests, generated proof, design gates, adversarial review, debugging, integration, and memory. Three later chapters show how the loop changes across demanding domains: preserving user data, checking taste by machine, and keeping semantics true across boundaries. The final chapters describe the human-agent division of labor, a practical operating playbook, the failure modes that look productive, and what good actually looks like. Appendix B compresses the whole book into eleven takeaways.

The chapters are short on purpose: not a new software-development religion, just a small set of moves that have repeatedly paid rent.

1 This is not a prompting book



The prompt supplies direction. The machinery supplies everything else.

The easiest explanation for a good agentic patch is that the human found the right words: the prompt was precise, the model smart, the instructions firm. That explanation is attractive because it makes the agent the unit of analysis. It is also too shallow to explain any codebase that has stayed good for months.

The directions that keep long-running agentic work moving are strikingly terse: "go," "keep going," "handle all of it," "build the XYZ feature." Taken alone, those are dangerously underspecified. They work because they land in a repository that already defines what "done" means: which checks must pass, how work is divided, which artifacts must be updated, how branches integrate, and which

decisions need your judgment. The prompt supplies direction, and the repository supplies detail. A two-word instruction is safe when it activates a thousand lines of machinery.

Terse delegation does not buy sustained autonomy, though. However wide its permissions, an agent tends to finish one bounded unit of work and stop to ask for inspection. What keeps execution moving is a completion harness: tests, workflows, review loops, and completion criteria that define done and make premature stopping difficult. Delegation grants permission. The harness supplies persistence.

1.1 The unit of quality is the loop

A coding agent sits inside a loop:

1. receive direction;
2. inspect the current state;
3. form a model of the problem;
4. design or choose a change;
5. implement it;
6. run checks;
7. report what happened;
8. integrate, release, or revise.

Every step can lie.

Your request may omit a constraint. Your repository may hold stale documentation. The agent's model of the problem may be wrong. A test may pass for the wrong reason. The branch may be green while `main` is red. The report may say "done" while the issue stays open or the site serves last week's release. Integration may combine two locally correct changes into a globally broken tree.

That is the book's title in mechanical form. The agent, the test suite, the pull request, the release workflow, and the running application each certify a different fragment of done, and every fragment can be true while the whole is false.

A better prompt improves one step. A better system makes every step easier to challenge. That is why the machinery worth building emphasizes executable invariants, design review, independent validators, real-binary reproductions, full-suite runs after each merge, and live-release checks. The method assumes that any single intelligence, human or machine, will sometimes be wrong. Reliability comes from forcing different representations of the work to agree:

- the design agrees with the requested behavior;
- the test fails on the known-bad implementation;
- the implementation passes the test;
- the integrated tree passes the full suite;
- the built artifact exhibits the intended behavior;
- the documentation and examples are regenerated from that artifact;

- the release system exposes the same version users receive.

Each agreement is an independent chance to catch a fiction.

1.2 Throughput is not velocity

Agents produce code fast, and that invites a measurement error: more diffs, more branches, more reviews, and more issue movement all look like speed. Real velocity is the rate at which trusted capability accumulates. Trusted capability has a different shape from raw output. It includes tests that defend yesterday's hard-won behavior, boundaries that make tomorrow's change smaller, diagnostic tools that shorten the next investigation, and documents that keep the next agent from rediscovering the same trap. A day that lands no feature but adds a faithful golden harness can buy more future speed than a day that lands five unguarded features.

An early ordering worth adopting: safety nets first, then correctness, then structure, then features. It feels backward only if code generation is the scarce resource. With agents it rarely is. What stays scarce is confidence in what has landed, attention at integration time, and your judgment about what the product should be. Build the workflow around those.

1.3 The agent is not the process owner

Agents are valuable because they can carry broad execution: reading, editing, testing, documenting, triaging feedback, preparing releases. Broad execution is not authority over truth. Empower the agent to act and forbid it to self-certify. It can say what it did. It cannot make that statement true by saying it confidently; git state, test output, generated artifacts, independent review, and the live system decide.

You are under the same rule. "Looks good to me" is weak evidence for a rendering engine, and memory is weak evidence for a performance claim. A familiar architecture proves nothing about whether a new interaction is safe. The method does not swap human trust for machine trust. It replaces personal trust with inspectable contracts wherever a contract can be written.

1.4 A compact formula

Agentic software quality can be approximated as:

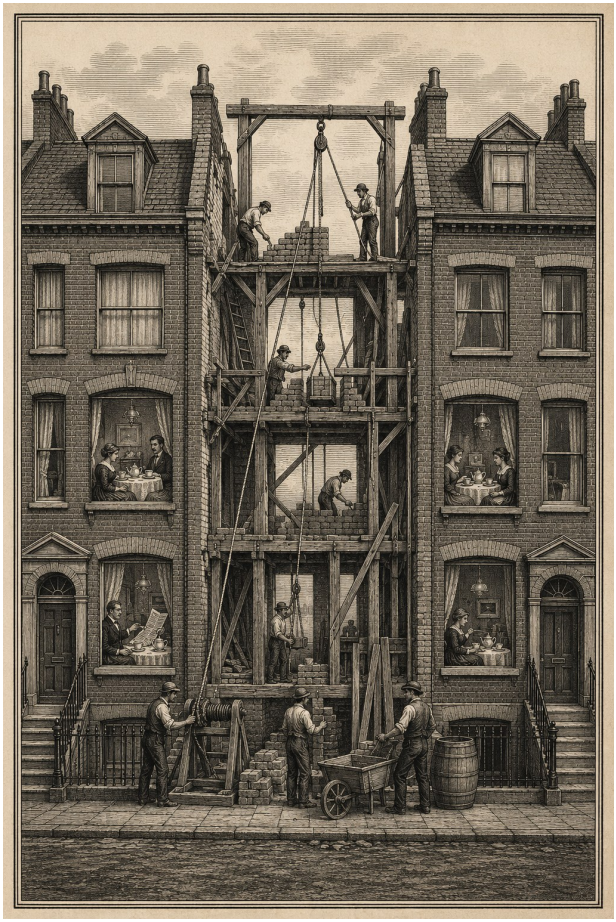
```
trusted progress
  = generated change
  x locality
  x detectability
```

```
x independent verification
x integration discipline
```

This is not a literal equation. It is a warning about multiplication. If a change is hard to localize, failures are hard to detect, reviews repeat the author's assumptions, or integration is casual, then abundant generation amplifies uncertainty rather than capability.

The rest of this book is about raising those multipliers.

2 Architecture localizes change



A good boundary lets the neighbors keep their tea.

The best agent workflow cannot rescue an architecture that makes every change global. Agents edit code fast, but they are not exempt from coupling. Architecture is the mechanism that makes delegation safe.

2.1 One truth, many projections

Consider an editor whose defining decision is that the Markdown source and its syntax tree are the only truth. The screen is a projection. The rich view is not an

independent data model that later attempts to round-trip back into the source.

That boundary removes an entire family of ambiguity. When the view and the source disagree, there is no negotiation over which one wins. The view is wrong. When a projection changes, the source should remain byte-faithful. When the caret moves, its relationship to the source should remain stable. The architectural choice turns philosophical questions about "document state" into concrete invariants.

This matters disproportionately for coding agents. An agent entering a WYSIWYG codebase with two semi-authoritative models must reconstruct a large web of synchronization behavior before it can predict a local edit. An agent entering the one-truth editor can reason from a single direction of authority:

```
source + AST -> projection -> view
```

The architecture narrows the proof burden.

The rule holds because ordinary code cannot bypass it. Every keystroke passes through a single interception point, where it becomes a relative source edit routed into the document session; the text view never commits the mutation itself. An architectural principle deserves trust in proportion to how hard the code makes it to cheat. If any handler can quietly write to the view model, the principle is a preference. If every edit must cross one narrow, observable boundary, the principle is a mechanism.

2.2 Frontend, intermediate representation, backend

Take a diagram renderer built on a related split:

```
source language -> parser/frontend -> platform-free IR ->
↪  renderer/backend
```

The intermediate representation is the leverage point. Adding a new export format to that renderer took one implementation file plus tests, because the scene representation already carried everything the format required. Adding a new input language for entity-relationship diagrams crossed four surfaces: the parser, a key field in the semantic model, one layout line for badge width, and a badge-rendering block. It could not stay in one layer because it added meaning, key badges, that the existing representations lacked.

That contrast is the claim at its true size. A boundary is cheap to extend only when the representations crossing it already carry the needed meaning. The architecture changes the economics of ideas, but the discount applies to changes the model anticipated. Measure leverage by counting the layers a change must touch, and remember that the count is a property of the change as much as of the architecture.

The same caution applies to "platform free." A typesetting engine can keep its layout layer free of platform imports, which buys headless testing on Linux, yet

the same input can still produce different geometry on different platforms, forcing a separate set of golden files for each. A portable core buys headless verification. It does not buy platform-identical output.

2.3 Locality is a verification feature

Software architecture is often discussed in terms of comprehensibility or reuse. With agents, another property becomes central: **how much of the system must be trusted for a small change to be believed?**

A local change has a local proof:

- a parser change can be checked against parse fixtures and IR output;
- a geometry change can be checked against numeric invariants and golden images;
- a renderer change can be checked without questioning source reconstruction;
- a language-surface change can be swept across docs, grammars, and examples without altering the runtime engine.

A cross-cutting change has a global proof burden. It requires more context and creates more chances for two agents to edit the same shared boundary differently.

So if you have six agent branches in flight and a whole-codebase language-mode migration waiting, do the migration alone. The mode flip is not necessarily a bad change. It is a bad concurrent unit. It destroys locality and turns otherwise independent branches into a merge and semantics experiment.

2.4 Design boundaries for agents, not around them

It is tempting to create "agent-friendly" code by adding more comments or smaller files. Those can help, but the more important move is to create boundaries where correctness can be observed.

A useful boundary has three properties:

1. **A narrow input and output.** The agent can state what crosses the boundary.
2. **A stable contract.** The expected behavior is less volatile than the implementation.
3. **An independent probe.** Tests or tools can inspect the result without exercising the entire application.

A platform-free layout layer satisfies all three. So does a projection boundary in an editor whose source is the only truth. Either way, a fresh agent can take a bounded problem and return a bounded artifact.

2.5 Dogfood through real package boundaries

The strongest modularity claim is one enforced socially and mechanically at once: make your host application consume its own engines as versioned packages from a public registry, the way an external consumer would. This is more rigorous than keeping the engines as internal folders and calling the arrangement modular.

A real package boundary forces several truths:

- public APIs must be sufficient;
- versions must be pinned;
- engine CI must own engine correctness;
- the host must prove compatibility with the released package, not an uncommitted sibling checkout;
- upgrading becomes an explicit pin change followed by an integration run.

The dogfood path is a contract test for the architecture. It catches the difference between "separable in principle" and "actually separable in use."

2.6 The architectural question to ask

Before assigning a feature to an agent, ask:

> *Can this change be expressed as one new thing crossing one existing boundary?*

When the answer is yes, implementation can often be broad and autonomous. When the answer is no, the work needs a design step first: create the boundary, then delegate. That is how architecture converts a large creative request into a small proof obligation.

3 Turn principles into executable contracts

A repository can contain excellent prose about correctness and still ship violations of every sentence. A principle becomes operational only when breaking it produces a failure that stops the work. Your codebase is strongest where its most important beliefs have names in the test suite.

3.1 Name the thing that must remain true

Consider an editor whose architecture document declares the source text authoritative. The claim gets teeth from named suites: one defends byte-lossless projection transitions, one defends a caret-line viewport invariant, one requires patch rendering and full rendering to agree.

The names matter. They give future agents a vocabulary for the architecture. "Do not move the caret's line when the projection changes" is easier to preserve as a named invariant than as an emergent property spread across event handlers.

Rendered output takes the same treatment. A diagram engine can convert visual complaints into named geometry rules such as `label-on-fixture`, `label-crowds-edge`, and `edges-doubled`. "These two labels are crowding each other" becomes a regression test named after the observed defect, plus a linter rule that can detect similar geometry elsewhere. Taste becomes executable.

3.2 Ratchets, not snapshots

A good regression test is a ratchet. It lets quality improve and prevents a known failure from returning.

For a rendering defect, the strongest pattern has two layers: fix the generator so it no longer produces the bad geometry, then add a linter or invariant that rejects the same class of bad geometry if another generator produces it later. The first layer corrects the present example. The second protects the concept.

