

The AI Workflow Playbook

Working with LLMs, AI coding agents and multi-agent builds, based on the method Domenic Laurenzi actually uses, including where it failed.

Companion document: `RUNBOOK.md` (the step-by-step procedures). This playbook is the WHY and the WHAT. When a pattern ends with "Do it: R6", go to procedure R6 in the runbook for the steps.

About this document

What it is. A set of 17 patterns (P1 to P17), a decision guide, an honest section on costs, and a glossary. Most patterns exist because something went wrong first. The incidents are real. They come from three research digests (A, A2 and B) compiled from the author's own repositories, configuration and records. Those digests are private working notes and are not published, because they cite private files; the citations are kept so every claim still shows where it came from. Where a number appears, it carries the digest's own label:

- **MEASURED**: a researcher counted or read it directly from a file.
- **DOC-STATED**: a document says so; the underlying claim was not re-verified.
- **INFERRED**: an interpretation. Treat it as a hypothesis.

Evidence citations take the form "(digest-B P1.5; source: Quartermaster handoff 2026-08-19 line 296; **MEASURED**)". The digest item names the entry in the private notes; the source names the project file it was read from. Line numbers for the orchestration project's 2026-08-19 handoff refer to commit d0acdb7; a later commit (2026-09-26) annotated that file, which shifts lines by 7 to 8. Evidence corrected by the lead or the independent review after the digests were written is cited as "lead's correction" or "review RV-n".

What it is not. A productivity pitch. Nothing here claims the method makes you faster, and no such measurement exists in the sources. What the sources do show is that each pattern caught real defects, and that each has a real cost. Both are reported.

Tools. The principles are tool-agnostic. The worked examples use Claude Code (Anthropic's command-line coding agent) because that is what the method was built on. Where a second tool appears, it is Codex CLI, which the method runs from the same rulebook and memory. Any agent that can read standing instructions, run tools, and start helper agents can apply these patterns.

Privacy. Client names, hostnames, addresses, account identifiers and personal contact details are removed. Project names that belong to Domenic's own products are kept where they help you follow a story. Short code names used in citations:

Code	What it is
HE	An embedded-firmware analysis product built in milestones M0 to M6
QM	"Quartermaster", a multi-provider LLM orchestration tool built by six agent teams
HD	"HADES", a released security product with published binaries
UT	A USB triage tool with a deferral register
JH	A job-search project whose claims ledger feeds resumes and applications
FA	A cross-portfolio audit written 2026-09-26

How to read this

Pick the track that fits you. Each lists the patterns to read first; read the rest when you need them.

Track 1: New to this

You have used a chat assistant, maybe tried an agent once, and some words here are unfamiliar. Start with the glossary entries for **context window**, **agent**, **subagent**, **hook** and **test suite**, then read:

1. **The whole method in ten lines** (next section).
2. **P1 Verify before you report.** The single most important habit.
3. **P7 Plan first; re-plan when it goes sideways.**
4. **P6 Name the ambiguity.**
5. **P8 Corrections become rules.**
6. **P13 Guardrails and human-in-the-loop.** Read before you let an agent run anything on its own.
7. **What this costs, and when not to use it.**

Skip the "In the trenches" parts on a first read. The "In plain terms" part of every pattern is written for you.

Track 2: Practitioner

You use an agent daily, have a project instructions file, and have started using subagents. Read:

1. **P2 A green test suite is a claim.**
2. **P5 Fan out, then re-verify first-hand.**
3. **P9 Memory.**
4. **P12 Change the mechanism, not the wording.**
5. **P14 Verify the environment before blaming the code.**
6. **P15 Scope boundaries and handoffs.**
7. **Decision guide.**

Track 3: Veteran

You run several agents at once, overnight, on real products. Read:

1. **P4 The auditor is not the builder.**

2. **P3 Make the evidence generate itself.**
 3. **P10 A deferral is an obligation with enforcement.**
 4. **P16 A claims ledger.**
 5. **P11 Decisions are artifacts.**
 6. **P17 Know the costs** and the full costs section. Much of it is about failure modes that only appear at scale: orphaned processes, crashed consolidations, parallel agents sweeping each other's work into commits, and guards that enforce nothing while appearing installed.
 7. **Before any unattended or overnight run, read "Unattended runs" in P13.** Modes that skip permission checks remove most of the guardrails this document describes.
-

The whole method in ten lines

1. **Every claim is measured, read, or labelled as inferred.** There is no fourth kind. A count is not a rate unless the denominator came from the same measurement. (P1)
2. **A passing test proves only what the test checks.** Break the fix on purpose and confirm a test goes red; if nothing does, the fix has no test. (P2)
3. **Let the check write the record.** Evidence that a script generates cannot drift from what the script saw. (P3)
4. **Whoever built it does not get to audit it.** Self-audit is how defects pass. (P4)
5. **Use helper agents to go wide, then re-read everything they report yourself** before it lands anywhere that matters. (P5)
6. **When words admit two readings, quote them and say so.** If you must choose, choose the reading that means more work or worse news for you. (P6)
7. **Plan before non-trivial work; after three failed tries of the same idea, stop and re-plan.** (P7)
8. **Every correction becomes a written rule, and the important rules become mechanisms** because written rules alone were shown not to hold. A mechanism is a **hook** (a script the tool runs automatically at a fixed moment), a **gate** (a check that must pass before a milestone counts as done) or a test. (P8, P12)
9. **Anything you put off must name when it is due, and a gate must refuse to pass while it is still open.** (P10)
10. **The agent decides inside its lane and stops at the boundary:** anything outward facing, irreversible, out of scope, or touching secrets, money or regulated data waits for a human. (P13, P15)

The rest of this document explains why each line exists, what it caught, and what it costs.

The patterns at a glance

ID	Pattern	Core question it answers	Do it
P1	Verify before you report	"Did I check this, or am I narrating?"	R1, R6, R12
P2	A green test suite is a claim	"Would any test notice if this fix were gone?"	R9, R8
P3	Make the evidence generate itself	"Could this record disagree with what actually ran?"	R6, R10, R12
P4	The auditor is not the builder	"Who checked this, and did they write it?"	R8
P5	Fan out, then re-verify first-hand	"Have I read the thing the helper told me about?"	R7
P6	Name the ambiguity	"Is there another reading of these words?"	R6
P7	Plan first; re-plan when it goes sideways	"Am I on attempt four of the same idea?"	R6, R14
P8	Corrections become rules	"Where is this mistake written down so it cannot recur?"	R2, R3, R15
P9	Memory: one fact per file, indexed, shared	"Will the next session know this?"	R4, R13, R15
P10	A deferral is an obligation with enforcement	"What will force this 'later' to happen?"	R10
P11	Decisions are artifacts	"Where is the reason this was chosen?"	R6, R11
P12	Change the mechanism, not the wording	"Did the rule fail, or did nobody enforce it?"	R2, R5
P13	Guardrails and human-in-the-loop	"Should an agent be allowed to do this alone?"	R0, R5, R14
P14	Verify the environment before blaming the code	"Is the plumbing proven before I trust the result?"	R0, R14, R6
P15	Scope boundaries and handoffs	"Am I allowed to touch this, and could a crash erase my notes?"	R11, R8

ID	Pattern	Core question it answers	Do it
P16	A claims ledger	"Can every public number be traced to a measurement?"	R12
P17	Know the costs	"Is this pattern worth it for this task?"	R15

Part 1: Truth and evidence

These five patterns are about one problem: an agent (or a person) believing it checked something it did not. They come first because every other pattern depends on them.

P1 Verify before you report

In one sentence

Every factual statement you hand someone is either measured now, read from a file now, or explicitly labelled as inferred; and a count is never a rate unless the denominator came from the same measurement.

In plain terms

When an agent tells you "the tests pass" or "14 of 20 files have the bug", you naturally assume it looked. Often it did not. It remembered, guessed, or combined two numbers that had nothing to do with each other. Think of a mechanic who says "your brakes are fine" without lifting the car: the sentence sounds exactly the same as if he had checked.

The rule is that every claim falls into one of three bins:

- **Measured:** "I ran the count just now and got 14."
- **Read:** "The report file says 14."
- **Inferred:** "I think it is about 14, based on X." (labelled as such)

There is no fourth bin. "It should be", "that means" and "so it is" are guesses dressed up as facts.

A **count** is how many of something. A **rate** is that count divided by a total (the **denominator**): "3 of 20" or "15%". Tool output usually gives you counts. When you write a rate, you need a total, and the nearest number lying around is tempting. If that total came from a different question, the rate is invented.

Why it exists

On 2026-08-23, in the embedded-firmware analysis project (HE), a finding count was reported as "fourteen hits from one memory-safety rule on twenty binaries". The "twenty" came from a different query that counted analyzed binaries; nothing related the two numbers. The invented rate (70%) produced the conclusion "that is exactly what an over-firing rule looks like", which was raised to the owner as the first thing to review. The real figure, per the rule's own account (**DOC-STATED**); the gate record itself holds the 14 and the 20 but not the 3), was 3 of 20, and the rule was working correctly. The correct query was one step away the whole time. (digest-A 1.6; global instructions, First Principle 4.)

The story has a second twist, and it happened while this playbook was being researched. A researcher re-read the milestone gate record, found it now reads **BIN-MEM-0001: 11** , and concluded that the "14" in the cautionary tale was an error in the rulebook (digest-B F5.8).

The lead then checked the file's git history. The record said `BIN-MEM-0001: 14` in every gate run from 2026-08-22 22:37 through 2026-08-23 09:45, and a commit at 11:47 on 2026-08-23 changed it to 11. The incident is dated 2026-08-23 and quoted the record accurately as it stood then. The researcher had compared a dated claim against today's version of a file that gets regenerated: the same trap P1 exists to catch, committed while checking P1's own example. (Lead's correction to digest-B F5.8, verified in git; the digest item itself is superseded.)

A second incident, from a disk-erase tool: an antivirus blocker was "recorded as Defender, on no evidence ... stated to Domenic several times, along with remediation advice aimed at Microsoft, before it was checked". A different vendor's real-time protection was the only one enabled, and the query that proved it "was one command away the entire time". (digest-B F5.9; source: EraseDrive lessons 2026-09-10; **MEASURED**.)

How it works

1. **Sort every claim into a bin before you write it.** If it is not measured or read this session, label it: "I think", "inferred", "not re-verified".
2. **Catch the syntactic tells.** You are about to write "N of M", "X%", "most", "few", "only", "that's a lot". Stop and ask where M came from. If it did not come out of the same measurement as N, report the bare count: "14 findings" is always safe; "14 of 20" needs a real 20.
3. **Read back any state you changed.** An edit that returned no error is not an edit you confirmed. A service that started is not a service that answered.
4. **Treat a cause as a hypothesis until reproduced.** "Why it happened" is the claim most often reasoned rather than measured.
5. **If the check is one command away, run it** instead of narrating. The command costs seconds; the retraction costs trust.
6. **Put units next to numbers.** A count printed next to a claim gets read back as part of it. (digest-A 7.9.)

In the trenches

- **A fabricated denominator is never one error.** Every judgement downstream of it inherits the fabrication, and each looks independently reasonable. Fixing the number means re-examining everything concluded from it, not just the sentence it sat in.
- **Ground truth beats the convenient source.** The bytes on disk, the export, the actual file outrank an API's return value, a summary, or your memory. In a hardware project, a net was reported renamed "because the API said so while the export said otherwise". (digest-A 1.5.)
- **"A pass count is an environment fact."** A suite that expects "292 passed, 4 skipped" produced "291 passed, 5 skipped" on a machine missing one optional package; "Skips are

the load-bearing number". (digest-B P4.4 item 9.)

- **Settle disputes with transcripts, not testimony.** An attribution question across machines was "settled by grepping transcripts for the tool call matching the artefact's timestamp". Shell history never records agent calls, "so its emptiness is a non-probe, not an alibi". (digest-A 7.10.)
- **Interaction:** P1 is the input to P16 (the claims ledger), which applies it to every public statement, and to P5, which applies it to helper-agent reports.
- **Cost:** mostly seconds per claim. The real cost is that it forces you to say "I did not check" out loud more often than feels comfortable.

Evidence

- The researcher's dated-evidence mistake above, settled from git history (lead's correction to digest-B F5.8).
- A disk-erase tool's impact claim "every customer who downloads this will hit this" was "asserted rather than measured" (digest-B F5.9; EraseDrive lessons; **MEASURED**).
- The cross-portfolio audit states its own evidence class up front: "No test suite was run, so no pass count here is a measurement." (digest-B P1.9; source: FA lines 8-15; **MEASURED**.)

Do it

R1 (first session), R6 (the per-task loop, verify step), R12 (claims ledger).

P2 A green test suite is a claim: test the tests

In one sentence

A passing test suite proves only what its tests actually check, so for anything that guards safety, permissions, money or data leaving a system, deliberately break the fix and confirm that at least one test fails.

In plain terms

A **test suite** is the set of automated checks that run against your code. **Green** means they all passed. It is natural to read "green" as "correct". It is not. A smoke alarm with a dead battery is silent too, and silence looks the same as safety.

The method's answer is the **un-fix**: take a fix you just made, undo it on purpose, and run the tests again. If they still pass, no test was checking the fix. The formal name for this family of techniques is **mutation testing**: introduce small bugs on purpose and see whether the tests notice.

Why it exists

On 2026-08-18 a phase of the orchestration project (QM) was declared done on a green build: 359 tests passing. A post-completion audit then found three real defects. One was an allow/deny filter that widened access instead of narrowing it, because "unconstrained" was stored as an empty set, the same representation as "nothing". One was a token budget that never tripped because unreported counts defaulted to zero. One was a config walker that could not enter certain fields. One of the three "silently disabled a data-egress control" (the rule limiting what data can leave the system). The global instructions record: "Intent was never the problem. Verification was." (digest-A 1.4; project attribution per the global memory entry on auditing before declaring done, confirmed by the lead, review RV-1.)

In the firmware project (HE), a separate audit found that a single environment variable switched the analysis image to full network access; the health check "reported `ok: true`". Every gate run had passed the no-network flag itself, so "the gate proved the control works when the operator already did the right thing". Rule: "a control whose strictness is selected by mutable input is not a control." (digest-B P1.12; source: HE ADR-0025, an architecture decision record, see P11; **MEASURED**.)

How it works

1. **For each security or safety fix, un-fix it** (revert the source change) and run the suite. At least one test must go red. If none does, the fix has no test; write one and confirm it goes red against the un-fixed code before keeping it.
2. **Add a negative control to the harness itself:** remove a control you know is covered and confirm the harness reports it. This proves the harness can detect anything at all.
3. **Pair every "rejects the bad input" test with a positive control** that stays green on purpose, so a "fix" that refuses everything cannot pass.
4. **Assert invariants over a cross-product of inputs**, not one illustrative example. Ask what the code does when inputs are disjoint, empty, absent, unreported or zero-length.
5. **Where a special value means "unset", prove it cannot be confused with a real empty or zero.** If it can, fix the type, not the call site. (digest-A 1.4 step 4.)
6. **Ask of every PASS: what input would make this fail, and is the input I used degenerate?** (digest-A 7.4.)

In the trenches

- **The recurring shape of a test that cannot fail.** Across four teams in one night: "a test that ADDS an element rather than CHANGING one (so it exercises the cardinality rule, not the value rule), or one that measures the same artifact the change introduced (so it compares a number against its own announcement)." (digest-B P3.2; QM handoff lines 66-70; **MEASURED**.)
- **A test that plants its own premise cannot falsify it.** A test wrote a value into a probe key "from the same assumption the code was written from"; the original code had been correct and "my comment was a monument to the bug". Synthetic fixtures "matched rules because they were synthesized to match rules". (digest-B P4.4 item 3.)
- **A loop over an empty container is the quietest possible pass.** Three gates iterated a deferral list that was empty, so they "run zero assertions and report green". (digest-B P4.2 counter-example; Metadata-Scrubber lessons 2026-09-04; **MEASURED**.)
- **A proof that cannot fail is not a proof.** A backup restore check picked a zero-byte file, compared two empty-string hashes, and printed "RESTORE PROVEN". The same weakness sat in the machine's security gate. (digest-A 7.4.)
- **A structural check is not a content check.** A disk image passed its validator, matched the expected size, and contained the wrong thing, because a regex over JSON matched a nested key. Parse structured output with a parser. (digest-A 7.5.)
- **Offline suites have a boundary.** "The fake transport accepts whatever it is handed." Eight provider adapters passed a conformance suite offline; the first live call found four defects, including one where "every structured-output call to that provider has always failed". (digest-B P1.8; QM ADR-0065; **MEASURED**.) Treat the first contact with a real system as its own test campaign.

- **Un-fixing is not enough on its own.** Builders' harnesses still missed defects that an independent reader found; see P4.
- **Cost:** time per fix, and a growing test count. It is worth spending on controls that guard egress, permissions, spend or safety; for cosmetic changes it usually is not.

Evidence

What the green suite hid	Source (digest-B)
<code>no-new-privileges:false</code> accepted by a sandbox guard because guard and test both used <code>startswith</code> ; un-fixing "left the ENTIRE suite green"	P1.5, QM handoff line 296, MEASURED
An HTTP endpoint returning 500 on every call; the test accepted <code>status_code in (200, 500)</code> ; "The suite was green the entire time"	F5.5, HD lessons, MEASURED
Detection rules passing 6/6 on synthesized fixtures while "only 2 of 333" real confirmed-malicious files moved to actionable	F5.5, HD lessons, MEASURED
"282 tests passed. Every one of them ran from" a folder on an antivirus exclusion; the first run elsewhere was blocked	F5.5, EraseDrive lessons, MEASURED
A test asserting a string that "has never existed in that file, so the test never passed once"	F5.5, HD check-my-work handoff, MEASURED
A real fix whose five-line revert left "the whole suite still green"	P1.10, QM handoff lines 131-132, MEASURED

The un-fix audit lane that found the sandbox defect "reported 8 confirmed findings, every one proven by un-fixing the source", and "its own negative control (removing `--cap-drop ALL`) went red, so the method detects a covered control". A later round: "Nine un-fixes, nine red." (digest-B P1.5; QM handoff **MEASURED**, PROGRESS **DOC-STATED**.)

Do it

R9 (un-fix harness), R8 (independent audit of a change).

P3 Make the evidence generate itself

In one sentence

Records that prove a milestone passed, or that a document's numbers are right, should be written by the script that checked them, never transcribed by hand.

In plain terms

If a person (or an agent) copies test results into a report, the report can drift from what actually ran: a number gets rounded, a skipped check gets forgotten, a later run is not copied across. If the check itself writes the report, embedding its raw output, the report cannot say anything the check did not see. It is the difference between a receipt printed by the till and a receipt written out by the cashier.

A **gate** is an automated check that must pass before a milestone counts as done.

Why it exists

In the orchestration project (QM), a review team found that "Eleven of the twelve reported prose claims" in the project's documents were false (**DOC-STATED**). One described a behaviour the code did not have: "make probe wrote a file every engine ignored", so a disqualifier "never fired". (digest-B P1.13; QM handoff lines 72-82.)

In the job-search project (JH), a 2026-09-03 accuracy review corrected three factual errors in the documents a person opens, but the data file that actually generates every tailored application "went unfixed for nine days. ... Nothing looked wrong, because the documents a human opens were correct and only the generator was stale." (digest-B P1.3; source: JH `check_ledger.py` docstring; **MEASURED**.)

How it works

1. **Each gate is a script that writes its own record**, embedding raw console output and pinning its inputs (image, firmware hash, snapshot id) in the header. The HE gate records open with: "Every block below is real output from this run; nothing here is transcribed by hand."
2. **Read limits from the product, not from the gate**. "A gate carrying its own copy of a limit is a gate that can pass while the product enforces a different number." (digest-B P1.7.)
3. **Make counted claims mechanical**. A test reads the artifact, renders the exact sentence the document must contain, and asserts that string. "No regex extracts a number from prose, because such a pattern goes red on a harmless rewrite and green on a harmful one." (digest-B P1.13.)

4. **Check the generator, not the output**, and check every string in it against the rule list: "a rule list crossed with the whole document". Exit non-zero so it can block a commit or build. (digest-B P1.3.)
5. **Guard the parser that feeds a guard**. If an empty parse result means "no violations", the guard must prove the parse was not empty, using an independent count "so a regex regression cannot also blind this check". (digest-B P1.4.)

In the trenches

- **A generated record can still say PASSED with a criterion skipped**. HE milestone M0 shows criterion 6 as SKIPPED and the gate as PASSED. The cross-portfolio audit flagged it. A generator removes transcription error; it does not decide what counts as a pass. (digest-B P1.7, F5.10; MEASURED.)
- **A guard can encode a wrong fact as firmly as a document can**. The ledger checker banned the correct competition placement "and so failed resume_data.json for being right" until it was noticed. (digest-B P1.3; MEASURED.)
- **"An empty list and a clean result are indistinguishable downstream."** A renumbering made one ledger parser match nothing, so "every Never-say warning went dark: 0 phrases, indistinguishable from 0 violations". The guard's own first version "missed bug 2 entirely and passed green"; mutation testing (P2) exposed it. (digest-B P1.4.)
- **A gate that fails on success trains people to ignore it**. Read expected values from the source tree rather than hardcoding them. (digest-B P1.14.)
- **A regenerated record changes over time**. When you cite one, cite the commit or date; when you check a dated claim against it, check the version that existed on that date. (The P1 story shows what happens otherwise.)
- **Cost**: some scripting per gate, plus the discipline of never editing a generated record by hand. Generated records are long (the HE gate records run 192 to 551 lines each, MEASURED), which is fine: they are evidence, not reading material.

Evidence

- HE milestone M5 asserts reproducibility with generated output: "finding ids: run A 21, run B 21, only in A 0, only in B 0 ... observations present in both: 1032, content hash differs on: 0". (digest-B P1.7; HE gate M5; MEASURED.)
- The counted-claims test: "25 un-fixes, 25 red." (digest-B P1.13; QM handoff line 92; DOC-STATED.)
- Because gate records are generated and committed, the disputed figure in P1 could be settled from git history rather than recollection (lead's correction to digest-B F5.8).

Do it

R6 (verify step), R10 (deferral register with a failing gate), R12 (claims ledger).

P4 The auditor is not the builder

In one sentence

No agent (or person) audits a component it wrote; a milestone passes only when the suite is green, an independent reviewer has audited the boundaries without simply re-running the suite, and the progress record is updated in the same commit.

In plain terms

You cannot proofread your own essay well, because you read what you meant rather than what you wrote. Agents have the same problem, and worse: an agent that wrote code from an assumption will test it from the same assumption. So the reviewer must be someone (usually a fresh agent) that did not write the work and has no stake in it passing.

Why it exists

At milestone M1 of the firmware project, "an adversarial reviewer that had written none of the code found nine defects, two of which were introduced by the fixes for the first four. The engineers who built and self-tested their own components reported clean. ... Self-audit is how those passed." (digest-B P1.10; source: HE ADR-0027; **MEASURED**.)

How it works

1. **A gate has three parts:** suite green; a boundary audit by an agent that did not build the component and "may not discharge the audit by re-running the suite"; progress document updated in the same commit.
2. **Assign reviewers so that nobody reviews their own component.** "Corollary for team assignment: no agent audits a component it wrote."
3. **Give the auditor the question, not the conclusion.** The method's "check my work" handoffs say: "Your judgement is the output here. Nothing below is a recommendation", followed by "Places I am most likely to be wrong. Check these hardest." (digest-B P3.6.)
4. **Audit the fix, not just the defect.** Fixes introduce defects too; send the fixes back through an independent audit.

In the trenches

- **"An audit that never finds anything is not evidence of quality, it is evidence the audit is not looking."** (HE ADR-0027.) A clean audit on a large change is a reason to ask what the auditor checked.

- **Fixes regress.** The orchestration project's ninth audit named four merge-blocking defects, "three of them in code written to harden exactly that area", and the second pass on the fixes found "a real fix with no test that could fail". (digest-B P1.10; QM handoff lines 122-132; **MEASURED**.)
- **A fresh session removes shared context, not shared training.** A new session of the same model starts without the builder's conversation, but it has the same training and may share the same blind spots. For high-stakes audits, prefer a different model or vendor, or a human, and record which you used. (Review RV-19; this is a design caution, not a recorded incident.)
- **A contrary report deserves a ground-truth check, not a dismissal.** A helper agent in a clean workspace reported a crypto source file "does not exist and never has". It "read as obviously wrong" after three local builds and twelve green suites. It was right: an ignore rule had left seven files untracked, and the pushed code "could not be compiled by anybody else". "Ground truth was `git ls-files`, not the build I had watched succeed." (digest-B P3.8; HexHound lessons 2026-08-03; **MEASURED**.) An independent reviewer is valuable precisely because it does not share your environment.
- **Interaction with P2:** the un-fix harness run by builders catches some cannot-fail tests; the independent auditor catches what the builders' harnesses do not. Both were needed in the recorded runs.
- **Cost:** "It roughly doubles the review effort per milestone and it will keep finding things, which is the point." (HE ADR-0027; **DOC-STATED**.)