A diagram engine repaired label crowding this way. The generator reserves a 14-point label stub on each side of an edge; the linter rejects any scene where the stub falls below 10 points. The gap is deliberate. The generator aims for comfort while the ratchet forbids only the known failure. A hostile fixture reconstructs the original crowded scene and confirms the linter still catches it.

The distinction matters because agents are excellent at satisfying a narrow failing test. If the test encodes only the exact fixture, an agent can produce a local patch that leaves the underlying family of defects untouched. A named invariant changes the optimization target.

3.3 Contracts should compare the fields that matter

An executable contract creates false confidence when its comparison is incomplete.

Take an editor with an incremental parser whose equivalence helper reparses the document source from scratch and compares the incremental result field by field: blocks, outline, footnotes, stats, source hash, and review metadata. The last field earned its place. The helper originally omitted it, so a fast path could leave a suggestion mark's absolute byte range unshifted while the whole suite stayed green. A real semantic difference passed as equality until the comparison named the field.

The repair went beyond adding the assertion. A companion test records the safety decision that typing near a live mark abandons the fast path entirely, because the block-shifting optimization cannot update an absolute range. If the fast path runs anyway, two comparisons now fail instead of zero.

An equivalence test proves only that the fields it names agree. Derive the contract from semantics, not convenience, and review the comparison helper as part of your trust boundary.

3.4 Guardrails should be hard enough to lean on

The strongest repositories treat pre-commit gates as ground truth. Complexity ceilings, pinned linters, tests, and race checks are not suggestions. The work changes until it passes, and `--no-verify` is not a route around the system.

That hardness creates leverage. You can delegate broadly because the repository rejects certain classes of debt automatically. An agent can refactor a high-complexity function during an unrelated change because the gate makes the debt visible at the moment it becomes relevant.

Hard guardrails also create friction. Suppose a complexity gate with no baseline blocks the same class of edit several times. The correct response is neither blind obedience nor bypass: respect the gate for the immediate work, and record the repeated papercut as a defect in the guardrail itself.

A guardrail that cannot be criticized becomes ritual. A guardrail that can be bypassed casually becomes theater. Obey it now; improve it deliberately.

3.5 Make extension obligations explicit

Consider an editor built from typed blocks and guarded by two invariant suites. Its most useful repository rule fits in one line: when adding a block type, extend both suites. The rule turns architecture into a checklist at the point of change. The agent does not need to infer every hidden obligation from the whole codebase; the repository says which contracts define completeness.

You can write the same kind of obligation for your own domains:

- a new syntax feature updates parser tests, formatter tests, docs, examples, and editor grammar;
- a new renderer updates backend conformance tests and golden fixtures;
- a new workflow transition updates checkpoint, audit, and resume tests;
- a new public API updates package-level compatibility fixtures.

Keep the list short enough to remember and strict enough to catch drift.

3.6 A contract hierarchy

Not every property deserves the same kind of check. A useful hierarchy:

1. type and schema contracts for shape;
2. unit invariants for local semantics;
3. property and differential tests for broad input spaces;
4. integration tests for boundaries between components;
5. golden or perceptual checks for rendered output;
6. release checks for the artifact users actually receive.

Agents can satisfy the lower levels while missing the higher ones, which is why agentic work needs all six. A parser can type-check and pass its unit fixtures yet disagree with a reference implementation on escaped input. A branch can pass every test while the merged tree fails, and a release workflow can succeed while the live site serves an old version.

The goal is a short, named path from each important claim to the nearest executable contradiction.

4 A test that has never failed is a rumor

A green test is evidence only if it can turn red for the defect it claims to guard. That sounds obvious, and it is among the most commonly skipped steps in agentic work, because an agent often generates the implementation and the test in the same pass. The two can share one misunderstanding. They agree beautifully and are wrong together.

4.1 Revert the fix

The simplest discipline is the strongest:

1. reproduce the bug;
2. write the regression test;
3. apply the fix;
4. observe the test pass;
5. revert or disable the fix;
6. observe the test fail for the expected reason;
7. restore the fix and run the relevant suite again.

The red run can come from either direction: stash the fix and watch the new test fail, or run the test against the buggy code before the fix lands. Consider a leak fix in file-watcher teardown. Run against the buggy code, the regression test created 150 watchers and caught the process holding 153 descriptors against a baseline of 23:

```
XCTAssertLessThanOrEqual failed: ("153") is greater than ("23")
```

That failure is what made the later green mean something.

One qualification. Demonstrations like this are usually manual: you stash the revert by hand, and the red run survives only as a note in a PR description. Unless CI enforces mutation checks, nothing proves the test can still fail today. “Was shown to detect the defect once” is weaker than “is continuously shown to detect it.” A recorded red run is evidence with a date on it.

The deliberate red run catches mistakes a green run hides:

- the fixture does not reach the buggy path;
- the assertion observes the wrong output;
- the test environment masks the behavior;
- the implementation was already correct for the fixture;
- the test helper normalizes away the difference;
- the new code and the test share the same faulty assumption.

4.2 Test the test harness with deliberate damage

Consider a golden-image harness for a math renderer, built to catch blank or wrong output by computing an ink signature over each rendered equation. To prove the harness worked, its author zeroed one equation's golden image, exactly the corruption it existed to catch. The test passed.

The signature averaged darkness across cells. Thin glyph strokes diluted toward white, and quantization erased what remained. The harness built to catch "broken render passes silently" passed broken renders silently. After the repair, the same tamper drifted 18 of 256 cells, 7.03 percent against a 2 percent limit. The zeroed golden finally failed loudly.

A second flaw surfaced later: the signature read only the red channel, so a red glyph could register as blank while sitting visibly on the page. Independent review found the blind spot. The fix, measuring ink by the darkest channel instead of the red one, was checked from both sides: a colored render's inked cell count rose from 39 to 48 while the uncolored controls did not move. The correction changed what it should have changed and nothing else.

The general rule:

> *Every new verifier should ship with a known-bad probe that it rejects.*

- a golden-image harness runs against an intentionally altered image;
- a leak detector runs against a fixture that leaks;
- an equivalence checker compares scenes that differ in each important field;
- a release check points at a missing asset and fails;
- a linter rule includes a fixture that violates it;
- a migration validator processes input known to be incompatible.

The probe can stay in the suite as a permanent test of the verifier, or live once in the change record. Keeping it is usually better.

4.3 Prefer differential and property tests for parsers

Suppose your agent hands you an escape-aware condition parser along with a dozen passing examples. Do not stop there. Write a slow, simple reference decoder and brute-force tens of thousands of generated inputs against it:

```
for input in generated_inputs:
    assert optimized_decode(input) == reference_decode(input)
```

Differential testing pays off most on agent-generated parsing code. A parser packs compact logic over a large behavioral surface, and a model can produce a plausible state machine that handles every example in the prompt and fails on combinations nobody thought to write by hand.

The reference need not be production quality. It exists only for comparison. Property tests add round trips, preservation, idempotence, and monotonicity. The

important move is generating disagreement opportunities beyond the author's examples.

4.4 Watch for tests that prove their implementation

A subtle failure occurs when the test reconstructs expected behavior with the same algorithm as the code under test. If both compute a midpoint by subtracting arrow lengths incorrectly, the test proves consistency, not correctness.

Stronger oracles come from a different representation:

- place a renderer's output beside an established independent engine, and name what the comparison can and cannot decide;
- compare a parser to a deliberately simple reference decoder;
- compare a screenshot crop to an on-screen calibration rather than to another capture path using the same graphics API;
- compare patch rendering to a full render built through a different route;
- compare release metadata to the live served version.

Independence is not binary, but every distinct representation reduces shared blind spots.

4.5 Failure quality matters

A test that fails noisily for many unrelated reasons is hard to trust. The deliberate red run should fail on the assertion connected to the defect, not because the fixture crashes or the build no longer compiles.

This is why reproduction comes before the fix. It establishes the observable symptom and the shortest path to it. The regression test should preserve that path.

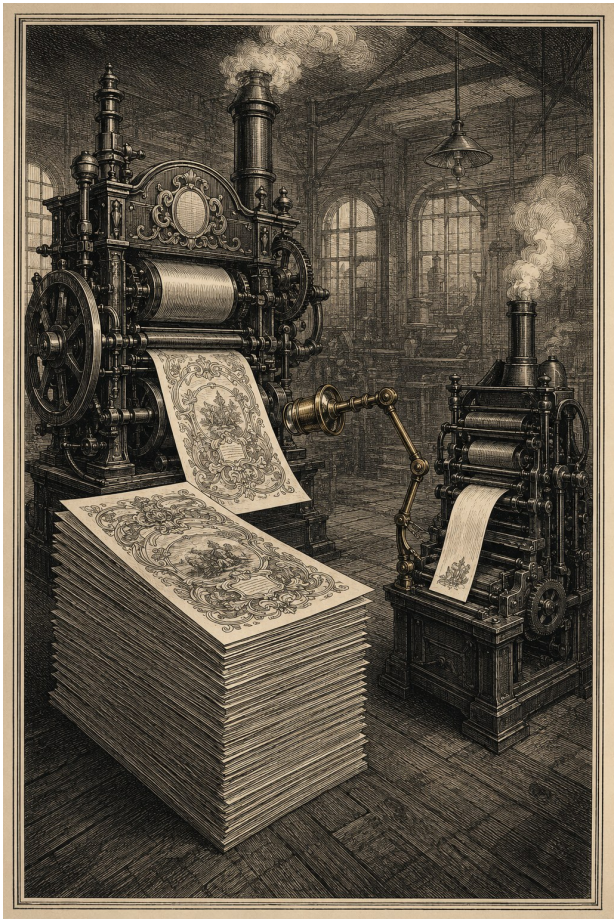
A useful test report answers:

- What known-bad behavior was introduced or restored?
- Which assertion failed?
- Did it fail for the expected reason?
- Which fixed behavior made it pass?
- Which surrounding suite was run afterward?

That record matters most when the author is an agent and you never watched the test being developed.

The common slogan says do not trust the model. It stops short. A test carries no authority from its author, human or agent, and none from months of green. Trust a check in proportion to the evidence that it detects what it claims to detect.

5 Generate the proof



Provenance is not verification.

Documentation, screenshots, benchmarks, and marketing claims drift away from the software that made them true. Agentic development accelerates the drift because it accelerates both code and prose. An agent will update a README with the same confidence it uses to update an implementation, even when the measurement behind the README is stale or misunderstood.

If a claim can be generated, do not type it.

5.1 Make the marketing a test

Consider a math typesetting engine whose gallery figures regenerate from the live engine on every push. Six generators run in CI and publish the results, so a broken gallery image means either the engine changed or the example no longer means what the text says. A public claim has become an integration check.

Watch how easily that claim inflates. The engine's announcement essay claimed every documentation figure was regenerated. On inspection, the two most persuasive images, the wordmark and a side-by-side comparison against an established engine, were committed static assets. Each had a reproducible generation script, but nothing in the build regenerated or validated them. The website's scoped sentence, "every image below," was accurate for the gallery. The essay's global sentence was false, and the gap fell exactly where the claim carried the most weight.

The distinction is operational. A generated artifact has provenance: the system it advertises really did produce it once. A verified artifact is regenerated or checked by the current build. The wordmark has provenance. The gallery has verification. Only the second kind can catch a regression.

The side-by-side comparison deserves the same narrowing. It renders the same input through both engines at matched height, with no automated diff and no acceptable-difference tolerance. At equal height, one example's aspect ratio measures 3.64 in the reference engine and 2.94 in the product: visibly close, not identical. The site's phrase "Judge the fidelity yourself" is honest, because visual judgment is exactly what a side-by-side supports. It is not an equivalence test. Name the evidence you have.

Now consider a language toolchain whose machine-consumed specification is assembled by a script and embedded in the binary. A freshness test regenerates the spec into a temporary copy and fails if the checked-in file is stale, and every example in the repository is parsed by the real parser and linter in tests. That artifact does not merely have provenance. The build proves it is current on every run.

The technique generalizes:

- generate CLI help from the parser's command schema;
- generate language references from accepted grammar fixtures;
- generate API examples as executable tests;
- generate compatibility tables from CI results;
- generate performance tables from committed benchmark scripts;
- generate screenshots from a deterministic demo project;
- generate release notes from merged, labeled changes and then review the prose.

The point is to eliminate hand-copied representations of facts the repository can derive.

5.2 Believe the measurement over the README

Consider a renderer whose README describes a cold render of about 0.3 ms covering parse through rasterization. A phase profiler, built to check that every microsecond of the path is understood, contradicts the claim. On macOS, the measured call returns a lazy image; rasterization happens later, on first paint. The number also came from a debug build, which the profiler shows runs the renderer's own phases two to five times slower than release. The corrected figures separate what the old claim merged: a warm render around 0.5 microseconds, a cold fill around 0.2 ms, the deferred paint around 0.23 ms, and a headless render around 8 microseconds.

The corrected numbers matter less than the corrected model. Fix the README, commit the profiler, and the next performance claim can be regenerated instead of remembered.

The pattern is worth keeping:

1. state the question in measurable phases;
2. build the smallest instrument that separates them;
3. run it in the environments whose behavior may differ;
4. update the claim to match the measurement;
5. keep the instrument close enough that the claim can be re-derived.

5.3 Generated does not mean trusted

Generation removes one class of drift and introduces another possibility: the generator can be wrong.

A screenshot pipeline can mirror an image, crop the wrong region, or capture a lazy placeholder. A benchmark can omit the expensive deferred work. A compatibility script can mistake "workflow completed" for "artifact published." A documentation generator can preserve stale metadata faithfully.

So damage the proof system on purpose and confirm it objects:

- alter a generated figure and confirm the check detects it;
- make the expected version deliberately wrong and confirm the live-site check fails;
- insert an unsupported syntax example and confirm the language reference marks it unsupported;
- delay rasterization and confirm the profiler attributes work to the correct phase;
- remove a release asset and confirm the release verification rejects the release.

Generated proof beats a typed assertion only when its source and verifier are inspectable.

5.4 Documentation should lose arguments with reality

It is tempting to treat documentation as a promise the code must be made to satisfy, even when the promise was written from a mistaken understanding. That temptation produces contorted implementations built to preserve a flattering sentence.

When a measurement disagrees with your README, fix the README unless the product contract truly requires a code change. When a design review proves a shipped headline feature semantically wrong, revert the feature rather than defend the announcement. When your live site is three releases behind, the work is not done, however long ago the merge happened.

Agents and people read documentation and act on it, so a wrong sentence propagates into wrong work. That makes documentation worth maintaining and still not sovereign. It should hold the best current model of the system and provide executable paths back to the facts.

5.5 Proof at each public boundary

A mature agentic project benefits from a proof chain:

```
source -> tests -> built artifact -> generated docs -> release
↪  -> live surface
```

Each arrow should have a check:

- the source builds without warnings you have chosen to ignore;
- the tests exercise the built binary, not an imagined equivalent;
- generated docs use the current binary;
- the release contains the expected assets and version;
- the live site serves that version;
- external consumers can install the published package.

This is why “merged” is not “shipped.” The chain ends where the user meets the product.

Treat a public claim like an API: versioned, tested where you can, revised when the evidence changes.

6 Write the disagreement before the code

Coding agents make implementation cheap enough that you are tempted to start there. A request arrives, the agent reads the repository, and a plausible patch appears before anyone has stated the hard decision. This feels fast until the patch gives the wrong model a few hundred lines of momentum.

The remedy is a gate before nontrivial code. Call it a brainstorm, a spec, a design, a plan. The artifact can be a page or a paragraph. Its job is always the same: give yourself something inexpensive to disagree with.

6.1 Find the decision that can invalidate the patch

Consider a workflow design sent through five reviewers, each with a different concern: control flow, checkpoint and resume behavior, audit integrity, simplicity, tests. One reviewer found that a gate which never ran could still be counted as covered. That was a state-model error, not a typo. In the design it cost a few sentences to repair. In code it could have produced a complete implementation, a matching test suite, and a confident completion report built on the same false premise.

Some errors run deeper than a repairable state model. Consider a shipped migration that converted a node's retry destination into a routing edge. A later design investigation established that retry and routing are distinct channels in the runtime that consumes the workflow, so no refinement of the conversion could be correct. The right change was to keep retry and fallback destinations as node fields and revert the premise of a feature that had already shipped. No clever implementation could save the model, because the model itself was the defect.

A useful design gate asks which decision could make the whole patch wrong even if every line is clean. Typical candidates:

- which representation owns the truth;
- where malformed input is rejected;
- whether a fallback preserves or invents user intent;
- how a workflow resumes after a checkpoint;
- whether two runtime channels are actually related;
- what compatibility means across versions;
- which state must be cleared in loops or retries.

These questions belong before implementation. Once code exists, authors become invested in preserving it. That is true for people and for models.

6.2 Scale the design to the risk

Design first does not mean a miniature standards process for every edit. The size of the document should follow the amount of judgment in the change.

A two-line geometry adjustment may need four sentences:

1. the visible defect;
2. the intended geometric rule;
3. the state that must not change;
4. the check that will fail on the old behavior.

A parser feature, persistence change, migration, or new projection model deserves more. It should identify the canonical state, failure behavior, compatibility policy, proof obligations, and rollback boundary.

The gate becomes theater when it rewards document length rather than exposed decisions. It becomes useful when implementation cannot begin until the author has answered the question most likely to invalidate the work.

When the design survives, its trail can stay inspectable. A feature can run from a dated design document through a plan, a series of implementation commits, review fixes, and a tagged release, each step a named artifact you can reopen later. The design yields a durable decision and a decomposition, not one large execution unit. Not every feature needs that much ceremony, but ceremony done this way leaves evidence instead of meeting minutes.

6.3 Remove the gate that cannot change the outcome

The lesson cuts the other way too. Suppose your process includes a standing pause for your own detailed review, while your repository policy already delegates detailed review to independent review agents. Once that policy stands, the pause cannot change the work. It is ceremony. Delete it.

Sort your gates into three kinds:

- a necessary gate settles a choice that changes semantics, scope, risk, or reversibility;
- a delegable gate verifies a preference the repository has already decided, and independent review can hold it;
- a ceremonial gate has a predetermined answer, and the pause cannot change the work.

The distinction is not whether the gate agrees with the recommendation. A gate can wave the recommended choice through and still be working, provided the choice carries a genuine blast-radius tradeoff and a different answer would have redirected the work. Ask whether disagreement was possible and would have mattered. If not, delete the pause.

6.4 Put human attention at the fork

You need not prescribe each helper function or file edit. That would throw away the agent's main advantage. Your attention is better spent on forks that encode product intent, acceptable loss, or cross-project reality.

A good fork might ask:

- Should invalid syntax be rejected during parsing or preserved for a later diagnostic pass?
- Should a composition feature cascade, include, or expand into tokens?
- Should a whole-codebase migration be isolated from the current feature batch?
- Is the fallback allowed to guess, or must it show the original source?

The agent can explain the consequences. You choose the policy. After that, execution can be delegated widely without leaving the important semantic choice implicit.

6.5 Review the design to break it

A design review should not ask whether the proposal sounds reasonable. Almost every flawed design sounds reasonable when summarized by its author. Give each reviewer a specific lens and a request to find a counterexample.

Useful lenses include:

- unreachable and repeated states;
- checkpoint, resume, retry, and rollback behavior;
- malformed and partially valid input;
- byte preservation and round trips;
- concurrency and ownership;
- migration and downgrade paths;
- audit history;
- platform differences;
- weak or shared test oracles;
- accidental complexity.

Disagreement between reviewers is often the valuable part. Put two reviewers on a workflow state model and one may argue that an explicit clear operation is unnecessary while the other shows why looping workflows require it. The conflict forces the state transition into the open, and the decision follows the behavior, not the vote count.

6.6 Keep the design after the merge

Do not discard the design when the code lands. It becomes one of three things:

- a lasting contract that belongs in architecture documentation;

- a short decision record explaining why one model won;
- a temporary plan whose durable lessons move into tests and standing notes about the failure.

The document should not narrate the diff forever. It should preserve the decision that a future agent might otherwise reverse by accident.

The best design gate is modest. It catches a wrong idea while changing the idea is still cheap.

7 Treat review as an adversarial search

Review goes soft when its social purpose is approval. The author wants the merge. The reviewer wants to be helpful. Both read the same explanation, look at the same tests, and inherit the same assumptions. A polite pass over the diff finds the style problems and leaves the state model untouched.

Ask your reviewers to disprove the work instead.

7.1 Fresh context matters

You know why every line is the way it is. That knowledge serves you during implementation and betrays you during self-review: you fill each gap without noticing there was one. A reviewer arriving cold sees the gap.

What cold readers catch is rarely obscure syntax. It is failed assumptions: an approval gate that can never run, two serialization modes that quietly disagree on trailing whitespace, a golden-image signature that ignores the color information its author was sure it covered. Nothing in the diff flags any of these. The path the author walked made the code feel inevitable, and the reviewer never walked that path.

When the reviewer is an agent, you control how cold it arrives. Give it the diff and the contract, not your explanation of why the code is right. An agent that reads your reasoning first inherits your assumptions along with it.

7.2 Give each reviewer a lens

Agents let you convene a review panel at will, and the temptation is to convene five copies of the same reviewer. That buys duplicated confidence. A useful panel divides the attack surface.

For a workflow change, the lenses might be:

- control-flow reachability;
- persistence and resume;
- audit integrity;
- malformed conditions;
- unnecessary state.

For a renderer: layout geometry, backend parity, font and color behavior, validity of screenshots or goldens, platform-specific execution. For a parser or a migration: byte preservation, escaping and malformed input, version skew, downgrade behavior, agreement with downstream consumers.

The lens does not restrict the reviewer. It gives each one a reason to question a premise the author and the rest of the panel share.

7.3 Findings are claims too

Consider a wide review pass: eleven lenses over one codebase, fifty-two candidate findings. Before anything is filed, a second validator rereads the code and tries to refute every finding. Five die. One alleged a dark-mode failure; the validator found the exact template configuration that made the behavior correct. Another claimed a typesetting rule had been inverted; the algorithm's published source showed the implementation was faithful and the reviewer's premise was wrong. The forty-seven survivors become issues worth a maintainer's attention.

None of this discredits review. It argues for completing it. A finding should include:

1. the claimed defect;
2. the code or contract involved;
3. an input or state that should expose it;
4. the likely consequence;
5. evidence that would refute the claim.

A separate pass then accepts, rejects, or defers each finding with a reason. File every plausible concern unfiltered and you degrade the tracker and teach yourself to ignore review output.

7.4 Require repair or reasoned rejection

Review turns into ceremony when a comment can be acknowledged without changing anything. An accepted finding should produce a repair and a check. A rejected finding should preserve the technical reason. A deferred finding should state what evidence or scope is missing.

Picture one round on one change: ten comments come back. Six get fixed. Three turn out to be already resolved, and you verify that rather than waving it through. One is kept with a written caveat because no minimal failing test can be constructed. None are accepted while wrong. Zero wrong acceptances matters as much as six repairs, and both counts come from checking each comment against the code rather than against the reviewer's confidence.

A decline can carry as much engineering as a fix: refusing to let a rendering backend mutate public scene state for its own convenience, or keeping round-to-nearest after a reviewer proposed changing the rounding, because round-to-nearest was correct.

Dispositions with reasons form a ledger rather than a pile of comments. Over time the ledger shows which parts of the system generate recurring confusion, which lenses find real defects, and which tools hallucinate the same domain rule again and again. The rejection record earns its keep twice: it stops future reviewers from reopening a settled false premise, and it proves you do not measure quality by issue count.

7.5 Make bad news useful

Adversarial review works only when the system welcomes unpleasant facts:

- `main` was already red;
- a screenshot diff was encoding noise;
- a reported memory bug was not a memory bug;
- an outside contributor's tests found a gap in the internal fix;
- a performance claim measured the wrong phase;
- a shipped feature should be removed;
- a guardrail was correct and still badly implemented.

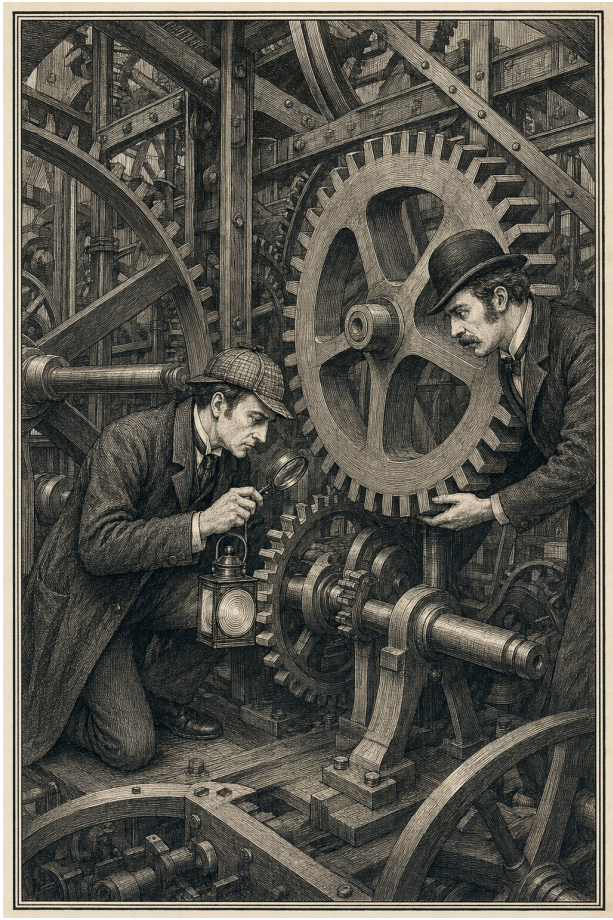
A collaborator punished for reporting facts like these will not produce fewer defects. They will produce fewer reports. Early bad news is cheap. Late reassurance is expensive.