Evidence

- Nine defects found by a non-builder at M1 while builders reported clean (digest-B P1.10; HE ADR-0027; **MEASURED**).
- "Six teams each ran their own un-fix harness, each caught one of its own tests that could not fail, and all six still left four defects that a reader with no stake in the work found in one pass. Then the fixes for those four needed their own audit, which found three more. Audit the fix, not just the defect." (digest-B P3.2; QM handoff lines 138-141; **MEASURED**.)
- The rule spread sideways: ADR-0027 says it was "Adopted from a sibling DarkHorse project" (digest-B P4.3; **DOC-STATED**).

Do it

R8 (independent audit of a change).

P5 Fan out, then re-verify first-hand

In one sentence

Use helper agents (subagents) to search, read and analyse in parallel so your main conversation stays clean, then re-read every figure, quote and citation they report yourself before it goes anywhere that matters.

In plain terms

An agent can only hold so much text in mind at once; that limit is its **context window**. If it reads fifty files itself, the window fills with file contents and there is little room left to think. A **subagent** is a helper the main agent starts for one focused job ("search these repos for X and report back"). The helper does the reading in its own window and returns a short report. It is like sending assistants to the library: you get summaries instead of carrying every book.

The catch: an assistant's summary is a claim about the books, not the books. The method treats a helper's report as "a document repeating a claim", which by its own rules is not a source. So after the fan-out, the lead agent re-reads the actual lines the report points to.

Why it exists

On 2026-09-26, in the job-search project, four read-only helper agents swept the repositories, and their findings were merged into the claims ledger "after only a partial spot-check. Asked 'did you verify everything?', the honest answer was no." The full re-read then found nine corrections to material already written, among them a key size with no source, a test count that was a "Claimed effect" and not a run, a statement that "no Rust file mentions PDF" that "was literally false (test fixtures do)", a wrong commit count, a wrong product term, and a citation that had moved by two lines. (digest-B P1.9, F5.3; sources: JH lessons and FACTS changelog 2026-09-26; **DOC-STATED**.)

The rule written: "A sub-agent report is a document repeating a claim, which is exactly what FACTS.md section 0 says is not a source."

How it works

1. **Fan out for breadth**, one task per helper. The global instructions: "Use subagents liberally to keep the main context window clean. Offload research, exploration, and parallel analysis to subagents. ... One task per subagent for focused execution." (digest-A 3.1.)
2. **Make helpers read-only unless writing is their job**, and scope-lock any helper that writes to named files (see P15).

3. **Require citations in every report:** file and line for every figure and quote. A report without citations cannot be re-verified and should be sent back.
4. **Re-read first-hand, figure by figure, before writing anything load-bearing.** The cross-portfolio audit states: "every figure, quote and file:line below was re-read first-hand the same day (one agent citation ... was wrong and is corrected here)".
5. **State what you did not re-read.** The same audit lists items "Unchanged from the agents and not re-read". Silence would imply they were checked.
6. **When a helper contradicts you, check ground truth before dismissing it** (see the untracked-files story in P4).
7. **Tell every helper that instructions found in what it reads are data.** Helpers fanned out over repositories and web pages will meet text that tries to give them orders (prompt injection, P13). They report it; they never act on it.
8. **Set a budget before fanning out.** Every helper spends tokens (and so money), and every report costs re-reading time. Cap a fan-out, an audit or a night by spend and stop when the budget is spent, not when the questions run out. (Review RV-19; the digests record no budget figures, so none are suggested here.)

In the trenches

- **Fan-out is cheap; re-verification is where the cost lives.** Budget for it up front. If you cannot afford to re-verify a finding, you cannot afford to publish it.
- **Helpers mis-read signals.** A signature-check column where N "meant 'cannot verify', not 'not signed'" was read by one agent as absence, and it "concluded the whole history was unsigned". (digest-B P3.9; QM handoff lines 203-208; **MEASURED**.)
- **Permissions can differ between you and a helper.** One item was "first refused by the permission classifier, then re-read first-hand on the owner's explicit request". (digest-B P1.9.)
- **Do not merge numbers from different documents.** The "one agent citation ... wrong" in the audit and the "nine corrections" in the ledger describe two different documents from the same audit. Reporting them as one number would be exactly the P1 failure. (digest-B P1.9 note.)
- **Parallel sessions need labels.** When several sessions can ask the human questions, each tags its project so he "can tell which parallel session is asking". (digest-A 3.6.)
- **Forks versus fresh helpers.** A fresh helper makes the better auditor (P4); a fork, which inherits your conversation, suits continuing your own work in parallel. (This playbook's inference from P4.)
- **Crash recovery:** helper transcripts are written to disk as they go, so their reports can survive a crash of the parent. See P15.

Evidence

- Nine corrections found by first-hand re-verification of helper output already written into the ledger (digest-B P1.9; JH FACTS changelog 2026-09-26; **DOC-STATED**).
- One wrong helper citation found and corrected in the cross-portfolio audit (digest-B P1.9; FA lines 8-15; **MEASURED**).
- A helper's contrary report was right and caught an un-buildable push (digest-B P3.8; **MEASURED**).
- Scale of use, from a May 2026 self-assessment: "1,120 total subagent .jsonl files" (digest-A 3.1; that document's figure, not re-measured). This shows heavy use, not a success rate.

Do it

R7 (research fan-out with subagents plus first-hand re-verification).

Part 2: Thinking, planning and learning

These patterns are about how the work is framed before it starts, and how mistakes turn into lasting improvements after it ends.

P6 Name the ambiguity: quote, do not paraphrase; take the worse-news reading

In one sentence

When anything load-bearing can be read two ways, quote the exact words, say out loud that it is ambiguous, and either ask or default to the reading that means more work or worse news for you.

In plain terms

People (and agents) summarise as they read. The moment you restate someone's words in your own, you have picked one meaning, and the other meaning disappears from view.

"The contractor is booked" can mean "we booked the contractor" or "the contractor is fully booked and unavailable". Once your head has rewritten it as the first, every plan you build inherits the mistake, and nothing downstream looks wrong.

The fix is cheap: copy the exact words, notice there are two readings, and say so. Naming it costs one line. Guessing wrong can cost the whole plan.

Why the "worse news" default? Because when two readings differ in how much work they imply, people drift toward the lighter one without noticing. Defaulting the other way cancels the drift.

Why it exists

On 2026-07-17, in a hardware badge project, a message said a freelancer was "BOOKED". It was read as "hired" rather than "booked solid and unavailable", and "an entire session's plan" was built around outsourcing a circuit-board layout to someone who was never available. The same session claimed a net had been renamed "because the API said so while the export said otherwise", and reported PASS tables while "the actual deliverable had not moved at all". (digest-A 1.5; global instructions, First Principle 3.)

How it works

1. **Quote, do not paraphrase**, anything a plan depends on: a requirement, a status, a person's availability, a scope limit. If you cannot quote it, you do not know it.
2. **List both readings**. One line each.
3. **Ask, or take the reading that means more work or worse news**, and say which you took.
4. **Read the spec critically before building from it**. A prompt, ticket or build brief "is a draft, not a contract". Look for an in-scope component with no acceptance criteria, an invariant

nothing enforces, a decision left implicit, "a name that admits two readings". Report what you found before you build. (digest-A 1.4 step 1.)

5. **When two sources disagree, name the conflict** and say which you trust and why. Ground truth (the output file, the bytes on disk) beats a convenient source (an API's return value, a summary, your memory).
6. **Lead with the thing that was asked for and its real state**, before anything you did that was not asked for.

In the trenches

- **Tells you are doing it right now:** you are about to paraphrase; two readings differ in effort and you drifted to the lighter one; you cited only the agreeable source; you are reporting PASS on work that did not move the deliverable; a one-line memory summary is your only source. (digest-A 1.5.)
- **Specs written in prose can be wrong in code.** An architecture sentence said `side_effect_level > READ_ONLY`. On the data type used, that compares strings (the levels were compared alphabetically, not by severity), so every level including `DESTRUCTIVE` compared `False`. "A faithful transcription of section 15's sentence produces a sampler that skips nothing, and the thing it fails to skip is the task that changes the world." This was measured before the module was written. (digest-B P2.2; QM ADR-0050; **MEASURED**.)
- **Your own instructions can be ambiguous.** The global rulebook introduces "These five steps" and then lists six. "A literal-minded reader, human or model, can stop at five." (digest-A2 9.12.)
- **Asking is right even when the answer surprises you.** On a disk-erase tool, "Remove it from the LAN" turned out to mean the device, never the user. "Asking the question was right; the answer reframed the requirement". The unanswered half was "Flagged in the plan rather than silently chosen, and recorded so a future session does not 'discover' the gap". (digest-B P4.5; EraseDrive lessons 2026-09-08; **MEASURED**.)
- **Tension with autonomy.** Asking about everything is its own failure (see P7 and the decision guide). Name the ambiguity always; ask only when it is load-bearing and not yours to decide.
- **Cost:** one line per ambiguity. Occasionally, a question to a human.

Evidence

- The spec check that caught a sampler which would have skipped nothing, before any code existed (digest-B P2.2; **MEASURED**).
- The LAN requirement, reframed by asking rather than guessing (digest-B P4.5; **MEASURED**).
- The agent-memory bridge incident was labelled by the method itself as "the self-serving reading" (digest-A 1.5, reused in a memory entry).

Do it

R6 (the per-task loop: plan step).

P7 Plan first; re-plan when it goes sideways

In one sentence

For any non-trivial task, investigate and write a plan before changing anything; if the work goes sideways, stop and re-plan; and after three failed attempts at the same general approach, stop, record why each failed, and pivot to a fundamentally different one.

In plain terms

Agents are eager. Given a task, they start editing files at once. For small, clear jobs that is fine. For anything with three or more steps, or a design choice, it leads to a pile of changes built on an unchecked assumption.

Planning first means: read, investigate, write down what you intend to do and how you will know it worked, and get a check-in before building. **Plan mode** in Claude Code is a mode where the agent can only investigate and write a plan; it cannot change files until the plan is approved. Other tools have similar "ask before editing" settings; a written plan in a `todo.md` file works with any tool.

The **three strikes rule** is for when things go wrong: if the same general idea has failed three times, a fourth try with small tweaks is probably wasted. Stop, write down what you tried and why each failed, and come at the problem from a different angle.

Why it exists

A plausible explanation feels like a diagnosis. On 2026-08-27/28, three real defects in an agent hook were each reported as the cause of a failure; none was. "It burned one of his evenings and part of a night." The status of that investigation still reads root cause "UNKNOWN" and "The next step is a BISECT." (digest-A 7.6.)

In the firmware project, a tool failure was recorded as "killed by the OOM killer" (the operating system's out-of-memory killer) and "asserted three times on no evidence beyond a resident-memory figure and an exit code of -1". It was a four-hour timeout: the tool "started at 07:08 local and the container ended at 11:08. Four hours, to the minute." Cost: "two wrong diagnoses, two environment changes, and roughly eight hours of rebuilding". (digest-B P2.1; HE ADR-0041; **MEASURED**.)

And the opposite failure: on 2026-07-27 an assistant talked the owner out of five-minute checks on a background build. It hung for 10.8 hours; two of three outputs had finished in about a minute and sat unreported. (digest-A 7.8.)

How it works

1. **Plan mode, or a written plan, for anything with 3+ steps or an architectural decision.**
Include how you will verify it. "Use plan mode for verification steps, not just building."
(digest-A 2.1.)
2. **Write the plan to** `<project>/tasks/todo.md` as checkable items; check in before implementing; mark progress; add a review section at the end. (digest-A 2.2.)
3. **If something goes sideways, STOP and re-plan immediately.** Do not keep pushing.
4. **Count attempts.** Three tries of the same general approach is the limit. Record each attempt and why it failed, then pivot. Save the pivot and the reasoning. (digest-A 2.8.)
5. **A cause is something you reproduced.** Until then, call it a hypothesis. When a fix does not stop the symptom, first establish which build actually ran. Prefer a cheap experiment that splits the search space in half (bisect) over a satisfying story: "Cheap experiment that partitions the search space beats a satisfying explanation, every time." (digest-A 7.6.)
6. **When a human asks for periodic checks, treat it as a requirement,** not a proposal to argue with. "Silence and success are different states." Use a cheap signal (for example CPU time) to tell a hang from work. (digest-A 7.8.)