7.6 Review the implementation and its proof separately

The code and its evidence can fail independently. A correct implementation can have a weak test. A strong test can reveal that the implementation still fails on one path. A benchmark can be accurate while its label overstates what it measured.

So ask two questions. Is the behavior correct? And would the supplied proof detect a plausible version of the behavior that is wrong? The second is easy to skip because the test suite looks like evidence. It is still authored work, and it deserves its own attack.

8 Debug without guessing



When the evidence stops, the patch stops.

A coding agent can produce a convincing root-cause story before the logs have finished scrolling. That is useful while you generate hypotheses and dangerous while you repair. The faster the explanation arrives, the more important it is to separate what was observed from what merely fits.

The discipline is a plain sequence: reproduce, trace, measure, change, prove. When the evidence stops, the patch stops too.

8.1 Reproduce against the real system

A bug begins with the built binary, not a description of what the code appears to do. A formatter corrupts valid escaped quotes while its validator accepts an unmatched quote with exit code 0. Before anyone touches the parser, both halves get reproduced against the built binary at a named revision: the exact input, the corrupted output, the exit code. If the complaint is visual, drive the actual rendering path, not the unit that seems responsible.

A reproduction should record:

- the revision;
- the build mode;
- the input or fixture;
- the command or user action;
- the observed output;
- the expected output;
- the smallest artifact needed to repeat it.

This keeps the investigation attached to behavior. It also lets another agent challenge the diagnosis without reconstructing the original session.

8.2 Trust numbers before screenshots

Visual bugs invite visual guessing, and a screenshot pipeline can lie as persuasively as code. Take a visual test harness that reads rendered frames back from the GPU. Its readback path used a bottom-up row order and a bottom-origin crop, and it matched a mirrored image at 92 percent, so a wrong fix passed its visual check against a wrong capture. The replacement went through the real compositor path and added an orientation anchor: a known red-and-blue image whose asymmetry calibrates row order and makes any mirrored band crater the match ratio. The capture path earned trust the same way a test does, by being shown to fail against known-bad input.

When someone says your output looks wrong, move quickly from pixels to a model:

- source offsets;
- line and grapheme positions;
- label boxes and fixture bounds;
- glyph ink bounds;
- crop rectangles;
- color channels;
- deferred versus completed drawing.

The screenshot still matters. It tells you the product looks wrong. The numbers tell the agent where the product's model disagrees with what the human saw.

8.3 Measure the phase named in the claim

Consider a renderer whose README quotes a cold render time from parsing through rasterization. The measured call returned a lazy image; drawing happened later, on first paint. The number also came from a debug build. A phase profiler showed platform text measurement and drawing consuming most of the cold path, while the renderer's own parse and geometry code used much less. Nothing was wrong with measuring image construction. The error was naming it rasterization.

A trustworthy performance investigation separates phases that the runtime may schedule differently:

```
parse -> shape -> layout -> allocate surface -> draw -> present
```

The exact phases vary by system. The rule does not. Instrument the work the claim names, run the relevant build and platform variants, and keep the profiler so the number can be regenerated.

When the measurement disagrees with the documentation, change the documentation unless the discrepancy exposes a real product contract.

8.4 Do not use "flaky" as a cause

Consider a CI matrix whose Linux job fails intermittently and wears the flaky label until someone refuses it. The refusal turns up an emulation layer faulting the compiler under a foreign architecture. A native build removes the failure, and the rule goes into the platform documentation, where later contributors inherit the diagnosis instead of rediscovering the symptom.

"Flaky" describes the investigator's view, not the mechanism. A better report states:

- how often the failure occurs;
- which environments show it;
- whether retries change the result;
- whether ordering, timing, load, or resource pressure correlate;
- which hypotheses have been ruled out;
- which traces are still missing.

"Not yet understood" preserves the obligation to keep looking. "Flaky" often closes the issue without an explanation.

8.5 Stop when the missing fact is outside the evidence

Suppose your agent is chasing a patch that comes out empty. It rules out the obvious extraction and path theories, and the remaining answer depends on a diagnostic artifact nobody has. The right result is a request for that artifact, not a

plausible code change.

A mature workflow permits these outcomes:

- reproduced, not localized;
- localized, root cause unproven;
- likely cause, missing artifact;
- report refuted;
- expected behavior;
- no safe patch recommended.

Agents are biased toward producing a diff. Humans are often biased toward rewarding one. Both need an explicit escape from speculative progress.

8.6 Preserve unsupported input

Debugging discipline shows in fallback behavior too. Picture a math renderer handed `a_b_c`, an invalid double subscript, which it displays as `a_c`. The engine silently discarded content. The repair preserved the original source and displayed a diagnostic that maps to a useful source range. It did not guess that the user meant `a_{bc}`.

The same ethic applies beyond parsers. A migration should preserve the record it cannot convert, and a workflow engine should expose the state it cannot resume. An editor keeps the bytes it cannot interpret. Degrade so the user's input stays visible and locatable; silent disappearance is not graceful.

The honest system says what it could not do and keeps the evidence needed to do better later.

9 Keep changes small and reversible

Coding agents lower the cost of parallel implementation. They do not lower the cost of deciding which integrated revision deserves trust. Fan the work out across branches and worktrees, then pull it back through a narrow integration path. Production can happen in parallel. Shared truth is established one revision at a time.

9.1 One decision per branch

A branch should represent one coherent claim. That keeps the diff reviewable, the failure local, and the revert meaningful.

In practice the rule looks unglamorous. Keep the new language frontend off the renderer branch. Give each issue its own branch. Hold pixel-changing work until its safety net exists. When a complexity gate refuses to let a feature hide inside an oversized entry function, decompose the function first, on its own branch.

The point is not branch purity. The point is causality. When a branch mixes a parser, a renderer, a migration, and a structural rewrite, a red test has too many parents and a revert removes too much.

A branch boundary holds when it has:

- one user-visible or architectural decision;
- one set of proof obligations;
- no unrelated cleanup unless the gate forces it;
- documentation changes that belong to the same contract;
- an independent rollback path.

Branch discipline covers other people's branches too. Consider an outside contributor's fix that has been open for half a day when an internal duplicate lands by fast-forward, putting the contributor's branch into conflict. The contributor's tests then find a real defect in the merged duplicate: quote termination that ignores escape parity. The right repair is to fix the gap, salvage the unique test cases with credit, and apologize. Repository state includes open pull requests. Conflicting code can be redundant while its tests contain unique evidence.

9.2 Isolate implementation with worktrees

Separate worktrees give each agent a stable checkout and reduce accidental overlap. They also preserve reasoning: a reviewer can see which assumptions belonged to the branch before later changes blurred them.

Isolation alone is not enough. Each worktree should receive the same standing rules: current base, commit discipline, local gates, required documentation, critical review, and completion bundle. Set the rules once so the conversation can stay

terse.

The agent should report the exact revision it tested. "Green in my worktree" means something only when you know the worktree's base and commit.

9.3 Merge one change, then test the new tree

Feature branches establish local facts. Integration creates a new object.

Take two pull requests, both green on their own branches: one introduces new language frontends, the other an edge-label linter. After both merge, a lint-cleanliness test fails on `main`. A fixture from the first branch left a 7-point label stub against the second branch's 10-point minimum. The fixture and the rule first shared a tree at the merge. Neither branch was wrong. The combination was.

This suggests a strict sequence:

1. refresh the branch against current `main`;
2. rerun its targeted and full checks;
3. merge one coherent change;
4. run the full suite on the resulting tree;
5. inspect generated and public artifacts;
6. continue only after the integrated revision is understood.

A merge queue can automate the mechanics. It cannot change the fact that green belongs to a specific tree.

9.4 Build the net before changing the pixels

Order the work: safety nets, then correctness, then structure, then features. Land the CI graph and the render harness before the fix they are meant to protect. Make pixel changes wait for a pixel detector that has itself survived tampering. Let structural cleanup follow the correctness repair instead of riding along as an unreviewable rewrite.

This order looks slow because the feature you want appears last. The net turns each later edit into a smaller risk, and that is where the time comes back.

9.5 Reversibility requires more than Git

Any commit can be reverted mechanically. A product change is reversible only when its data, compatibility, and public surfaces can also move backward without guesswork.

For language and workflow changes, ask:

- Can old data still be read?
- Can new data be downgraded or rejected visibly?
- Does the migration record enough information to undo itself?

- Can the feature be removed without leaving editor grammars or docs behind?
- Will downstream consumers receive a clear version boundary?
- Does the PR mix a policy decision with unrelated cleanup?

When runtime reality disproves a shipped feature, you want the reversal to be ordinary: no sunk cost to defend, no concept to salvage. That is only possible when the work was bounded in the first place.

9.6 Completion reaches the user

A merged PR is not a release, and a green release workflow proves less than it appears to. Consider a project whose release automation and required checks live in separate workflows. The publish workflow stays green and ships two releases while a required CI step, a migration parity check, has been red for weeks. Nothing connects the green workflow to the red one, so the green one publishes. The rule that survives the repair is blunt: verify the whole CI run, not one job.

The completion path should be explicit:

```
source -> integrated tests -> whole required CI graph -> built
↪  artifact -> published release -> live surface
```

Be as skeptical of regeneration as of any other publishing step. A gallery regenerates in CI on every push. One run marks 62 images dirty; a pixel comparison at 2 percent fuzz finds 52 of them byte-different but visually identical, and exactly the ten meaningful changes are committed. Regeneration reports rewritten artifacts. Pixel comparison reports visual changes.

Small increments help here too. A release containing one coherent change is easier to inspect, explain, and roll back.

Agents make it cheap to create branches. Engineering still happens where those branches become one product.

10 Make memory durable

The context window is temporary. The codebase has to remember for it. An agent can read a repository quickly and still repeat a mistake you solved six months ago.

Good project memory does not record every event. It preserves the failure classes that would otherwise return.

10.1 Write down scars

The most useful agent instructions read like scar tissue: each rule traces to a shipped bug. Take a text editor built largely by agents. Its instruction file warns that carriage return and line feed behavior matters at grapheme boundaries, that per-glyph backgrounds produce ugly strips, that two recognizers for one grammar will drift, that equivalence tests compare only the fields they name. Each warning explains the trap, not only the preferred helper.

Compare two instructions:

> *Use the unified matcher.*

and:

> *Use the unified matcher because two recognizers for the same grammar diverged in production.*

The second survives refactoring. A future agent may replace the helper, but it knows the system must keep one grammar implementation.

The best scars sit in the code they protect. Picture a comment in an editor's styling code: the line-prefix loop must advance by the full line range, because the old clamped-location version re-derived the same final line forever once the scan reached the end of the text, hanging the UI with a pinwheel. Four facts in a few lines: the rule, the tempting wrong approach, the shipped symptom, and the location. That is the anatomy of a scar worth keeping.

Other scars worth carrying share the shape:

- never bypass the pre-commit gate;
- check for an existing pull request before starting overlapping work;
- verify the whole CI run, not only the release workflow;
- test golden-image detectors with deliberate corruption;
- preserve malformed source instead of guessing user intent.

A scar should be brief enough to read and specific enough to alter behavior.

10.2 Give documents authority

Documentation turns dangerous when several files describe overlapping truths without saying which one wins. A fresh agent will quote the wrong document with perfect confidence.

A useful document map identifies:

- the source of product behavior;
- the architecture description;
- the current execution or handoff plan;
- the invariant and scar list;
- the release and compatibility policy;
- the conflict rule when two sources disagree.

Treat the current handoff document as authoritative for the work in flight, and record when it overrides older material. The conflict stays visible instead of silently splitting into two realities.

Keep the map small. Adding a new canonical document should be harder than adding a note to the right existing one.

10.3 Update the whole contract surface

Ship a language feature and your parser is the smallest part of the change. The feature is incomplete until the documentation, the website, the embedded machine-readable specification, the editor grammar, the examples, and the migration behavior agree with it.

Agents and tools consume these surfaces as operational inputs, so this is more than documentation hygiene. A stale embedded specification can teach another agent to emit syntax your parser no longer accepts. An old editor grammar presents removed syntax as valid. A migration page can preserve a semantic model the runtime has abandoned.

Make the update in the same branch as the behavior, while the author still holds the full model in context.

A contract-surface checklist might include:

- source and tests;
- CLI help;
- reference documentation;
- examples and fixtures;
- website and generated figures;
- editor integration;
- embedded agent instructions;
- migrations and compatibility notes;
- issue and release state.

Not every project has every surface. Every project has more than the code.

10.4 Record rejected ideas when they will recur

Review sometimes proves a concern false, and the refutation is worth keeping when the false premise is attractive. Consider a port of a published typesetting algorithm. A reviewer concludes that a rule was inverted during the port; the algorithm's published source shows it was not. A short note or test tied to that reference stops the same finding from returning with every new reviewer.

Durability is where this usually fails. Refutations tend to live wherever the review happened, in a scratch file one reboot away from vanishing. Move them into the repository's committed notes. The raw candidate list from the review will not survive, and it does not need to: the accepted findings and the reasoned rejections are what the next session needs.

Do not archive every rejected comment. Preserve the ones that reveal an ambiguous design, a misleading code shape, or a domain rule that is easy to misremember.

10.5 Let guardrails remember and evolve

Suppose your pre-commit gates on complexity and file size catch real debt and also create repeated friction, because the hook does not understand the code that predates it. The wrong responses are to bypass the gate or to declare the friction a feature. Pay down some of the debt and record that the gate needs to learn the baseline.

That is the right relationship with guardrails. They should be:

- enforced enough to rely on;
- transparent enough to debug;
- narrow enough to explain;
- revisable when they impose accidental cost;
- accompanied by the failure they are meant to prevent.

10.6 Memory should shorten the next session

The test of project memory is not page count. It is how fast a fresh agent can answer:

- What owns the truth?
- What must never be lost?
- Which checks defend behavior that must not regress?
- Which tempting shortcuts already failed?
- What is the current plan?
- Which document wins a conflict?
- What must be updated before the work is actually shipped?

Good memory turns old pain into a lower future cost. The bug gets paid for once.

11 Preserve what the user wrote

An editor's most dangerous defects leave the screen looking right. The rendered document appears fine even though the source changed, a delimiter vanished, or the caret moved to a different logical position. The same exposure exists in any system that holds user data behind a friendlier projection of it, and appearance alone cannot verify any of them. The product needs a clear answer to one question: what is the document?

11.1 Keep one source of truth

The strongest design treats the source string and its syntax tree as the document. Everything on screen is a projection. A projection may hide syntax, replace a source range with a rendered block, or add visual decoration. It does not own an independent version of the content.

This asymmetry removes an entire reconciliation problem. The view can be discarded and rebuilt. The source cannot depend on reconstructing itself from the view.

A rich-text editor that treats both the view model and the source as authoritative must synchronize two truths after every edit. That design creates hard questions:

- Which model wins when whitespace differs?
- How are malformed constructs represented in the rich model?
- What happens to source details the view cannot express?
- Can a selection survive conversion in both directions?

The one-truth design refuses the premise. The source wins.

11.2 Prove byte preservation

Semantic equivalence is not enough for a source editor. Two documents may render identically while differing in line endings, spacing, delimiters, or user formatting. Those differences belong to the author.

The preservation suite should cover:

- line-ending forms;
- grapheme boundaries;
- hidden delimiters;
- malformed but retained syntax;
- whitespace around blocks;
- text revealed and concealed by projections;
- edits adjacent to projected content;
- repeated round trips.

A useful fixture begins with an exact byte sequence, performs a projection or edit cycle, and compares the resulting bytes. The regression for an ordinary edit can state the smallest permitted mutation and prove everything outside it stayed unchanged: replace one word in the middle of a line, then assert that every byte outside that span is identical, that undo restores the exact original, and that redo reapplies the exact edit. A separate test should pin UTF-8 slicing against a fixture that mixes one-, two-, and four-byte characters, say an ASCII letter, an accented vowel, and an emoji, because the interesting offsets are not where ASCII intuition puts them. Normalizing a fixture before comparison would erase the contract the test is supposed to protect.

Details at this level are not trivia. `\r\n` is one grapheme cluster in two bytes, and that fact marks a place where text, bytes, and cursor behavior intersect.

11.3 Treat the caret as state

Projection changes can preserve text and still betray the user by moving the caret. A user who reveals source syntax expects to remain on the same conceptual line, not jump because the visible range changed.

Name this as a viewport invariant and test it as a number. Suppose your editor reveals a table's source when the user activates it. Record the screen position of the row, activate it, relocate the caret, and assert the row moved less than 2 points. The failure symptom is the one the user would report: the row you clicked jumps when it becomes editable. The test operates on a user-visible property while allowing implementation details to change.

The best bug reports in this domain are exact and perceptual:

> *The cursor lands between the `n` and `g` of `formatting`.*

That sentence gives the agent a concrete anchor. The investigation can trace source offsets, rendered ranges, grapheme clusters, and line layout until the model explains the position.

11.4 Compare incremental and full projection

Editors optimize by patching only the affected projection. That creates two implementations of the same contract: incremental update and full recomputation.

Equivalence tests should require both paths to produce the same relevant state. The hard part is defining relevant. Whatever fields you enumerate first, expect the list to be incomplete. If your editor carries suggestion marks, an equivalence check that omits their metadata lets the fast path leave a mark's byte range stale while the suite stays green.

Attack the checker field by field. Change one field in a known fixture and confirm that the test fails. A comparison that has never noticed a selection or

decoration mismatch should not be assumed to cover them.

11.5 Pay the invariant's cost openly

A constraint that costs nothing may not be constraining much. The one-source rule forbids the most attractive editor shortcut: replacing revealed source with rendered content inline. Byte mapping depends on every character staying where the file says it is. Under that rule, hidden delimiters become one-point clear glyphs instead of being removed. Live preview moves to a side panel. A held-preview state carries the last good render while mid-edit source is invalid. Suggestion marks with absolute byte ranges force a full parse, because the incremental fast path cannot shift them safely.

None of this is free. The design accepts a slower path and a more complex product to keep the mapping exact. The visible costs are also the evidence that the invariant is real. Ask yourself where your strongest architectural rule made a feature harder rather than easier. A specific answer means the rule has been tested against real work. No answer usually means the rule constrains nothing.

11.6 Preserve unsupported source visibly

A source editor will encounter syntax it cannot render or edit structurally. The safe fallback is visible source, not a partial projection.

The rule holds beyond prose. If your editor typesets math or lays out diagrams, a broken diagram should become a labeled source card, pathological math should degrade to an unsupported marker instead of crashing, and raw embedded markup should display as styled literal source. When the projection cannot represent the input faithfully, show the source and the diagnostic rather than guessing or deleting.

The fallback must remain locatable. A generic error card that loses the original span prevents the user from repairing the text. Preserve the range, the literal source, and the reason the projection declined it.

11.7 Use the model to diagnose the pixels

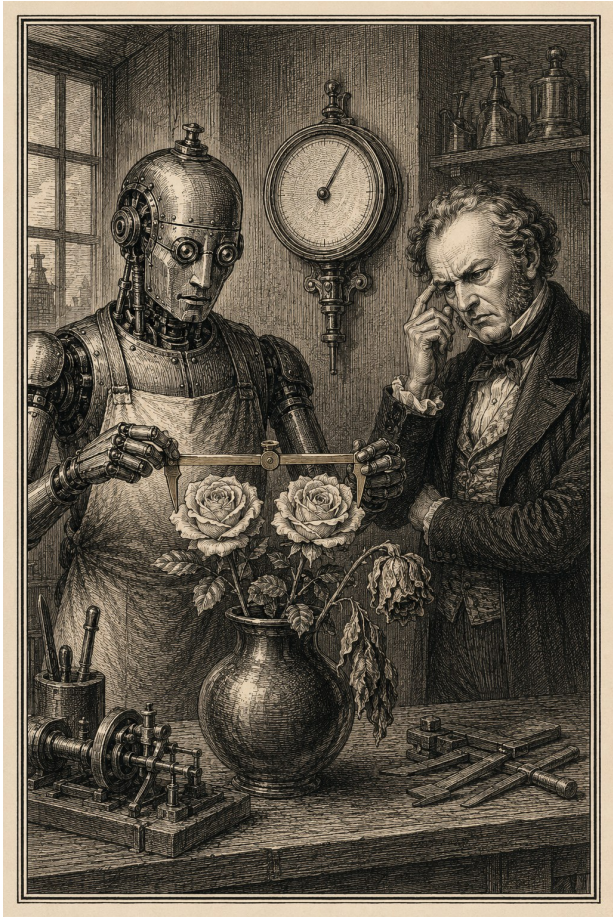
A user sees the screen. The agent should inspect the layout model behind it. Exact offsets and geometry are more reliable than a second round of squinting at a screenshot.

This does not eliminate visual tests. It puts them at the right boundary:

- human observation identifies the perceptual failure;
- a calibrated capture records it;
- model dumps localize it;
- invariant tests preserve the repair.

The editor remains pleasant because the human supplies perception. It remains trustworthy because the source model supplies proof.

12 Machine-checked taste



Some judgments have proxies. The rest still need the critic.

Rendering systems turn structured input into geometry and pixels. Their hardest defects pass type checking and look plausible in data. A label sits on a bend. Two words crowd each other. A glyph is present but shifted. A render harness declares a colored equation blank.

The defense is layered: separate layout from rendering, then give each layer its own proof. Some perceptual judgments can become executable rules. Knowing which ones is an engineering skill in its own right.

12.1 Keep layout platform free

Strong rendering systems share a shape: frontends that parse languages into a shared representation, a platform-free layout layer, and backends that render scenes.

The boundary makes several things cheap:

- a new language reuses layout, linting, and every backend;
- a new backend traverses an existing scene;
- geometry tests run headless on Linux;
- platform rendering can be injected and compared;
- a failure localizes to parse, layout, or draw.

Without the split, a visual bug collapses into a platform screenshot. With it, the agent can ask whether the wrong box was generated or the correct box was drawn wrong.

"Platform free" deserves a precise reading. Your layout layer can import no platform UI and still consume platform ink metrics through an injected measurer, and a clamp on a font's ink ascent can dominate the constants around it. Expect the test doubles to lie about this. A mock measurer with generous ink can hide the very clamp a test means to exercise, so proving the clamp works may take a special low-ink mock. Substitute metrics injected for headless runs can be wrong for most of the fonts they stand in for. The core is portable and headless. Its output still differs by platform, and separate goldens per platform are the honest way to say so.

12.2 Convert a perceptual complaint into geometry

The complaint usually arrives as a sentence that cannot be unit tested:

- the label sits on the bend;
- these two labels feel crowded;
- the arrows make the midpoint look wrong;
- the accent is not seated on the glyph.

The agent's job is to find the stable geometric rule underneath. Consider a diagram engine that centers each edge label on its arrow, and a complaint that one label sits on an arrow too short to see. The rule that comes out is a product decision, not a coordinate: a label belongs to the visible shaft, so the midpoint is computed after subtracting an 11.5-point arrowhead inset from each end, and if too little shaft remains, the layout extends the segment.

The durable rules in this family involve:

- label box distance from a fixture;
- overlap between text and edge segments;
- vertical staggering of neighboring labels;

- duplicated or coincident edges;
- midpoint of the usable shaft after arrowheads come off;
- accent attachment to glyph metrics.

Fix the generator and add a linter rule. The generator stops producing the bad geometry; the linter catches any other route that reintroduces it. Give the two different constants on purpose. In the diagram engine above, the generator reserves a 14-point label stub while the linter fails below 10 points, so the aim point has headroom over the ratchet.

This turns one act of taste into a permanent floor. You remain responsible for noticing new ugliness. The machine remembers old ugliness.

12.3 Use exact geometry before pixels

Geometry tests are fast, deterministic, and easy to localize. They should assert domain concepts, not incidental coordinates.

A weak test says a label's x coordinate is 143.5. A stronger test says the label does not intersect a fixture and is centered on the usable shaft. The coordinate changes with font metrics. The rule stands.