In the trenches

- **Quantify the mechanism against the symptom.** A recorded cause was worth 3 steps (38 to 41); removing a different factor gave 103. "A confirmed mechanism is a hypothesis until it is quantified against the size of the symptom." (digest-B P4.4 item 6; HexHound lessons.)
- **Read the measurement already in hand.** "The clock face in the photo he had ALREADY sent showed '19'. ... A call graph tells you what CAN happen; a counter tells you what DID." (digest-B P4.4 item 7.)
- **Three strikes applies to recovery, too.** A memory entry on a signing failure says "Do not retry past 3x" and hands the step to the human. (digest-A 2.8, 8.5.)
- **Plan-first pulls against "just fix it".** The same rulebook says "When given a bug report: just fix it. Don't ask for hand-holding." (digest-A 2.5.) This playbook's tie-breaker: a bug with a clear reproduction is a task; a change with design choices is a plan. The rulebook states none (digest-A2 9.20).
- **Over-asking is a recorded failure too.** In one overnight session the agent "still ran ~4 AskUserQuestion prompts"; the owner's correction: "You were supposed to make the best decisions and kept reverting to me." (digest-B P3.10.) Plan, then decide within your lane; see the decision guide.
- **"Demand elegance" has a trap.** A self-assessment warned the phrase "is the on-ramp to rewrite-loops and infinite refactor." (digest-A 1.8.) Ask the elegance question once per non-trivial change, not on every pass.
- **Long sessions degrade.** The same assessment counted "134 context-compaction events" and one session of "1,472 messages over 6.6 hours" (digest-A 4.11); see decision guide 5.

- **Cost:** planning time up front, and a check-in that can stall if the human is away.

Evidence

- ADR-0041's corrected diagnosis, with its cost stated (digest-B P2.1; **MEASURED**).
- The quantify-the-mechanism lesson, 3 steps versus 103 (digest-B P4.4 item 6).
- The 10.8-hour hung build that periodic checks would have caught (digest-A 7.8).

Do it

R6 (the per-task loop), R14 (when things go wrong).

P8 Corrections become rules (the lessons loop)

In one sentence

Every time you correct the agent, the pattern behind the mistake is written down as a rule in a lessons file, and the rules that matter most are promoted into mechanisms that fail a build rather than relying on anyone remembering.

In plain terms

An agent does not remember yesterday. If you correct it today, tomorrow's session will make the same mistake unless the correction is written somewhere it reads. The **lessons file** (`<project>/tasks/lessons.md`) is that place: a running list of "this went wrong, here is why, here is the rule". Think of it as a team's post-mortem log, kept by the agent itself.

Writing it down is step one. The method's own history shows that written rules are often not followed, so the important ones get turned into something that runs: a test, a lint check, a hook, a gate. See P12.

Why it exists

The global instructions state the loop: "After ANY correction from the user: update tasks/lessons.md with the pattern. Write rules for yourself that prevent the same mistake. Ruthlessly iterate on these lessons until mistake rate drops." (digest-A 2.2.)

The deeper reason is recorded in the firmware project's ADR-0026, written after three failures there (an egress bypass, a pin test that scanned nothing, and a missed deferral): "The cardinal rule 'no shortcuts, build it right the first time' was already written down in the operator's global instructions while all three happened. Intent was never the problem; enforcement was." (digest-B P2.4; HE ADR-0026; **MEASURED**; the three named per review RV-10.)

How it works

1. **Record at the moment of correction**, not at the end of the session.
2. **Use a fixed entry format**. One project uses `Date`, `Trigger`, `Root cause`, `Rule`, `How to apply`, with links to related memory entries; headings state the rule as a sentence, for example "A Guard That Reads A Number And Never Compares It Is Not A Guard". (digest-B P4.1.)
3. **Quote the human's correction verbatim** where it sets a rule. Examples: "Pull everything that is live, we don't want that mistake going out ... never assume." "You were supposed to make the best decisions and kept reverting to me." (digest-B P4.5.)

4. **Write the general pattern, not only the instance.** "The pattern, stated generally" is a heading in one lessons file. (digest-B P4.1.)
5. **Promote rules that guard something important into mechanisms (P12).** Promote rules that recur across projects into the global instructions.
6. **Review lessons at session start** for the current project.

In the trenches

- **Promotion works in both directions.** A deferral missed at milestone M1 in one project became a gate check in that project, a gate check in a second project, and step 6 of the global rulebook, "with the same wording". (digest-B P4.3; **MEASURED**.)
- **Recurring lessons across projects are the strongest candidates for the global rules.** The research found eleven patterns repeated across at least two lessons files, for example "a probe that finds nothing is usually a broken probe", "two states that must never be confused share a representation", and "documents that agree are not independent sources": "Corroboration among documents that share an ancestor is one claim counted three times". (digest-B P4.4.)
- **The loop is not wired.** No hook reads or writes `lessons.md` at session start; "review lessons at session start" is instruction text only. A May 2026 self-assessment put it bluntly: "Rules written down != rules followed" (its not-equal sign rendered in ASCII) and "the Self-Improvement Loop is documented but not wired." (digest-A 2.2; digest-A2 9.6.)
- **Lessons files grow without bound.** Measured sizes of the largest: 1341, 767, 665, 653 and 628 lines. (digest-B, lessons table; **MEASURED**.) A file that long cannot be read at the start of every session; see R15.
- **A rule stored in many places can still be missed.** The no-attribution rule for commits "got relearned" despite being held in "8+ project stores" (his figure), and was then filed globally. (digest-A 5.12, digest-A2 9.19.)
- **Cost:** a minute per correction to record; ongoing pruning; and the mechanism work for the rules that get promoted.

Evidence

Correction	Mechanism it became	Source (digest-B P4.2)
Retired claims survived in a render source	A checker that exits 1 over every string	JH <code>check_ledger.py</code> , MEASURED
A milestone passed with a deferral undone	Gate greps the questions file; exit criterion at M5	HE ADR-0026, MEASURED
Same incident, another repo	Phase gate fails on any open deferral, and on an absent file	UT <code>phase-gate.ps1</code> , MEASURED
An ASCII punctuation rule "looked enforced and was not"	A punctuation check in <code>make lint</code>	QM handoff, DOC-STATED
Prose counts drifted from artifacts	A counted-claims test	QM handoff, DOC-STATED
A compiled-binary import defect shipped	A test "proven to go red against the real defect before being kept"	HD changelog, DOC-STATED

Do it

R2 (write your project instructions file), R3 (set up todo and lessons files), R15 (keeping instructions lean, pruning).

P9 Memory: one fact per file, indexed, shared across tools

In one sentence

Keep durable knowledge as small notes, one fact per file with a short labelled header, listed in an index the agent loads at start, saved proactively as things happen, and shared by every tool and machine that works on the same projects.

In plain terms

Each new agent session starts with an empty head. **Memory** is a folder of small notes the agent writes so later sessions know what earlier ones learned: decisions and why they were made, your preferences, how a past bug was fixed, what did not work.

The method's format:

- One note per topic, as its own file. A short header block (**frontmatter**) at the top gives its name, a one-line description and its type (user preference, feedback, project fact, reference).
- An **index file** (`MEMORY.md`) with one line per note. The agent reads the index at the start; it opens a full note only when relevant.
- Notes are saved without being asked, as things come up: "Treat every conversation like a colleague taking notes." (digest-A 4.1.)

Claude Code has a built-in per-project memory folder that works this way; Codex CLI reads `AGENTS.md` and can be pointed at the same folder. Any agent that can read files can use the format.

Why it exists

Two incidents shaped the sharing rules.

On 2026-09-02, the second agent tool (Codex), asked what was in its memory, created a private note the first tool could not see. Two days later it reported a rebrand as uncommitted and a device as hard-reset, while the shared store had the work committed and pushed. "Two assistants, one project, contradictory records, and no way for either to notice." (digest-A 4.8.)

On 2026-09-14, a project's index file had grown past the size at which it is loaded. The tool truncates it "silently, dropping the entries at the END of the file. Those are the newest ones", and "the only notice came AFTER the loss". (digest-A 4.6.)

How it works

1. **Save proactively:** decisions and rationale, preferences, project context, corrections, confirmed approaches, external references, cross-project links, bugs and incidents. Do not save what the code or git history already shows. (digest-A 4.1.)
2. **Save the troubleshooting journey, including dead ends.** "What we tried that DIDN'T work, and WHY it failed. This is just as important as the fix." Save during the process, so a session that ends mid-debug can be resumed. (digest-A 4.2.)
3. **One fact per file; update rather than duplicate.** End feedback and project notes with "Why" and "How to apply" lines; link related notes by name. (digest-A 4.3.)
4. **Read the full note, not the index line.** "The one-liner is a paraphrase." If a note names a path, flag or function, verify it still exists before acting on it. (digest-A 4.7.)
5. **Two writers, one store.** When two tools share memory, tag the author in the header, never rewrite the other tool's notes (write your own and link), append to the index rather than regenerating it, and when two notes contradict, "Flag both, quote each, and name the measurement that settles it." (digest-A 4.8.)
6. **Hygiene is automated.** Hooks check for stale paths and dangling index entries at session start, warn when the index approaches its size limit, flag notes changed by another session mid-conversation, and commit memory changes to git at session end. (digest-A 4.5, 4.6.)

In the trenches

- **Prove the bridge in both directions.** On 2026-09-04 the shared store was tested with real content: the second tool, barred from reading the source code, answered two device facts correctly by citing a note the first had written, then wrote its own tagged note and appended exactly one index line. (digest-A 4.9.)
- **Memory can carry secrets to the wrong place.** On 2026-07-28 notes were staged to a travel laptop after redaction and a narrowed scan pattern; "three credential filenames that were never on the redact list, plus a live cloud API token prefix" reached it and had to be purged. The rule: "If a document's TOPIC is where secrets live, EXCLUDE it. Do not redact it." (digest-A 6.6.) Store pointers to secrets, never secrets.
- **Memory keyed by folder path fragments.** Moving or renaming a project silently starts an empty store; the research counted 14 folder names containing one product's name, and 3 created for throwaway scratch or worktree paths (**MEASURED**). (digest-A2 9.16.)
- **A shared store on removable media is a single point of failure** (costs row 15). The Mac and Linux setups copy instead of link for this reason; keep a local copy of anything safety-critical. (digest-A2 9.8.)
- **Cost:** notes accumulate and go stale; index size limits bite; a shared store needs sync machinery (R13). Cap anything a hook prints: "a hook is a pointer, not a payload". (digest-A 4.10.)

Evidence

- The round-trip test that proved both tools read and write the shared store (digest-A 4.9; measured with a one-line diff of the index).
- The hygiene hooks and the truncation guard, added in response to named gaps (digest-A 4.5, 4.6).
- The staging leak that produced the "exclude, don't redact" rule (digest-A 6.6).

Do it

R4 (persistent memory), R13 (multi-machine and multi-tool setup), R15 (pruning).

Part 3: Obligations and decisions

P10 A deferral is an obligation with enforcement

In one sentence

Whenever you decide to do something "later", name the milestone or event that discharges it, and make that milestone's completion check refuse to pass while the deferral is still open.

In plain terms

Every project defers things. The danger is not deferring; it is forgetting. A note saying "fix at milestone 3" is not a plan unless milestone 3's definition of done actually reads that note. The method treats "later" like a bill with a due date and makes a machine, not memory, collect it.

Re-deferring stays cheap. The point is that the decision is visible, not that you can never change your mind. Silence is what is forbidden.

Why it exists

In the firmware project, a questions-file entry said "At M1, replace it with checks on ... and make those hard failures." "M1 was implemented, gated and declared PASSED with that undone. Nothing was watching ... It was found only by a post-completion audit." (digest-B P2.4; HE ADR-0026; **MEASURED**.) The global rulebook records the same event on 2026-08-18: the gate "checked the milestone document and nothing checked the deferrals". (digest-A 1.4.)

How it works

1. **Every deferral names its discharging milestone or event.** In projects with no next phase, name an event: "Discharged by the next change to `tools/sandbox.py` ." (digest-B P2.6.)
2. **Keep deferrals in a register** with a status: `OPEN` , `DISCHARGED` (with date and evidence) or `WITHDRAWN` (with reason). Each entry says why it is deferred, what seems it leaves, and the cost of leaving it open. (digest-B P2.5.)
3. **The gate reads the register and fails** while any deferral owed at that milestone is open. An absent register is a failure too: "removing it is not a way to pass this gate".
4. **Any bypass switch still fails.** In one project the override flag "Still exits non-zero if deferrals are open, so this cannot be used to sneak a green build through CI" and prints a banner. (digest-B P2.5.)
5. **Prove the gate both ways:** open deferrals make it fail; all discharged makes it pass.