Property tests cover whole classes:

- no label intersects any fixture;
- parallel edges remain distinguishable;
- every visible node has a finite bounding box;
- route segments connect within tolerance;
- layout is deterministic for a fixed input and font set.

These checks do not prove the render looks good. They remove many ways it can look predictably bad.

12.4 Add pixel proof where geometry stops

Fonts, antialiasing, clipping, color, and backend behavior live below geometry. Golden images or render signatures can catch failures there, but only if the detector has proved it can see. The quiet failures are the dangerous ones: an ink signature that averages thin strokes into invisibility, a golden file of zeros that still passes, a red-channel metric that calls a red glyph blank. Test the harness with fixtures that vary:

- thin and thick strokes;
- grayscale and colored glyphs;
- empty and nearly empty images;
- small translations;
- clipping at each edge;
- font fallback;

- lazy drawing that has not been forced;
- expected antialiasing differences across platforms.

The harness may use exact pixel diffs, perceptual metrics, ink bounds, or structural signatures. Whatever the metric, it should follow the contract and reject a known-bad probe.

12.5 Compare against an independent renderer

The strongest reference is a system that shares none of your implementation. Your layout code and your tests can repeat the same wrong assumption, and a separate renderer gives that assumption a chance to disagree. Publishing your output beside the established engine, same input, matched height, puts the disagreement where anyone can see it. But a side-by-side supports visual judgment, not an equivalence claim. State which differences matter, and until a tolerance exists, do not call the comparison a test.

Independence can also come from inside the system: a full-render path checked against a patch-render path, multiple backends, structural linting against the generated scene.

12.6 Generate the documentation

Rendered documentation should come from the live engine. A gallery that regenerates in CI on every push cannot freeze a changed example inside a stale screenshot, and it exposes the marketing to the same failures as the product, which is exactly the point.

Watch for the assets that sit outside the loop. A wordmark your engine typeset once and you then committed as a static image proves nothing about today's build. "Typeset by the engine itself" is historically true and verifies nothing. Provenance and verification are different properties, and a documentation pipeline should know which one each figure has.

Generation alone is not enough. The figure pipeline needs tamper tests, calibrated captures, and review. The proof should be easier to inspect than the claim it supports.

12.7 Know where the proxy stops

The hardest defect to encode is label-to-edge association: a label can clear every geometric fixture and still appear to annotate the wrong one of two near-parallel edges. The judgment is semantic, and no lint rule expresses it yet.

The constants have a domain too. A minimum-clearance floor is a proxy for perceived crowding, and perception moves with font, scale, line weight, and neighboring geometry while the constant stands still. Every proxy has a range

where it stops matching experience.

Trust in a renderer is layered: exact structure, tested pixels, independent comparison, and a human who still notices when the composition feels wrong.

13 Semantics that cross boundaries

The repository boundary is not the truth boundary.

A parser accepts the text, but two output modes disagree on trailing whitespace. A workflow declares a gate that can never run. A checkpoint resumes while the audit trail describes a different path. In language tools and workflow engines, syntactic validity says nothing about semantic truth.

These systems need proof that follows state across boundaries, including boundaries the current repository does not contain.

13.1 Design the language model before the parser

Design gates surface the decisions that shape every later component. Should composition cascade, include, or expand into tokens? Where should an unterminated quote be rejected? What does a migration preserve? Which runtime consumes the output?

A parser can implement any of these models faithfully. Correctness depends on choosing the right one.

The design should state:

- the accepted language;
- the canonical representation;
- the malformed-input policy;
- byte-sensitive behavior;
- compatibility and migration rules;
- downstream runtime assumptions;
- the independent oracle or corpus used for proof.

The parser comes after the contract.

13.2 Test byte boundaries explicitly

Language tools often pass semantic tests while corrupting bytes that other tools care about. Consider a prompt-composition feature with two packaging modes, inline and source, whose contract is byte identity at the final line. The inline path routed composed fragments through a formatter block that right-trimmed; the source path used the raw composition. A fragment ending in a newline produced different bytes in different modes, on the exact final-line contract the feature existed to preserve. The repair normalized whitespace in one shared function, applied only when fragments are composed, with an early return that leaves ordinary content untouched. The scope of a byte-level fix is part of its correctness.

Round-trip tests should cover:

- empty and final lines;
- trailing spaces and tabs;
- escape sequences;
- quoted delimiters;
- invalid and partially valid constructs;
- alternate packing or serialization modes;
- migration through multiple versions;
- source reconstruction after diagnostics.

When byte identity is the contract, parse-tree equality is not a substitute.

13.3 Use a simple reference implementation

Pair compact language machinery, an escape-aware condition parser for instance, with a reference implementation that is allowed to be slow and inelegant. Its job is not production. Its job is disagreement.

```
for input in generated_cases:
    assert production(input) == reference(input)
```

Generated inputs are cheap, so run tens of thousands of them. Emphasize combinations that hand-written examples miss: nested escapes, empty tokens, repeated separators, partial endings, and boundaries around maximum lengths.

The reference must differ enough from production to avoid sharing the same bug. A straightforward decoder and an optimized state machine are a useful pair. Two copies of the same algorithm are not.

13.4 Prove workflow reachability, not declaration

A goal gate that never executes can still appear covered. Workflow tests need to distinguish several states that broad coverage metrics may merge:

- declared;
- reachable;
- selected;
- executed;
- completed;
- persisted;
- resumed;
- cleared for a later loop.

A test that observes only final success can miss the path that produced it. The audit and checkpoint state should allow the system to explain which edges ran and why.

Useful workflow properties include:

- uninterrupted and resumed runs agree where the contract requires;
- retries do not duplicate irreversible effects;
- loop-local state is cleared at the correct boundary;
- an unreachable gate cannot count as executed;
- audit entries correspond to actual transitions;
- failure states remain inspectable;
- terminal outcomes cannot be resumed accidentally.

These are state-machine claims, even when the code is written as ordinary functions.

13.5 Keep runtime reality in the loop

The sharpest cross-boundary failures depend on facts from a separate repository. Consider a workflow definition language whose output executes in a runtime that lives in another checkout. The language tool modeled retry destinations as routing edges. In the runtime, a retry outcome dispatches to a dedicated path that never consults edge selection at all. From the tool's side, retry and routing looked related. In the runtime they were disjoint channels with their own defaults.

The blind spot extended to the tool's own simulator, which exercises graph structure. Retry outcomes arise from runtime infrastructure, cost limits, and no-progress conditions that no graph walk produces. A migrated workflow could keep a byte-identical happy path while corrupting retry behavior, and every local tool would agree it was fine.

Your checkout may define syntax without defining execution. An agent working in it can reason perfectly from incomplete premises.

Cross-repository assumptions should be written into the design and verified against the consumer. Useful methods include:

- contract tests that run both repositories;
- pinned integration fixtures;
- generated compatibility matrices;
- explicit links to the consuming code and version;
- a human checkpoint when the behavior depends on product knowledge outside the tree.

The repository boundary is not the truth boundary.

13.6 Sweep every language surface

A language feature changes more than a parser. A complete sweep covers documentation, the website, examples, editor grammars, embedded agent instructions, migrations, and downstream expectations in the same batch.

This prevents a distinctive agentic failure: one model updates the parser, another later reads stale instructions and generates the old language with equal confidence.

The sweep itself should be tested. Parse every example in the repository with the real parser and linter, with specific lint codes forbidden. Give the embedded machine specification a freshness test that regenerates it and fails on staleness. Require exactly one explanation per diagnostic code, no more and no fewer. Instructions that machines consume need executable checks, because to another agent they are code.

13.7 Refuse speculative workflow patches

Workflow failures often depend on remote state or diagnostic artifacts. When you have ruled out every code path in the checkout and the remaining cause lives in an artifact you do not have, stopping is the correct completed result. A speculative patch to a workflow engine can create a second state inconsistency while hiding the first.

The language and workflow domains reward patience because their failures travel. A one-line parser change can alter migrations, editors, agents, and runtimes. A one-line state change can alter checkpoints, retries, and audit history. Design, proof, and release must travel with it.

14 Divide the work where each side is strongest

No formula for dividing work with coding agents survives contact with real tasks. "Human plans, machine codes" fails, and so does the tidier "human supplies taste, machine supplies execution." The line moves from task to task. What stays stable is the kind of contribution each side makes best.

You keep the tactile, product-level judgment: whether the software feels coherent when you actually run it, not merely whether it satisfies repository contracts. You set the standing pressure toward simplicity, decomposition, and incremental work. You watch plans as they form, challenge assumptions, and recruit adversarial reviewers from other domains. Agents choose frameworks and implementation approaches, and they supply search, implementation breadth, repetition, and stamina. Neither side certifies its own work.

14.1 Name the symptom, not the fix

A precise perceptual report can be more useful than a detailed implementation request:

> *The cursor lands between the `n` and `g` of `formatting`.*

> *The `fail` and `reject` labels are crowding each other.*

> *The midpoint should exclude the arrows.*

Each report tells the agent where reality violated intent without forcing a solution. The agent can dump offsets, inspect geometry, trace the render path, and build the regression proof.

You do not need to translate taste into code before asking for help. That translation is part of the work you are delegating.

14.2 Ask questions that raise the bar

Some of the most valuable directions you can give are questions:

- Do we understand every microsecond?
- Can your typesetting engine render its own logo?
- Is that issue actually done?
- Why are two open pull requests touching the same code?
- What happens when this gate never runs?

Each one exposes a standard the current work may not meet. A question is

often cheaper and more generative than a full specification because it invites the agent to investigate rather than comply.

A strong question points at evidence. "Make it robust" is vague. "What known-bad input proves this harness can fail?" changes the next action.

14.3 Keep corrections cheap

Corrections should be fast and small. A day's worth might read:

- the Linux path uses a different renderer;
- `--no-verify` is not allowed;
- the work belongs in a separate pull request;
- the site is several releases behind;
- the count is nineteen, not seventeen.

The value is the low ceremony. The correction enters the system, the work changes, and a repeated failure may become a standing rule.

Cheap correction is what permits broad delegation. If every correction requires a defense of the original work, you will micromanage and the agent will optimize for reassurance.

14.4 Delegate judgment, then verify state

You can safely say "keep going" when the repository already encodes the limits: the test gate, the branch protocol, the documentation sweep, the review requirement, the completion bundle.

You still spot-check state at important boundaries:

- Is the issue closed?
- Does the pull request already exist?
- Is `main` green after the merge?
- Are the release assets published?
- Does the live site serve the new version?
- Did the test fail on the old code?

This is not micromanagement. It is checking the places where conversational state and repository state commonly diverge.

The spot checks do not stop at the repository. Even with execution fully delegated, you still run the application, exercise the interaction, and open the published page. The shipped result exists outside the code and the test suite, and someone has to meet it there. These product-level checks are a deliberate part of the division, not a residue of distrust.

14.5 Delegation is not autonomy

It is tempting to assume that broad delegation lets agents run indefinitely. It does not. Delegation does not create autonomy; a completion harness does.

Agents tend to stop after a bounded unit and ask you to inspect it. That behavior does not disappear because the permission was broad or the prompt was confident. Sustained execution comes from machinery that defines completion and makes premature stopping difficult: tests that must pass, workflows that continue, review loops that consume the output, and completion criteria the agent cannot satisfy by summarizing.

The operating model this produces keeps you interactive. You watch plans form, ask questions, and apply pressure, while the harness supplies the persistence. That much interaction looks like a bottleneck. The expensive arrangement is the other one, where your attention is the only thing keeping the work moving.

14.6 Supply facts outside the checkout

You may know how a downstream runtime behaves, why a product promise matters, or which compatibility break users will accept. A decision about retry semantics can hinge on how a consumer in another repository calls your library, and no amount of reading the local checkout will surface that fact.

Agents should make these dependencies visible rather than invent them. A design can list its external assumptions and ask for the one fact that changes the policy. Once supplied, the fact should move into a contract test or a durable note.

Your knowledge is most valuable when it becomes less necessary next time.

14.7 Do not let either side self-certify

You can be wrong about a screenshot. The agent can be wrong about a test. A reviewer can be wrong about a typesetting rule. A profiler can be wrong about work deferred by lazy evaluation. The remedy is not a better judge. It is disagreement between independent representations:

- your taste challenges the render;
- computed geometry challenges the screenshot;
- a known-bad fixture challenges the test;
- a reference implementation challenges the parser;
- the integrated tree challenges the feature branch;
- the live artifact challenges the release report.

This is why high trust can coexist with skepticism. People and agents get wide freedom to produce. Evidence decides what survives.