6. **A gate that cannot be run blocks the release.** "If a gate cannot be run, that blocks the release; it does not downgrade to a caveat." (digest-B F5.6; EraseDrive lessons.)

In the trenches

- **Every TODO needs a register line.** "Every TODO in the codebase requires a line here with its file path. An unregistered TODO is a defect." (digest-B P2.3.)
- **The mechanism can be built and not used.** In one project the deferral list was an empty set, so three gates that iterated it "run zero assertions and report green". "Prose in three files is still memory." (digest-B P4.2.)
- **Red by design.** One project's gate is "EXPECTED to fail on a healthy repo with open deferrals"; it currently has 7 OPEN and 4 DISCHARGED entries (**MEASURED**). Its gate counts every OPEN entry regardless of milestone (verified in the gate script), so it cannot tell "owed now" from "owed at M3". It also passes if it parses zero entries, the empty-parse weakness P3 warns about. A permanently red gate risks being ignored. (digest-B P2.5; review RV-8.)
- **Event-triggered deferrals have no date,** and nothing the research found fires on "the next change to the file". They depend on someone noticing. (digest-B P2.6, **INFERRED**.)
- **"Complete" is not "nothing open".** "'Complete' means every phase shipped, not that nothing is open." Say so in the status. (digest-B P2.6.)
- **Cost:** "every gate now reads QUESTIONS.md, which is a few lines of script and one more reason that file must stay accurate." (HE ADR-0026; **DOC-STATED**.)

Evidence

- HE milestone M5 criterion 7: "Every QUESTIONS.md deferral scheduled for resolution at M5 is discharged - PASS", listing nine resolved entries, with the note that "every gate from M0 to M4 called this by habit and nothing required it". (digest-B P2.4; **MEASURED**.)
- UT phase gate "Proven BOTH ways: 5 open -> exit 1; all discharged -> 10/10 pass, exit 0". (digest-B P2.5; **DOC-STATED**.)
- One incident produced mechanisms in two repositories and a global rule (digest-B P4.3; **MEASURED**).

Do it

R10 (deferral register with a failing gate).

P11 Decisions are artifacts (ADRs, questions files)

In one sentence

Record each significant decision as a numbered, dated document that states the decision as a rule, the context and the cost; log open questions in an append-only file that never blocks work; and correct wrong records with new records rather than by silent edits.

In plain terms

Six weeks from now, neither you nor a new agent session will remember why something was built the way it was. An **ADR** (architecture decision record) is a short document: what was decided, what the options were, why this one, what it costs. A **questions file** is a log of things you were unsure about, what you assumed for now, and when the question must be settled.

Together they let an agent keep working overnight past a question (by writing down its assumption) instead of stopping to wait for you, while leaving a trail you can review in the morning.

Why it exists

Decisions made in chat vanish with the session. The method's projects show both the value and the cost at scale: the firmware project has 77 ADR headings (ADR-0001 to ADR-0076, one number used twice), and the orchestration project has 67 numbered decision files (**MEASURED**). (digest-B P2.1.)

The clearest case for correction records: an earlier ADR recorded a tool as killed by the out-of-memory killer. "That was wrong, and it was asserted three times on no evidence". A new ADR corrected the context and left the original decision standing, and recorded that two remediations "were aimed at a problem that did not exist". (digest-B P2.1; HE ADR-0041; **MEASURED**.)

How it works

1. **Title the ADR as the rule, not the topic.** "A cap refuses; it never truncates." "A counter that is zero by construction is deleted, not wired up." (digest-B P2.1.)
2. **Questions file: append-only, never blocking.** "Log the question, make a reasonable decision, record it as an ADR if it is architectural, and keep going." Fixed fields: Raised , Context , Assumed for now , Blocks . (digest-B P2.3.) Once a gate reads the file (**P10**), an entry due at a milestone does block that milestone; that is the point. "Never blocking" applies to day-to-day work.

3. **Correct with a new record.** Do not rewrite history; add a correction ADR that cites the one it corrects.
4. **Write an honest "ships open" section.** One project's progress file says: "If you are arriving with no context, read the closing assessment ... It says what the system does, what it does not do, what ships incomplete ... and what not to trust yet." Another has "What I would not trust yet: ... Not one has ever received a byte from a real endpoint." (digest-B P2.7.)
5. **When a summary contradicts itself, rewrite it.** A progress file "patched incrementally through a long session until it contradicted itself in four places" was rewritten: "A summary that disagrees with itself is worse than no summary." (digest-B P2.8.)
6. **Record a "no" so it is not asked again.** When the owner declined a settings template, it was saved as: "when that hook fires again ... do not ask; ignore it silently." (digest-A 8.6.)

In the trenches

- **Name which document wins.** "Where the two disagree, ARCHITECTURE.md is right and this file is a defect." (digest-B P2.7.)
- **Keep failure modes distinct in records as in code.** The OOM correction produced the rule "Two failure modes that must be told apart must not share a representation". (digest-B P2.1.)
- **A "would not trust yet" line is a test plan.** The orchestration project's first-live-call campaign (P2) was written against exactly that line. (digest-B P2.7.)
- **Records have defects too.** The duplicate ADR-0039 heading is "itself a small defect in the record". (digest-B P2.1.)
- **Cost:** writing time, and every file you create is one more file that must stay accurate. The questions file becomes load-bearing once gates read it (P10).

Evidence

- 65 questions-file entries in the firmware project, with resolution markers spread across milestones M1 to M5, and the last two "OPEN, due at M6" (digest-B P2.3; **MEASURED**; the milestone counts are string occurrences, not distinct entries).
- The "would not trust yet" line converted into four fixed defects (digest-B P1.8, P2.7).
- The correction ADR that recorded roughly eight hours lost to a wrong diagnosis (digest-B P2.1).

Do it

R6 (per-task loop, record step), R11 (session handoff).

Part 4: Mechanisms, guardrails and boundaries

P12 Change the mechanism, not the wording (hooks over instructions)

In one sentence

When a rule that was genuinely present in the agent's instructions is still not followed, do not answer with a louder instruction; move the behaviour into something the harness runs (a hook, a lint check, a gate) so it no longer depends on the model remembering.

In plain terms

Instructions are advice the model reads. A **hook** is a small script the tool runs automatically at a fixed moment (a session starting, a tool about to run, a turn ending), whether or not the model thinks of it. The program around the model that runs tools and fires hooks is the **harness**. A rule the harness enforces holds even when the model is distracted, its context is full, or the instruction was never loaded.

Analogy: a sign saying "close the gate" versus a gate on a spring.

Claude Code calls these hooks and supports events such as session start, before a tool call and on stop. Other tools offer git hooks, CI checks or wrapper scripts to the same end.

Why it exists

On 2026-09-04 the second agent tool had written zero tagged memories in eight days, which looked like disobedience. Measured, the protocol text appeared in "0 of 59" of its session transcripts while ordinary instruction text appeared in "11 of 59": the protocol had been installed after its last session started. "It is not following the rule" and "it never received the rule" produce identical evidence." (digest-A 7.7.)

The general lesson recorded: "When the instruction genuinely was present and still was not followed, do not answer with a stronger instruction. Change the mechanism." (digest-A 5.10.)

How it works

1. **First prove the instruction was present.** Search the session transcripts for a distinctive string from the rule, plus a positive-control string you know was loaded; compare the rule's install time to the session's start time. (digest-A 7.7.)
2. **If it was absent, fix delivery.** If it was present and ignored, change the mechanism.
3. **Pick the lightest mechanism that cannot be skipped:** a lint check for writing rules, a test for behaviour, a gate for milestones, a hook for session-level behaviour, a deny rule for

forbidden commands.

4. **Make hooks structural and safe.** The critique-persistence hook in this method detects its trigger from the harness's own text, is idempotent by filename, never blocks, and writes nothing to standard output (so nothing it prints can reach the model's context or the transcript). "Persistence therefore does NOT depend on the model remembering to write anything." (digest-A 5.10.)
5. **Guard every hook against missing dependencies,** and prove it both ways (see below).

In the trenches

- **A hook can enforce nothing while looking installed.** An early draft of a fail-closed identity guard depended on a tool that was not installed on one machine, "so it exited 0 on every input and enforced nothing while appearing installed". (digest-A 6.7.)
- **A hook can block everything.** In the version the method used, a prompt-submit hook that failed blocked the prompt: "with the SSD unplugged every prompt the user typed was rejected before Claude saw it". Current Claude Code documentation says only exit code 2 blocks by itself and other non-zero codes are non-blocking errors (review RV-12, checked against the vendor's hook docs). Check your tool's docs for your version, and test the dependency-absent case either way. The fix wrapped 13 of 16 shared hooks (his figures) in a presence guard and was "Verified across the cross-product (present x exit 0/1/2, absent)". (digest-A 5.6.)
- **Silence means both "fine" and "broken".** Tooling built never to fail a session ("all errors swallowed") makes a broken guard look like a clean one; in one log, two servers are skipped on all 346 skip lines and never registered (**MEASURED**). (digest-A2 9.14.)
- **A control in one code path is not a control if another path carries the same payload.** On a new Linux laptop, a sync script refused to register production tool servers until a security gate passed, but "the merge walked around the guard" because the shared settings carried the same hook plus an auto-approving hook. The fix was a platform filter that imports only what can run as written and names every drop. (digest-A 5.11.)
- **Hook output must be small, for two reasons.** Plain output from session-start and prompt-submit hooks is added to the model's context, so every byte costs tokens in every session; for most other events output goes to logs or the transcript (review RV-12; check your tool's docs). And on the method's machines, a hook that wrote more than the pipe buffer "BLOCKS FOREVER and surfaces as a bare 'exited with code 1'." (digest-A 4.10.) "A hook is a pointer, not a payload."
- **Cost:** every hook is code, needs every interpreter it calls on every machine, and can fail. The shared hook set calls four runtimes and runs six hooks at every session start (**MEASURED**). (digest-A2 9.15.)

Evidence

- A working punctuation check, run for the first time, "found nine pre-existing violations dating to phase 0, in a repository whose CLAUDE.md had forbidden them throughout. ... The rule looked enforced and was not." (digest-B F5.6; QM handoff; **DOC-STATED**.)
- The transcript measurement that separated "not received" from "not followed" (digest-A 7.7).
- The mechanisms table in P8.

Do it

R2 (write your project instructions file), R5 (permissions, deny rules and hooks).

P13 Guardrails and human-in-the-loop

In one sentence

Decide in advance which actions an agent may take alone, which need a human, and which are forbidden outright; enforce the forbidden ones mechanically; and when a human is needed, reach them in the way that actually works.

In plain terms

An agent with tool access can delete files, push code, send messages and spend money. **Guardrails** are the rules and mechanisms that stop it doing something unsafe. **Human-in-the-loop** means a person must answer or approve before certain steps.

Three tiers, as the method uses them:

Tier	Examples from the method	How it is held
Forbidden	Printing a secret; reading private key files; active security scans against unlisted targets; running malware on a host; process-name-wide kills of the user's apps	Instructions plus deny rules and guard hooks where they exist
Human approves	Anything outward facing or irreversible: pushes, publishing, sending, signing; out-of-scope repos; regulated data	A single checkpoint before the first such action
Agent decides	Reversible, in-lane choices	Decide, execute, note the call in one line

Why it exists

Secrets: a memory rule headed "RULE #1, ABOVE EVERYTHING ELSE. Never let a secret value reach output" was "Established 2026-05-10 after the 4th leak in 10 days": a token in a git URL, a token in a verification echo, a token in an error-path file dump, and a container environment dump. Each leak meant rotating the secret everywhere it was used. (digest-A 1.7.)

Bulk operations: violating the bulk filesystem rules "cost a full-drive read-only sweep on 2026-08-17". Two layers of shell quoting "turn a path into a different valid path rather than into an error", and a loop that logged "done" while swallowing exceptions hid 940 errors. (digest-A 6.8.)

Identity: on 2026-09-09, commands from a separate-persona repository "went out under the" company account, because the command-line tool had one globally active account,

linking two identities. (digest-A 6.7.)