15 A small operating playbook

The method works best when the disciplined path is the easiest path, and easiest is something you build. Prepare the repository once. Then run every change through one loop, at whatever weight the change deserves.

15.1 Prepare the repository

Before you assign several agents, put seven things in place.

15.1.1 A document map

Name the sources of truth for product behavior, architecture, active plans, invariants, compatibility, and scars. State which source wins when they disagree.

15.1.2 An invariant suite

List the properties the project cannot casually violate, and give each one a named executable check when practical. Write them in domain language: byte preservation, caret-line stability, route reachability, checkpoint equivalence, label clearance.

15.1.3 A complete command-line path

An agent should be able to build, test, lint, collect logs, capture diagnostics, regenerate documentation, prepare releases, and inspect the live result without hitting a step that only works in a GUI.

15.1.4 Fast local gates

Pin the formatters and linters. Keep targeted checks quick and the full suite reliable. A slow or opaque gate invites bypass.

15.1.5 Branch rules

Decide the worktree, commit, rebase, merge, issue, and documentation protocol once. Reuse it instead of renegotiating mechanics in every conversation.

15.1.6 A scar file

Record the bug classes behind your standing rules. Keep it short enough that a new agent will actually read it.

15.1.7 A completion harness

Define what done means for a delegated unit and encode it where the agent cannot argue with it: passing gates, a required review pass, an updated contract surface, a verified release. Agents stop at bounded units and hand the work back. Delegation grants permission to continue; only the harness makes continuing the default.

15.2 Run the change loop

15.2.1 Reproduce

Use the real binary at the current revision. Save the input, the command, the trace, or the artifact.

15.2.2 Name the semantic decision

A small fix gets a paragraph. A larger change gets a design. Either way, identify the canonical state, the failure behavior, the compatibility rule, and the proof.

15.2.3 Attack the design

Assign independent review lenses. Look for unreachable states, data loss, migration traps, platform differences, and weak test oracles.

15.2.4 Implement in isolation

One branch or worktree per coherent decision. Keep unrelated judgment out of the diff.

15.2.5 Add the ratchet

Write the named regression check, then revert, tamper, or mutate to prove it detects the old behavior. Use an independent oracle when one exists.

15.2.6 Review the code and the proof

Ask fresh reviewers to refute both. Require concrete code references and failing cases, and validate their findings before acting on them.

15.2.7 Repair or reject

Every accepted finding gets a fix and evidence. Every rejected finding gets a technical reason. Every deferred finding states what is missing.

15.2.8 Integrate one change

Refresh against current `main`, rerun the branch checks, merge, and run the full suite on the merged tree.

15.2.9 Sweep the contract surface

Update the docs, examples, site, editor integration, embedded agent instructions, migrations, issue state, and release notes that belong to the change.

15.2.10 Verify the published result

Check the built artifact, the release assets, the install path, and the live surface. Confirm that the whole required CI graph concluded green, not one workflow: a release job can publish happily while a required check sits red. A green workflow is not the thing users receive.

15.2.11 Preserve the lesson

If the work exposed a reusable failure mode, add the smallest durable test, tool, or scar note that prevents rediscovery.

15.3 Use evidence bundles instead of "done"

A completion report should contain handles another person or agent can inspect:

- behavior changed;
- root cause;
- old behavior reproduction;
- regression proof;
- deliberate red run or known-bad probe result;
- review findings accepted and rejected;
- commands and revision tested;
- integrated-tree result;
- documentation and release state;
- remaining uncertainty;
- new scar or standing rule.

The bundle does not need to be long. It needs to make every important claim rerunnable.

15.4 Keep the human decision surface small

The agent should not ask permission for routine mechanics the repository already covers. It should stop for choices that change product meaning, compatibility,

acceptable loss, or scope.

A good checkpoint presents:

- the fork;
- the evidence for each branch;
- the cost of reversal;
- the recommended choice and why;
- the one fact only the human can supply.

Once you make the choice, execution resumes without repeated approval.

Audit the checkpoints themselves occasionally. A gate whose answer is predetermined by standing policy cannot change the work; remove it rather than respect it. Keep the gates where disagreement is possible and would matter.

15.5 Know when not to use the full loop

A typo in prose does not need a five-reviewer panel. A safe dependency pin may need only the build, the tests, and the generated artifacts. Let the process follow the risk.

Use the full loop when the change touches canonical state, persistence, malformed input, migrations, security, data loss, public contracts, or a proof system. Use a smaller loop when the behavior is local and the existing invariants already cover it.

The method should reduce uncertainty, not manufacture paperwork.

16 Failure modes that look productive

Agentic development can fail while producing impressive activity. The repository fills with branches, tests, review comments, plans, and release notes. The surface looks rigorous. The product grows harder to trust. Each pattern below earns suspicion because it imitates a discipline without doing its work.

16.1 Prompt theater

You write a long prompt that tells the agent to be careful, test thoroughly, and think step by step. The repository still lacks executable invariants, a complete command-line path, and an integration gate.

The prompt may improve behavior. It does not create proof. Move standing rules into scripts, tests, documents, and branch policy until the prompt can get shorter.

16.2 Design theater

Every change gets a document, but the document never exposes a decision that could invalidate the work. It restates the ticket, predicts file names, and declares familiar principles.

A useful design names the canonical state, the failure policy, the compatibility boundary, and the proof. A tiny change may need only a paragraph. More pages do not compensate for an implicit semantic fork.

16.3 Review volume as quality

A fleet of review agents generates dozens of findings, and every finding becomes an issue. No one checks whether the code already handles them or whether the reviewer misunderstood the domain.

This burns trust. Review output is candidate evidence. It needs reproduction, triage, and reasoned rejection.

16.4 Self-consistent proof

The implementation and the test share a helper, an algorithm, or a mistaken assumption, so they agree perfectly. A test recomputes the expected midpoint with the production formula. A screenshot tool captures through the same broken rendering shortcut. A parser oracle duplicates the optimized state machine.

Use a different representation when you can. At minimum, break the implementation and prove the test notices.

16.5 Green-branch optimism

Every feature branch passes. No one tests the merged tree after each integration. A release is cut on the assumption that local green composes.

Interactions appear only in the shared tree. Green must name the revision and the environment. The same optimism has a release-time form: a green publish workflow treated as repository health while a required check sits red in another lane. Verify the conclusion of the whole required graph.

16.6 Provenance worn as verification

Consider a typesetting engine whose wordmark the engine itself once rendered, committed alongside the script that generated it. Nothing in the current build regenerates or validates the file. "Generated by the engine" states provenance; readers hear verification. The two properties diverge silently as the engine changes underneath the committed asset. Say which property each artifact has, and promote provenance to verification wherever the claim carries weight.

16.7 Parallelism without an integrator

Agents work in isolated branches, then merge each other's changes opportunistically. No one owns the final sequence or reruns the full suite after every merge.

Parallel production requires a serial trust boundary. The integrator does not need to write the code. The integrator must vouch for the tree.

16.8 Snapshot dependence

Visual quality is reduced to screenshot approval. The capture path may be wrong, lazy work may not have run, and small geometry defects hide in noise.

Use screenshots for user-visible evidence, exact geometry for localization, and a tested pixel detector for backend behavior. None suffices alone.

16.9 Guardrail worship

A complexity limit or lint rule blocks your work, so you bypass it or restructure code only to satisfy the number. No one asks whether the rule still tracks the risk.

A guardrail should be enforced and debuggable. Record the debt it catches and the accidental friction it causes. Improve the mechanism instead of worshiping or evading it.

16.10 Documentation after the fact

The code lands first. The site, the examples, the embedded agent instructions, and the migrations go to a later cleanup ticket.

This opens a window where every consumer sees a different language or product. Update the contract surface while the implementation context is still active.

16.11 Forced continuation

The agent cannot prove the root cause but produces a patch anyway, because returning empty-handed feels like failure.

A good workflow accepts "missing artifact" or "no safe change" as completed investigative results. Keep uncertainty visible instead of converting it into code.

16.12 Tidy stopping

The agent completes a bounded unit, files a tidy report, and waits. Every increment looks disciplined. The queue never advances without a human nudge.

This is the mirror image of forced continuation and easier to miss, because each pause resembles care. Agents stop at locally satisfying units unless a completion harness defines completion and makes premature stopping difficult. When the repository can already verify the next step, pausing to ask is not caution. It is missing machinery.

16.13 Human bottleneck disguised as quality

You review every line, rerun every command, and answer routine questions the repository could settle. The work is correct only while you are awake and attentive.

Move repeatable judgment into invariants and scripts. Reserve yourself for taste, product choices, external facts, and novel failure classes.

16.14 Synthetic certainty

The final report is polished, complete, and free of doubt. It does not state the revision, the old failing case, the rejected review findings, or the remaining uncertainty.

Confidence is fine. Confidence you cannot inspect is not.

17 What good looks like

Projects that go well with coding agents do not go well because the agent stopped making mistakes. It never does. It still writes the review finding that dissolves under scrutiny, the benchmark that flatters, the fix that quietly breaks a neighbor, the feature that ships and then has to come out. What improves is the price of exposure. In a healthy project, each mistake gets caught earlier, by something cheaper than your full attention.

That is the standard to look for.

17.1 The repository carries the discipline

A healthy agentic project does not depend on anyone remembering the perfect prompt. A fresh agent, cold, can find:

- one canonical source for each important behavior;
- named invariants that use domain language;
- commands that reproduce, test, build, and publish;
- a branch protocol that keeps changes bounded;
- a record of failures worth remembering;
- a definition of done that reaches the live product.

The conversation can be terse because the repository is not.

17.2 Proof gets harder to fake

The checks do not all repeat one assumption. The regression test fails on the old code before it passes on the new. The fixture verifier rejects a deliberately damaged fixture, which proves it can reject at all. A reference implementation disagrees the moment the parser drifts. A layout check and a pixel check cannot be fooled by the same bug. The integrated tree runs the full suite after each merge, not just the branch's slice. The published artifact gets opened by hand instead of inferred from a green badge.

No single check is trustworthy. The system earns trust by giving a plausible fiction several places to break.

17.3 Your attention goes to novel problems

You notice the cursor jump, the crowded labels, the claim that sounds too flattering. The agent turns the observation into traces, tests, and documentation. Once the failure class has a name and a detector, a ratchet carries it forward, and you never have to notice that defect again.

That is how taste compounds.

17.4 Stopping is allowed

A good session can end with a revert, a rejected finding, a corrected document, or a request for a diagnostic that does not exist yet. The workflow does not demand a patch as proof that work occurred.

That freedom matters. It keeps generation from becoming the default answer to uncertainty.

17.5 Reversal is ordinary

Small branches, explicit migrations, and narrow contracts let you change course without defending sunk cost. "We were wrong" becomes a normal engineering state rather than an emergency. Correctness has to cost less than pride, and these structures are what make it cost less.

17.6 The verdict is assembled, not announced

Ask whether the software is done and you get testimony, not a verdict. The agent says the task is complete. The test suite says its assertions hold. The pull request says reviewers found nothing more. The release workflow says the pipeline ran. The application, opened by hand, says what it actually does. These witnesses never describe quite the same thing, and that is not a defect to engineer away. Each covers ground the others cannot see, and the disagreements between them are where the defects live.

So the honest answer to "is it done?" is almost always "mostly," followed by particulars: which conditions are established, on what evidence, and which remain open. The craft is making that sentence precise and cheap to say. Name the conditions. Give each one an executable witness. Prove the witnesses can object. Let no author, human or machine, close the case alone.

One question sorts workflows:

> *When the agent is confidently wrong, what happens next?*

If the answer is that you will eventually notice, the system runs on vigilance, and vigilance does not accumulate. A stronger answer is visible in the repository itself: the change is small, the claim has a known-bad test, independent reviewers try to break it, the merged tree runs clean, the public artifact agrees, and the lesson survives the session.

Software declared finished that way is still only mostly done. But it is mostly done the way a well-run investigation is mostly closed: the open questions are named, the evidence is filed, and no one had to take anyone's word for it. That is enough to build actually good software with a fallible author. It is also a good way to build software with people.

Appendix A: Checklists

These checklists compress the operating method. They are short on purpose. Add a project-specific line only when a real failure in your project earns it.

Before implementation

☐ Reproduce the current behavior against the real binary or artifact.
☐ Record the revision, environment, input, command, and observed result.
☐ State the one semantic decision that could invalidate the work.
☐ Name the canonical state.
☐ State how the system treats malformed, unsupported, or partial input.
☐ Identify compatibility, migration, and rollback boundaries.
☐ Name the invariant that will prove success.
☐ Separate unrelated judgment calls into other branches.