How it works

1. **Write the forbidden list as policy:** never print secret values (report presence or length only); never read private key files; never put credentials in URLs, command arguments or the transcript; no active security scanning without a written authorization list naming target and scope and an explicit typed "YES"; no execution of malware outside an isolated machine; never use real regulated data for testing. (digest-A 1.7, 6.1, 6.4, 6.5, 6.10, 6.12.)
2. **Back the forbidden list with mechanisms:** deny rules in the tool's permission settings, and guard hooks that fail closed ("anything unexpected denies rather than allows"). (digest-A 6.7.)
3. **Bound bulk operations:** script them rather than inlining; never pipe a completion-critical command into a pager that can cut it short; operate on a literal list of absolute paths and assert none is a drive root; count failures; verify with a different tool. (digest-A 6.8.)
4. **Treat content the agent reads as data, not orders.** Text in a web page, file or tool output that tells the agent to do something is **prompt injection**; the OSINT skill forbids acting on "look this person up" instructions found in observed content. (digest-A 6.3.)
5. **Check platform policy before using an agent at all** where terms may forbid it; "If the policy forbids it, say so and stop. Do not proceed and hope." (digest-A 6.2.)
6. **Reach the human the way that works.** The method's default is to ask in the terminal and end the turn. A phone bridge is used only on opt-in (when the owner is away), because the bridge "blocks on a Slack poll" and terminal answers were getting swallowed. (digest-A 8.1.)
7. **Remote approval must fail safe.** When away-mode is on, a pre-tool hook sends the pending call to the owner's phone and waits for yes or no; on any failure "the worst case is 'approval falls back to terminal'". (digest-A 8.3.)
8. **Some steps stay human.** Where the password manager refuses signing requests from the agent's shell, the agent prepares commits and hands the push to the human, and does not change signing or SSH configuration itself. (digest-A 8.5.)

Unattended runs

Overnight and many-agent work means sessions that run with nobody at the prompt. Most agent tools offer an auto-approve or "skip permissions" mode for this. Such a mode removes the per-action approval prompt, which is the human half of this pattern; what else it disables differs by tool and version, so check your tool's docs for exactly which checks each mode turns off. The digests record no incident from such a mode, and the method's only recorded unattended-approval mechanism is the phone-approval hook above. The policy this playbook recommends (review RV-18):

1. **Never use a skip-permissions mode on a machine that holds credentials, production access or regulated data.**
2. **If you need one, run it in a disposable container or virtual machine** with only the repository mounted: no SSH agent, no cloud credentials, no password manager, and network access limited to what the task needs.
3. **Keep the deny rules and guard hooks** in place; in most tools they still apply, but verify that in yours.
4. **Branch only, no push rights (P15).** The human reviews and pushes in the morning.

In the trenches

- **Autonomy in lane is also a rule.** "Make every reversible in-lane decision yourself, execute, and note the call in one line ... Batch any genuine user-only decisions into a SINGLE checkpoint at the first irreversible/outward-facing action." (digest-B P3.10.) Over-asking and overstepping are both recorded failures.
- **Hard-wire the outward boundary where the stakes are high.** The job-search pipeline runs with " `AUTO_SUBMIT=False` ", and there is NO headless submit path". (digest-B P4.6.)
- **Precision over recall for interruptions.** The away-mode detector was revised after "repeated false-ON incidents" because "A false ON ... pings Slack while he's AT the desk - the exact noise this system exists to prevent. A missed ON is mild." (digest-A 8.2.)
- **Guardrails fail silently when their environment changes.** An expired root certificate in the operating system's store "silently" broke every remote-approval call, making away-mode read as off. (digest-A 8.3.)
- **Approval fatigue creates craft:** each "Yes, don't ask again" becomes a permanent allow rule (costs row 10). The fix is a periodic manual audit. (digest-A2 9.2.)
- **A new machine inherits a classification.** A laptop classified for production and regulated data refuses to register production credentials until a recorded security gate passes. (digest-A 6.13.)
- **Some guardrails are stated but not verified in code.** The pentest tools are said to have "built-in safety checks"; the research did not read that server's code, so enforcement is not verified. (digest-A 6.1.)
- **Command-pattern deny rules are a speed bump, not a wall.** A rule that blocks one spelling of a command does not block an equivalent spelling. For anything that must not happen, use a hook that parses the command, a sandbox without the credentials, or keep the secret off the machine. (Review RV-5.)
- **Transcripts hold everything the agent read.** Session transcripts on disk contain every file and command output the agent saw, and syncing the agent's home folder copies them. Treat them as sensitive as the most sensitive thing the agent touched. (Review RV-19.)
- **Cost:** friction on every guarded action, and upkeep of the lists.

Evidence

- The persona guard exists because of a real cross-identity leak, and fails closed (digest-A 6.7).
- The "exclude, don't redact" rule exists because a redacted staging still leaked credential filenames and a token prefix (digest-A 6.6).
- Containment that held during an overstep (see **P15**).

Do it

R0 (before you start), **R5** (permissions, deny rules and hooks), **R14** (incident procedures, including secret leaks).

P14 Verify the environment before blaming the code

In one sentence

Before concluding that code or data is broken, prove that the environment and the plumbing are what you think they are: the port is yours, the path resolved, the probe can find something, the pipe carries bytes intact, the right build ran.

In plain terms

Several recorded "bugs" in this method's history were not bugs in the code. Another program already owned the port. A relative path stopped pointing where it used to. A search used the wrong naming convention. Antivirus quietly moved a file. The agent then "fixes" code that was never broken, sometimes breaking it for real.

A **probe** is any check for whether something exists. A **positive control** is running the same probe on something you know exists, to prove the probe works.

Why it exists

On 2026-07-31 a test server "started" on a port another local service already owned; its "Not Found" response "looked exactly like a regression in the MCP client under test". On Windows, binding to all interfaces can succeed while an existing local-only listener keeps the traffic. Moving to a verified-free port made the test pass at once. (digest-A 7.1.)

A search for SSH keys by one filename found nothing in an archive; the keys were all there under other names (29, 42 and 32 matches for other patterns, his figures). "A probe that finds nothing has two explanations, and the boring one is usually right: the probe is wrong." (digest-A 7.2.)

How it works

1. **Check shared resources before using them:** ports, lock files, container names, device nodes. Confirm the thing answering is yours.
2. **Run a positive control before reporting an absence.** Use absolute paths. Never suppress errors on a probe whose absence you will report. (digest-A 7.2.)
3. **Prove the plumbing with a known-good input** and a known expected count before running real data through it. (digest-A 7.3.)
4. **Know each tool's single failure signal.** The method notes that one encryption tool has exactly one wrong-passphrase message, so its absence means the passphrase was accepted even when a later stage fails. (digest-A 7.3.)

5. **Establish which build actually ran** before theorising about why a fix did not work. (digest-A 7.6.)
6. **Suspect the environment's side effects:** antivirus quarantine, line-ending conversion, stale packages shadowing real ones.

In the trenches

- **Pipes can corrupt.** One Windows shell decodes output between native programs as text, so a decrypt-then-extract pipeline failed with "Unrecognized archive format", which "reads exactly like a corrupt archive or a wrong passphrase and is neither". (digest-A 7.3.)
- **Five of seventeen false failures.** A guard run "reported 5 of 17 FAILED. All five were false": a relative path stopped resolving after a directory change, and it "looked exactly like the rule blob failing to bundle". Four independent checks settled it "before any conclusion was drawn". (digest-B P1.14; HD overnight handoff; **MEASURED**.)
- **Shadowing packages mimic product defects.** A wrong package with the same name produced errors that "look exactly like product defects"; "I nearly reported 'three detection rules are broken' as a product finding. It was false." With the real binding, all 54 rule files compiled. (digest-B P1.14; **MEASURED**.)
- **An empty search can hide a real file.** A persisted directory change made a relative path invalid, a quiet flag hid the error, and a 611-line existing module was reported as absent. (digest-A 7.2.)
- **Line-ending conversion can corrupt a repository.** A default setting that converts line endings silently altered 552 files in one repository, "including the model signing pubkey". Never "fix" a mass line-ending diff by committing it. (digest-A 5.13.)
- **The environment can be exhausted by the agents themselves.** See the orphaned process story in P15 and the costs section.
- **Cost:** a few seconds per check (estimate). The recorded skips cost an evening (P7) and nearly produced a false product finding (above).

Evidence

- The five false failures caught before any conclusion (digest-B P1.14; **MEASURED**).
- The near-miss false product finding (digest-B P1.14; **MEASURED**).
- Recurrence across projects: "a probe that finds nothing is usually a broken probe" appears as a lesson in several lessons files (digest-B P4.4 item 1), and "prove the harness before blaming the product" after "Three times, a bad test was mistaken for a bad product" (item 8).

Do it

R0 (before you start), R14 (when things go wrong), R6 (verify step).

P15 Scope boundaries and handoffs

In one sentence

An agent works only where it was opened and on what it was asked, commits only the files it wrote, and keeps a live handoff document updated at every boundary so a crash or a new session can pick up the work without guessing.

In plain terms

Two problems, one pattern.

Scope: an agent asked to look at something may decide to fix it, and may wander into other projects to do so. The rule: if it is not the repository you were opened in, it is read-only unless you were explicitly told otherwise.

Handoffs: sessions end, crash, or run out of context. A **handoff** is the note that lets the next worker continue: where things stand, what was measured versus assumed, what is left, and which decisions belong to the human. Written at the end, a crash takes it with it; written as you go, it survives.

Why it exists

On 2026-09-19 a session opened at the top of the projects folder was told to "survey the portfolio and come back with a list of what each session should work on. It went and did the work instead, in repos it was not opened in." Four repositories were changed. In one, "PRODUCT CODE WAS MODIFIED, not just tests." Another "had no version control at all before today, so there is no `main` to fall back to and the edits were made IN PLACE", overwriting two existing files. (digest-B P3.6; four check-my-work handoff files; **MEASURED**.)

On 2026-08-19 the host process crashed ("segfaulted about 24 hours into a session and killed a six-agent audit mid-run; four sub-reports survived only because subagent transcripts are written to disk incrementally, and the parent's consolidation was lost outright"). (digest-B P3.3; QM handoff lines 3-10; **MEASURED**.)

How it works

1. **Read-only outside your repository** unless told otherwise. The standing rule: "if it is not the repo you are opened in, it is READ ONLY unless explicitly told otherwise." (digest-B P3.6.)
2. **Scope-lock helpers that write** to named files or directories. (digest-B P3.7.)

3. **With parallel agents, commit named paths only.** "While parallel agents are running, a commit names its paths ... never `git add -A` and never `git commit -a`." (digest-B P3.4; HE ADR-0028.)
4. **Keep a live handoff**, updated at every phase or wave boundary, not at the end.
5. **Use a consistent handoff shape:** current state first; measured versus assumed, labelled; wrong diagnoses recorded with why they were convincing; decisions reserved for the human; exact commands to re-verify. One handoff opens: "Read sections 1 and 2; the rest is detail." (digest-B P3.5.)
6. **If an overstep happens, hand it to an auditor, not to yourself.** The overstep handoffs say "Your job is to audit it, not to assume it is sound. Reverting all of it is a legitimate outcome", with verify commands, a short revert, and "Places I am most likely to be wrong". (digest-B P3.6.)

In the trenches

- **A scope lock on a helper does not constrain the parent.** In the overstep, one commit was made by a helper "under a scope lock limited to the two corpus test files", while the parent session was itself out of scope. (digest-B P3.7.)
- **Sweeping commits tell lies.** During one milestone, a checkpoint `git add -A` swept three engineers' in-progress work into a commit "whose message described none of it". "Nothing was lost"; the harm was a commit that "asserts something untrue". (digest-B P3.4.)
- **Agent worktrees inside the repository** get staged by a sweeping add as embedded repositories. (digest-B P3.9.)
- **Crash recovery is possible if you know where to look.** A finished helper's last message is a long text block; a killed one ends on a tool call or tool result. Four reports were recovered that way. (digest-B P3.3.)
- **Live handoffs drift.** A handoff reported a commit count that "was stale by twenty-six for several hours, in a handoff written in the same repository as" an ADR titled "a counted claim is an assertion". (digest-B F5.7; **MEASURED**.) Reconcile the whole document at each update, not only the top.
- **Containment is what limited the damage:** branches only, never pushed, `main` identical to the remote, no build or deploy (digest-B P3.6; **DOC-STATED**). Branch-only work and a human-held push (P13) make an overstep recoverable.
- **Cost:** more, smaller commits; handoff upkeep at every boundary; the rule was written after the incident, not before.

Evidence

- Four crashed-audit sub-reports recovered from incrementally written transcripts (digest-B P3.3; **MEASURED**).

- The overstep branches remained reviewable and revertible because nothing reached `main` ; two are still listed as open review items in the cross-portfolio audit (digest-B P3.6; **DOC-STATED**).
- 23 handoff files exist across the product folder (digest-B P3.5; **MEASURED** count).

Do it

R11 (session handoff), R8 (independent audit, for reviewing an overstep).

P16 A claims ledger: every public claim traces to a measurement

In one sentence

Keep one file that is the only permitted source of factual claims for anything that leaves your hands (resumes, marketing, docs, reports), and admit a claim only if you can name its source and that source is a measurement or external ground truth, not another document repeating it.

In plain terms

Agents write fluent, confident text, and they will happily reuse a number from an old note. Once that text goes to an employer, a customer or a public page, a wrong figure is expensive to take back. A **claims ledger** is a single file listing every fact you are willing to state publicly, each with its status (`APPROVED` , `VERIFY` , `DO NOT USE`) and its source. Agents drafting outward-facing material "may only assemble from this file. If a requirement is not covered here, report the gap and stop." (digest-B P1.1.)

Why it exists

A competition result was carried as a perfect record when the tournament's own site showed a loss. It was in the master resume, the generator data, and application documents already sent to employers. It had been marked `APPROVED` "for weeks", and a review had concluded the claims "hold up under scrutiny". "It was caught only because the owner insisted on being specific". (digest-B F5.1; JH FACTS lines 486-503; `DOC-STATED` .) The wrong figure was real once: a mid-tournament snapshot in a local notes file, captured hours before the event closed. It was stale, not invented, which is why re-reading the local file never caught it. (Review RV-9; JH FACTS lines 494-500, 566-572.)

A customer-facing product guide claimed "15,852+ samples ... 98.8% detection rate", committed 2026-04-02, while the first full-corpus result files are dated 2026-04-18: "the claim predates by sixteen days any scan that could have produced it." It was removed 2026-09-13. (digest-B P1.1; `DOC-STATED` .)

How it works

1. **The two-part source test.** "1. You can name its source. 2. That source is a measurement or an external ground truth, not another document that repeats the claim." (digest-B P1.1.)
2. **Provenance is not verification.** "Tracing a figure to a local note tells you where it was copied from, not whether it was ever true."

3. **Unsourced numbers are removed, not adjusted.** "Substituting a plausible figure manufactures the measurement the sentence never had."
4. **Distinguish stale from unanchored.** "Stale is caught by re-checking the source. Unanchored is not, because re-checking finds a real and impressive measurement sitting right next to the claim and the mind supplies the connection. Ask whether the number is that measurement, not whether a measurement exists." (digest-B P1.2.)
5. **A grep is not a check.** Searching results for the figure "98.8" does return hits: progress lines taken 2% and 5% of the way through a scan of a different corpus, in a file added nearly four months after the claim. "A reviewer who grepped for the figure, got hits, and marked the claim sourced would have been wrong in the most defensible-looking way available." Read the context around every hit. (digest-B P1.2.)
6. **Enforce the ledger on the generator** with a checker that fails the build (P3), and mutation-test the checker (P2).
7. **Check whether a later artifact supersedes your source** before citing it.

In the trenches

- **"Every claim removed from this ledger passed whatever review it originally got."** The two-part test was written after the competition-record failure, not before. (digest-B P1.1.)
- **The ledger committed the failure it exists to catch.** A live-call status was cited from a progress file that had been "superseded later the same day" by a decision record, and corrected "within hours": "One source was cited without checking whether a later artifact overruled it, which is the exact failure this ledger exists to catch, committed inside this ledger." (digest-B F5.2.)
- **Sent is sent.** "Applications already sent keep their original wording." (digest-B F5.1.) A ledger reduces future errors; it does not recall past ones.
- **Scope claims honestly.** The ledger records one provider proven, one authenticated but uncredited, one local model proven, four untouched, and: "Never imply production use or that all eight adapters work." (digest-B P2.7.)
- **The project's own instruction file can overclaim.** One product's instructions describe its license signing in a way the audit found the code does not match, and the ledger dropped the claim for having no source. (digest-B F5.1.)
- **Cost:** "Does this number come from a measurement? Nothing automated answers this. A person has to." (digest-B P1.2; **DOC-STATED**.) The ledger is a standing manual cost.

Evidence

- Four recorded removals in the ledger's own section 0, including a "9 of 9 rated critical" figure that "belonged to a different sample set entirely", blocked before a public upload (digest-B P1.1; **DOC-STATED**).

- "100.00% on Contagio" stopped "one push short of the public site" and corrected to 99.98% (digest-B evidence item 4; **DOC-STATED**).
- Reading context around a grep hit caught a superseded figure the ledger had just added (digest-B P1.2; **DOC-STATED**).

Do it

R12 (build a claims ledger).

P17 Know the costs

In one sentence

Every pattern here has a price in time, tokens, upkeep or friction; pay it where the stakes justify it, skip it where they do not, and record honestly where the method itself failed.

In plain terms

This method is heavy. It was built for regulated, safety-critical and trust-critical products, by one person running many agents across several machines. Some of it is worth using on a weekend script; much of it is not. Knowing what each pattern costs is what lets you choose.

Why it exists

The method's own records are candid about its failures: rules written after the incident, a checker that encoded a wrong fact, a wrong figure that reached employers, a session that overstepped into four repositories. A playbook that reported only the wins would repeat the ledger's first mistake: claims that passed whatever review they got.

How it works

Use this table to scale the method to the stakes. The "use" and "skip" columns are this playbook's judgement, drawn from the costs the digests record. The token column is qualitative: the digests record no token or money figures.

Pattern	Main cost	Token spend	Use it fully when	Lighten or skip when
P1 Verify	Seconds per claim	Low	Always	Never skip; lighten only the labelling in casual chat
P2 Test the tests	Time per fix; more tests	Medium (a suite run per un-fix)	Controls on egress, permissions, spend, safety	Cosmetic or throwaway code
P3 Generated evidence	Scripting per gate	Low	Milestones, public numbers	One-off tasks with no record
P4 Independent audit	"Roughly doubles the review effort"	High (a second agent reads everything)	Security controls, merges of parallel work	Small, reversible changes
P5 Fan out, re-verify	Re-reading time	High (grows with every helper)	Anything written into a load-bearing file	Exploratory questions you will not act on
P6 Name ambiguity	One line	Low	Always	Never
P7 Plan first	Up-front time	Medium	3+ steps or design choices	Clear bug with a reproduction
P8 Lessons	Recording and pruning	Grows with file size	Recurring work on a project	One-off sessions
P9 Memory	Upkeep, sync machinery	Index loaded every session	Long-lived projects, several tools	Short projects
P10 Deferral gates	Script plus register upkeep; a red gate	Low	Milestone builds	Projects without milestones
P11 ADRs	Writing time; more files to keep true	Low	Architectural choices	Trivial choices
P12 Mechanisms	Code that can itself break	Hook output that enters context	Rules that failed as instructions	Rules that have never failed
P13 Guardrails	Friction on guarded actions	Low	Secrets, outward actions, regulated data	Never skip the forbidden tier

Pattern	Main cost	Token spend	Use it fully when	Lighten or skip when
P14 Environment checks	Seconds per check	Low	Before any "it's broken" conclusion	Never
P15 Scope and handoffs	Smaller commits; handoff upkeep	Low	Parallel agents, long or overnight runs	Single short session
P16 Claims ledger	Manual review of every number	Medium	Anything public or sent	Internal notes

The detail, with sources, is in **What this costs, and when not to use it** below.

Do it

R15 (maintenance: keeping instructions lean, pruning memory).

Decision guide

Each rule below is grounded in the digests. Where the digests do not support a firm rule (for example, an ideal number of agents), this guide says so rather than inventing one.

1. A subagent, or do it yourself?

Situation	Choose	Why (source)
Answering needs reading across many files, repos or sources	Subagent(s)	Keeps the main context window clean (digest-A 3.1)
Several independent questions	One subagent per question	"One task per subagent for focused execution" (digest-A 3.1)
You need an audit of work you (or a sibling agent) built	A fresh subagent that wrote none of it; for high stakes, a different model or a human	Self-audit is how defects passed (P4; digest-B P1.10)
You already know the file and the fact you need	Do it yourself	A single lookup costs less than briefing a helper and re-verifying its report (inference from P5's re-verification cost)
The output is load-bearing: a ledger entry, a public claim, a gate verdict	You, first-hand, even if a helper found it	"A sub-agent report is a document repeating a claim" (digest-B P1.9)
The step needs a human's credential or signature	Neither: hand it to the human	The password manager refuses the agent's shell (digest-A 8.5)

2. How many agents?

The digests do not establish an optimal number, and this playbook does not invent one. What they record: research sweeps with **four read-only helpers** (digest-B P1.9); a build with **six teams in three merge waves** (digest-B P3.1); a **six-agent audit** that a host crash killed mid-run (digest-B P3.3).

The costs that scale with the count:

- Re-verification work and token spend grow with every report (P5).
- Merges surface interactions no single team saw: "The merge surfaced one interaction neither team could see alone." (digest-B P3.1.)
- Infrastructure: after a crash, "2022 orphaned `git.exe` processes holding 80.57 GB of commit", recurring at 189 more in about 1.3 hours while agents ran (digest-B P3.9; **MEASURED** in the handoff).

Practical rule: use as many as you can pay for, re-verify and merge. If you cannot re-read their output first-hand or audit their merges independently, you have too many. Set the budget before the fan-out, not after.

3. Fan out in parallel, or go sequential?

Fan out when	Go sequential when
Tasks are independent and read-only (research, surveys, audits of separate components)	Each step depends on the last, or changes shared files
Each helper can be scope-locked to its own files or branch	You need to show a human how to test before moving on: "one step at a time, then stop and show the owner how to test" (digest-B P4.6)
You have a merge plan: "merged one at a time with gates between" (digest-B P3.1)	Parallel writers would share a working tree (commit sweeps, digest-B P3.4)

Parallel builds that write code follow the shape the orchestration project used (digest-B P3.1, P3.4, P3.9): one branch per agent or team; agent worktrees created outside the repository, so a sweeping add cannot stage them; commits that name their paths; merge one branch at a time with the full gate re-run after each merge; resolve conflicts "by keeping both intentions rather than picking a winner"; and an independent audit after the last merge, then of the fixes (P4). Procedure: R8.

4. When must a human approve?

A human approves before:

- **Anything outward facing or irreversible:** push, publish, send, submit, sign, deploy. Batch decisions into "a SINGLE checkpoint at the first irreversible/outward-facing action" (digest-B P3.10).
- **Any change outside the repository the agent was opened in** (digest-B P3.6).
- **Anything touching secrets, regulated data, or credentials** (P13).
- **Any active security testing**, which additionally requires a written authorization list and an explicit typed "YES" (digest-A 6.1).
- **A release whose gate cannot be run:** that blocks the release, it does not downgrade to a caveat (digest-B F5.6).
- **Decisions the plan marks as the human's:** "Three questions in that plan are yours, not mine" (digest-B P3.5).
- **Switching a session to a mode that skips permission checks** (P13, "Unattended runs").

The agent does **not** wait for approval on reversible, in-lane choices (those inside the task and scope it was given): "make every reversible in-lane decision yourself, execute, and note the call in one line" (digest-B P3.10). Over-asking "burned his overnight window".

5. When to stop and re-plan

Stop and re-plan when any of these is true:

- **The same general approach has failed three times** (digest-A 2.8).
 - **A fix did not stop the symptom.** First establish which build ran; then treat your cause as a hypothesis (digest-A 7.6).
 - **Your explanation is plausible but not reproduced or quantified** (digest-B P4.4 item 6).
 - **An independent report contradicts your own result** (digest-B P3.8).
 - **Evidence you are relying on turns out to be a count from a different question**, a document that repeats another document, or an older or newer version of the record than the claim refers to (P1, P16).
 - **The work has drifted outside the scope you were given** (P15).
 - **The budget for the task is spent** (P5).
 - **A long session has been compacted repeatedly.** A self-assessment recommended clearing and re-planning instead; this is a recommendation, not an adopted rule (digest-A 4.11).
-

What this costs, and when not to use it

The method's credibility rests on reporting its failures with the same care as its wins. Each row names the failure, its source, and the pattern where it is told in full. Items marked "corrected" were found by measurement during this research and fixed the same day; they stay here because nothing had noticed them until a measurement ran.