Design review

☐ Can a declared path be unreachable?
☐ Can a resumed path differ from an uninterrupted one?
☐ Can data disappear while the UI still looks plausible?
☐ Can malformed input be silently coerced?
☐ Can two modes agree semantically but differ in bytes?
☐ Does the proof share logic with the implementation?
☐ Does the change rely on a fact outside this repository?
☐ Can the change be reversed without unrelated damage?
☐ Is the design more complex than the contract requires?

Regression proof

☐ The new test fails on the old implementation.
☐ The failure occurs at the expected assertion.
☐ A deliberate tamper or mutation is detected.
☐ Every semantically relevant field is compared.
☐ Lazy work is forced before measurement.
☐ The test uses an independent oracle where practical.
☐ The targeted test and surrounding suite both pass on the fix.

Review triage

- ☐ Each finding cites code or a contract.
- ☐ Each finding supplies a concrete exposing input or state.
- ☐ Accepted findings receive a repair and proof.
- ☐ Rejected findings receive a technical reason.
- ☐ Deferred findings identify the missing evidence or scope.
- ☐ High-value false positives are recorded when likely to recur.

Integration

- ☐ The branch is refreshed against current `main`.
- ☐ Targeted and full checks pass on the refreshed branch.
- ☐ One coherent change is merged.
- ☐ The full suite passes on the new integrated tree.
- ☐ Regenerated artifacts contain real changes, not encoding noise.
- ☐ A whole-codebase migration merges alone, not inside an unrelated parallel batch.

Release

- ☐ Documentation and examples match the accepted behavior.
- ☐ Every surface that restates the behavior is current: site, editor integration, embedded agent instructions, migration notes.
- ☐ The whole required CI graph concluded green, not only the release workflow.
- ☐ Expected release assets exist and open.
- ☐ The install path delivers the intended version.
- ☐ The live surface serves the intended version.
- ☐ Issue and pull-request state match what shipped.
- ☐ Artifacts described as generated are regenerated or validated by the current build.

After the work

- ☐ Record the reusable bug class, not only the local fix.
- ☐ Add or revise a standing rule only when the failure justifies it.
- ☐ Record guardrail friction that needs a mechanism fix.
- ☐ State any remaining uncertainty.
- ☐ Remove temporary process notes that no longer define behavior.

Appendix B: Eleven takeaways

If you keep nothing else, keep these eleven.

1. **The system, not the prompt, is the unit of quality.** Terse instructions work when the repository defines done, encodes the checks, and carries the conventions. Improve the machinery before improving the wording.
2. **Architecture is a verification feature.** Choose boundaries by asking how much of the system must be trusted for a small change to be believed. A boundary is cheap to cross only when the representations crossing it already carry the needed meaning.
3. **A principle you cannot execute is a preference.** Give every important belief a named check that fails when the belief is violated, and enforce it at a boundary ordinary code cannot bypass.
4. **A test that has never failed is a rumor.** Revert the fix and watch the red. Tamper with a golden file and watch the harness object. Trust a check in proportion to the evidence that it detects what it claims to detect.
5. **Distinguish provenance from verification.** "The system generated this once" and "the build regenerates or validates this now" are different properties. Only the second catches a regression. Say which one you have.
6. **Write the disagreement before the code.** State the decision that could invalidate the whole patch while changing it costs a few sentences. Then remove every gate whose answer is predetermined, because a pause that cannot change the work is ceremony.
7. **Review findings are claims, not defects.** Give each reviewer a distinct lens, then validate every finding before filing or fixing it. Record the rejections; a durable refutation prevents the same false premise from returning.
8. **Green belongs to a tree, not a branch.** Two green branches can merge into a red `main`, and a green release workflow can publish over a red required check. Test the integrated revision and verify the whole required graph.
9. **When the evidence stops, the patch stops.** Reproduce against the real binary, measure the phase the claim names, and accept "missing artifact" as a completed result. Never let "flaky" close an investigation.
10. **Preserve what the user wrote.** When a system cannot represent input faithfully, it should show the source and the diagnostic, locatable and intact. Silent disappearance is never graceful, in an editor, a parser, a migration, or a fallback.
11. **Delegation is not autonomy.** An agent works to the end of a bounded unit and stops there, unless a completion harness defines done and makes premature

stopping difficult. Reserve the human for taste, product feel, external facts, and adversarial pressure, and let neither side certify its own work.

Appendix C: Sources and further reading

The practices in this book were worked out by people who faced the same problem, a fast and fallible producer of work, long before that producer was a language model. This appendix names the debts and points to the originals. Attributions were checked against primary sources before printing; where a claim could not be fully confirmed, the entry says so. Links point to the persistent identifier where one exists, otherwise to the canonical home of the work; entries without links are in print and easily found.

Tests that must be seen to fail

- Richard J. Lipton, "Fault Diagnosis of Computer Programs," student term paper, Carnegie Mellon University (1971). The origin of mutation testing.
- Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", IEEE Computer (1978). Established mutation testing as a discipline: seed small faults, then measure whether the test suite detects them. `doi.org/10.1109/C-M.1978.218136`
- Timothy Budd, "Mutation Analysis of Program Test Data", PhD thesis, Yale University (1980). The first implemented mutation-testing system. `gwern.net/doc/cs/algorithm/1980-budd.pdf`
- Gerald M. Weinberg, "The Psychology of Computer Programming" (1971). Introduced bebugging, seeding known defects to estimate how many unknown ones remain; also the first sustained study of programming as human work.
- Kent Beck, "Test-Driven Development: By Example" (2002). Codified the cycle in which a test is watched failing before the code that passes it is written.
- Yury Izrailevsky and Ariel Tseitlin, "The Netflix Simian Army", Netflix Tech Blog (2011). Extended deliberate failure injection to production systems. `techblog.netflix.com/2011/07/netflix-simian-army.html`

Oracles and disagreement

- Edsger W. Dijkstra, "Notes on Structured Programming", EWD249, written in 1969. The source of the principle that testing shows the presence of defects, never their absence. `www.cs.utexas.edu/~EWD/transcriptions/EWD02xx/EWD249/EWD249.html`
- Elaine J. Weyuker, "On Testing Non-testable Programs", The Computer Journal (1982). Defined the case where no oracle exists to judge whether output is correct, and what can be done about it. `doi.org/10.1093/comjnl/25.4.465`
- William E. Howden, "Theoretical and Empirical Studies of Program Testing", IEEE Transactions on Software Engineering (1978). Early formal analysis

of what a test result can establish; the origin of the test-oracle concept. doi.org/10.1109/TSE.1978.231514
- William M. McKeeman, "Differential Testing for Software", Digital Technical Journal (1998). Named and demonstrated differential testing: independent implementations compared on generated inputs. www.semanticscholar.org/paper/Differential-Testing-for-Software-McKeeman
- Xuejun Yang, Yang Chen, Eric Eide, and John Regehr, "Finding and Understanding Bugs in C Compilers", PLDI (2011). Demonstrated differential testing at scale; the Csmith generator exposed hundreds of defects in production compilers. users.cs.utah.edu/~regehr/papers/pldi11-preprint.pdf
- Koen Claessen and John Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs", ICFP (2000). Introduced property-based testing: stated properties checked against generated inputs. doi.org/10.1145/351240.351266
- Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo, "The Oracle Problem in Software Testing: A Survey", IEEE Transactions on Software Engineering (2015). The comprehensive account of the oracle problem and its partial solutions. doi.org/10.1109/TSE.2014.2372785

Generated proof

- Donald E. Knuth, "Literate Programming", The Computer Journal (1984). Program and documentation produced from a single source, so neither can drift alone. doi.org/10.1093/comjnl/27.2.97
- Ken Thompson, "Reflections on Trusting Trust", Turing Award lecture, Communications of the ACM (1984). Demonstrated that trusting a binary requires trusting its entire build lineage, the strongest form of the distinction between provenance and verification. doi.org/10.1145/358198.358210
- Tim Peters, the doctest module, Python standard library (1999). Made documentation examples executable tests. docs.python.org/3/library/doctest.html
- Jérémie Bobbio and the Debian Reproducible Builds project (2013 onward). Established bit-for-bit reproducibility as a verifiable property of a build system. reproducible-builds.org/docs/history

Boundaries and design

- David L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", Communications of the ACM (1972). Established information hiding as the criterion for decomposition: a module should hide a decision likely to change. doi.org/10.1145/361598.361623
- Michael E. Fagan, "Design and Code Inspections to Reduce Errors in Program

Development", IBM Systems Journal (1976). Introduced formal inspection with measured defect-removal rates, applied before implementation is complete. doi.org/10.1147/sj.153.0182

- Michael C. Feathers, "Working Effectively with Legacy Code" (2004). Defined the boundary concept this book uses throughout under other names: a place where behavior can be altered and observed without editing the code in place.
- Barry W. Boehm, "Software Engineering Economics" (1981). Quantified how the cost of correcting a defect grows with the phase in which it is found; the underlying analysis appears in "Software Engineering", IEEE Transactions on Computers (1976). selab.netlab.uky.edu/homepage/boehm-sw-eng-paper.pdf

Small batches and integration

- Kent Beck, "Extreme Programming Explained: Embrace Change" (1999). Argued small, continuously integrated changes as the core of development practice rather than a tactic.
- Martin Fowler and Matthew Foemmel, "Continuous Integration," martinfowler.co, (2000). Defined the practice of continuous integration as it is now understood. martinfowler.com/articles/originalContinuousIntegration.html
- Grady Booch, "Object-Oriented Analysis and Design with Applications," second edition (1994). Documents internal releases forming "a sort of continuous integration"; the phrase is often dated to the 1991 first edition, but the confirmed citation is the second.
- Donald G. Reinertsen, "The Principles of Product Development Flow" (2009). The economics of batch size and queues that underlie chapter 9.

Checklists, scars, and honest failure

- The Boeing Model 299 crash at Wright Field, October 30, 1935, and the pilot's checklist created in response. The founding case: a machine too complex to operate from memory alone. www.nationalmuseum.af.mil/Visit/Museum-Exhibits/Fact-Sheets/Display/Article/610002/model-299-crash
- Peter Pronovost and the Michigan Keystone ICU project, New England Journal of Medicine (2006). A five-item checklist produced a large, sustained reduction in catheter-related infections in intensive care. doi.org/10.1056/NEJMoa061115
- Atul Gawande, "The Checklist Manifesto: How to Get Things Right, (2009). Generalized the checklist as a safety instrument across medicine, aviation, and construction. atulgawande.com/book/the-checklist-manifesto
- The NASA Aviation Safety Reporting System (1976 onward). A confidential,

non-punitive incident reporting system; the model for blameless failure reporting. `asrs.arc.nasa.gov`
- Sidney Dekker, "Just Culture: Balancing Safety and Accountability" (2007). Argued that punishing error reporting suppresses the information safety depends on.
- John Allspaw, "Blameless PostMortems and a Just Culture," Etsy Code as Craf, (2012). Brought blameless postmortem practice to web operations. `www.etsy.com/codeascraft/blameless-postmortems`
- John Lunney and Sue Lueder, "Postmortem Culture: Learning from Failure," Site Reliability Engineering, chapter 1, (2016). Documented the practice as standard operations at scale. `sre.google/sre-book/postmortem-culture`

The division of labor

- Charles Babbage, "On the Economy of Machinery and Manufactures, (1832). Analyzed the division of mental labor in computation a century before electronic computers. `archive.org/details/oneconomyofmachi00babb`
- Ada Lovelace, Note G to her translation of Menabrea's "Sketch of the Analytical Engine" (1843). Distinguished what the engine could do from what remains the operator's judgment. `www.cs.yale.edu/homes/tap/Files/ada-lovelace-notes.html`
- J. C. R. Licklider, "Man-Computer Symbiosis", IRE Transactions on Human Factors in Electronics (1960). Proposed the human-computer partnership in which each side contributes what it does best. `groups.csail.mit.edu/medg/people/psz/Licklider.html`
- Laurie Williams, Robert Kessler, Ward Cunningham, and Ron Jeffries, "Strengthening the Case for Pair Programming", IEEE Software (2000). Measured the effect of two people sharing one problem, neither certifying their own work. `doi.org/10.1109/52.854064`

If you read only three: Parnas (1972), Dijkstra's EWD249, and Licklider (1960). Everything in this book is downstream of somebody, and mostly of them.

Appendix D: Index of subjects

Numbers are chapters and letters are appendices; both stay put across editions, which page numbers do not. The locations below are found in the text on every build, so the index cannot drift from the book.