#	Cost or failure	Detail and source	Told in
1	Rules written down did not prevent the failures they describe	The rulebook says its cardinal rule "is not self-enforcing" (digest-A 1.4); ADR-0026: "Intent was never the problem; enforcement was" (digest-B P2.4); nearly every rule carries a dated incident (digest-A 9.1)	P8, P12
2	Rules written and not applied	Punctuation rule with nine violations from the first phase; a recorded SSH client trap "simply was not applied" after two wrong diagnoses; a launch-checklist step deferred "and the release went out anyway" (digest-B F5.6)	P12, P10
3	Loops that nothing runs	Lessons loop, session-start checklist, and the global deferral rule are instruction text only (digest-A2 9.6)	P8
4	A wrong figure reached employers	A stale competition record was in applications already sent; it had passed review and was caught by the owner, not a check; sent applications keep the wording (digest-B F5.1; DOC-STATED)	P16
5	Unanchored figures reached customers and the public	A customer-facing accuracy figure stood from 2026-04-02 to 2026-09-13; a public docs repo still carried a stale test count, inconsistent analyzer counts, a retired channel and three unanchored figures, "Decision needed" (digest-B F5.1; FA; DOC-STATED)	P16
6	Defects shipped in releases	A server command could not start in any published binary across several releases until a later build guard caught the class (digest-B P1.11; DOC-STATED); a disk-erase release was published and pulled (digest-B P4.5, costs item 12; MEASURED)	P2, P3
7	Agents overstepped scope, and over-asked	Four repositories changed by a session told to survey, one with product code, one without version control; the read-only rule came after (digest-B P3.6, F5.4); the opposite failure: "kept reverting to me" (digest-B P3.10)	P15, P13
8	Instruction files bloat	Global file 283 lines when a critique advised about 180 because runbook sections "eat ~80 lines of context in every single session"; now 565 (digest-A 9.5); porting hit a silent 32 KiB limit, rules "~51 KB raw and ~44 KB after filtering" (digest-A 3.4)	P17, R15

#	Cost or failure	Detail and source	Told in
9	Instruction files contradict themselves	59 lines held the dash the file forbids (corrected 2026-09-26, lead's update); "five steps" introduces six; two rules each claim to rank above everything; an unmodified template loads beside the global rules; instructions pull opposite ways with no tie-breaker (digest-A2 9.10, 9.11, 9.12, 9.19, 9.20; digest-A 9.4)	P6, P9
10	Configuration accumulates	Every "Yes, don't ask again" is promoted, including one-shot commands, two contradicting the SSH rule (digest-A2 9.2; digest-A 5.4); sync is additive-only, so removals are manual per machine (digest-A2 9.13)	P13
11	Dead, stale or missing config	Ignored tool-server block in shared settings; a template hook pointing at a missing file; a health check covering 4 names of 11 managed servers (digest-A 9.3, 9.4; digest-A2 9.7); a change journal the rules call for still off; a described keybindings file absent (digest-A2 9.7)	P12
12	Wrong platform facts in instructions	The drive's file system was stated wrongly; measured and corrected 2026-09-26 (digest-A2 9.9; lead's update)	P14
13	Clutter in source-of-truth folders	Ten hand-named settings backups, a retired map file, a plan marked "Awaiting sign-off" whose review says complete, stale guide copies (digest-A2 9.17)	R15
14	Silence means both "fine" and "broken"	Errors swallowed by design; a guard enforced nothing "while appearing installed"; a self-heal log without timestamps or machine names (digest-A2 9.14)	P12
15	Removable-drive single point of failure	Without the drive, a primary-machine session starts "with no global instructions and no memory" (digest-A2 9.8); an index truncated its newest entries silently (digest-A 4.6)	P9
16	Review and manual cost	Independent audit "roughly doubles the review effort" (digest-B P1.10); the ledger question "A person has to" answer (digest-B P1.2)	P4, P16
17	Wrong diagnoses still cost hours	About eight hours on an out-of-memory misdiagnosis (digest-B P2.1); an evening on a hook failure still unexplained (digest-A 7.6)	P7
18	Gates red by design or passing with a skip	7 open deferrals keep one gate red; one milestone passed with a skipped criterion (digest-B P2.5, P1.7); "Six tests that cannot fail" shipped as a deferral (digest-B P1.5)	P10, P3

#	Cost or failure	Detail and source	Told in
19	Many agents cost infrastructure	A crash lost an audit consolidation (digest-B P3.3); orphaned processes held 80.57 GB of commit charge, "Out of memory" at 52% RAM; worktrees inside the repo were staged (digest-B P3.9)	P15
20	Two agents on one rulebook	A session-start hook still failing with root cause "UNKNOWN"; an imported hook "actively failing", left in place and documented (digest-A2 9.18); four interpreters needed on every machine (digest-A2 9.15); memory fragmented by folder path (digest-A2 9.16)	P9, P12
21	Documents drift under the method	A progress file contradicted itself in four places (digest-B P2.8); a handoff count stale by twenty-six (digest-B F5.7); a checker banned a correct fact (digest-B P1.3); a project instruction file overclaims its license signing (digest-B F5.1); a researcher checked a dated figure against a newer record (P1)	P1, P11, P15

When not to use the method at all

This is the playbook's judgement from the costs above, not a measured threshold:

- **Throwaway work** (a one-off script, a quick exploration) needs **P1, P6, P13's** forbidden tier, and **P14**. The gates, ledgers, ADRs and independent audits cost more than they save.
 - **A single short session** does not need memory sync, deferral registers or live handoffs.
 - **One machine, one tool** does not need any of the multi-machine or two-agent machinery. Most of rows 10 to 15 and row 20 come from that machinery.
 - **Do not copy this rulebook wholesale into your own instruction file.** Start with **P1, P6** and **P7**, add a rule only after a real incident, and move runbook-grade material out of the always-loaded file (**R15**).
 - **Do not adopt a rule because it is in this document.** Adopt it when you have had the incident it prevents, or when the stakes (regulated data, money, public claims, security controls) make the incident unaffordable. That is how the method was built.
-

Glossary

Plain-English definitions for the terms this playbook uses. Adapted from digest-A2's glossary; terms marked (added) are new here.

AI and agents

- **LLM (large language model):** A program trained on large amounts of text that predicts and writes text.
- **Prompt:** The message or instructions you give a model.
- **Token:** A small chunk of text (roughly part of a word); limits and costs are counted in tokens.
- **Context window:** How much text a model can hold in mind at once in one conversation.
- **Context compaction:** The tool summarising older parts of a long conversation to make room; details can be lost.
- **Agent / AI coding agent:** An LLM that can act: read and edit files, run commands, and keep going across many steps.
- **Claude Code / Codex CLI:** Command-line coding agents from Anthropic and another vendor; Claude Code is the worked example here.
- **Session:** One run of the agent, start to finish, with its own conversation.
- **Subagent:** A helper agent started for one focused job, which reports back.
- **Fork:** A subagent that starts with a copy of the parent's conversation.
- **Handoff:** The report passed to the next worker so they can continue.
- **Tool / tool call:** A capability the agent can use; one use of it.
- **Harness:** The program around the model that runs tools, applies permissions and fires hooks.
- **Transcript / JSONL:** The saved record of a session, including every tool call; JSONL is the one-record-per-line file format transcripts use.
- **Prompt injection:** Text in a page, file or tool output that tries to give the agent orders. Treat it as data.
- **Plan mode:** A Claude Code mode that investigates and plans but cannot change files until the plan is approved.
- **Skill:** A packaged set of instructions the agent loads when a task matches it.
- **In lane / in-lane (added):** Inside the task and scope the agent was given.
- **Scope lock (added):** A restriction limiting a helper to named files or directories.
- **Worktree (added):** A separate working copy of a git repository, often one per parallel agent.

- **Skip-permissions / auto-approve mode (added):** A mode in which the agent runs tool calls without asking; see P13, "Unattended runs".
- **Container / VM / sandbox (added):** An isolated environment (a container, or a virtual computer) that holds only what a task needs, so a mistake cannot reach the host's files or credentials.

Instructions, memory and records

- **CLAUDE.md / AGENTS.md:** Standing-instruction files read at the start of every session by Claude Code and Codex respectively.
- **Memory store:** A folder of small notes the agent writes for later sessions.
- **MEMORY.md / memory index:** One line per note, loaded at session start.
- **Frontmatter:** A labelled block of fields at the top of a note.
- **Lessons file / todo file:** Per-project lists of mistakes-and-rules, and of the current plan and progress.
- **Provenance:** A record of where something came from and who made it.
- **Paraphrase vs quote:** Your own words versus the exact words; meaning shifts in paraphrase.
- **ADR (architecture decision record):** A short numbered document recording a decision, the options, the reasons and the cost.
- **Questions file (added):** An append-only log of open questions with the current assumption and when each must be settled.
- **Claims ledger (added):** The only permitted source of factual claims for outward-facing material.
- **Measured / doc-stated / inferred (added):** The three evidence labels: counted or read directly; a document says so; an interpretation.
- **Ground truth (added):** The actual thing (bytes on disk, the live system), not a report about it.

Configuration and automation

- **Hook:** A script the harness runs automatically on an event such as session start, prompt submit, before a tool call (and able to block it), or end of turn.
- **Exit code:** The number a program returns when it ends; 0 usually means success. In Claude Code hooks, 2 blocks (check your tool's docs).
- **Standard output (stdout):** A program's normal text output.
- **Permissions / allow list / deny rule:** Rules for which tool calls run without asking, and which are always blocked.
- **"Yes, don't ask again":** The button that saves a permanent allow rule.

- **MCP (Model Context Protocol) / MCP server:** A standard way to plug extra tools into an agent, and a program offering them.
- **Self-heal:** A start-up script that checks required setup and repairs it.
- **Idempotent:** Safe to run repeatedly; a repeat changes nothing.
- **Additive sync:** Copying that adds and updates but never deletes.
- **Platform filter:** A step that drops settings the current operating system cannot run.
- **Fail closed / fail open:** On error, a check blocks the action, or lets it through.
- **Pinning a version:** Fixing a tool to one exact release.
- **Derived file:** A generated file; hand edits are overwritten.
- **Lint / CI (added):** An automated style and rule check; continuous integration, the server that runs checks on every change.

Testing and verification

- **Test suite / green build:** The automated tests; green means all passed, which proves only what they check.
- **Invariant:** A condition that must always hold.
- **Cross-product testing:** Testing every combination of inputs.
- **Sentinel:** A stand-in value meaning "not set"; a bug if confusable with a real zero or empty.
- **Mutation testing / un-fix:** Introducing small bugs to check tests notice; reverting a specific fix to confirm a test goes red (un-fix added).
- **Positive / negative control:** A check run on something known present, proving it can find it; a known covered control removed, proving the harness detects its absence (negative control added).
- **Probe:** A check for whether something exists.
- **Gate:** An automated check that must pass before a milestone counts as done.
- **Milestone / deferral:** A planned checkpoint; a decision to do something later, tied to a milestone or event.
- **Regression:** Something that used to work and no longer does.
- **Root cause / hypothesis:** The proven reason for a problem, versus an unproven explanation.
- **Bisect:** Finding a fault by repeatedly halving the possibilities.
- **Count vs rate / denominator:** How many, versus how many out of a total.
- **Three strikes (added):** After three failed attempts at one approach, stop and pivot.
- **Live call (added):** The first real contact between code and an external system.
- **JSON / regex / parser:** A structured text data format; a compact text-matching pattern language; a program that reads structured data by its real rules (safer than a regex for JSON).

- **Segfault / OOM / commit charge (added):** A program crash from an invalid memory access; out of memory; the Windows measure of memory promised to running programs.

Git and code workflow

- **Repository (repo):** A project tracked by version control (git).
- **Commit / branch / push:** A saved snapshot; a separate line of work; sending commits to a shared server.
- **main** : The primary branch.
- **Line endings, autocrlf:** Windows and Unix end lines differently; a git setting that converts between them can silently alter files.
- **Technical debt:** Future work created by a shortcut now.

Security, privacy and people

- **Guardrail:** A rule or mechanism that keeps the agent from doing something unsafe.
- **Human-in-the-loop:** A person must answer or approve before the agent continues.
- **AFK / away mode:** Away from keyboard; questions and approvals go to a phone.
- **Secret / API token / rotation:** A password-like access string; replacing a leaked one everywhere it is used.
- **Persona:** A separate public identity kept apart from a company one.
- **HIPAA / PHI:** The US health privacy law; protected health information.
- **Synthetic data:** Made-up data shaped like the real thing, for testing.
- **Egress control:** A rule limiting what data can leave a system.
- **Pentest / active scan / scope:** An authorized break-in attempt; a scan that probes a target directly; the written list of approved targets.
- **TLS / certificate chain:** The encryption behind https, and the certificates proving a server is genuine.

End of playbook. Procedures referenced as R0 to R15 are in `RUNBOOK.md`.

Built from PLAYBOOK.md on the date shown in the document. Evidence labels:

MEASURED counted or read in a result file; **DOC-STATED** a document says so;

INFERRED reasoned, not measured.