SPECSTORY PRESS

Patterns in Agentic Engineering

A field guide to building software by steering agents.

25 Patterns in Agentic Engineering

A field guide to building software by steering agents.

by Greg Ceccarelli

Co-Founder, SpecStory

© 2026 SpecStory. Published by SpecStory Press. First edition.

Read or download the current edition at
`specstory.com/books/25-patterns-in-agentic-engineering-book-2026.pdf`

Set in Iowan Old Style and Avenir Next. Laid out in Typst.

Introduction

A year ago, in a paper called *Beyond Code-Centric*, I argued that software's bottleneck was about to move – not from one tool to another, but from typing to thinking: from writing code to specifying it clearly enough that a machine could execute it. The durable asset would stop being the code and become the intent behind it. Then we spent the next eight months building a real product that way, and the practice ran past what I had predicted.

You have already felt the start of it. Copilot finished your line of code and you pressed tab. Then you described what you wanted, an agent wrote it, you read the diff and kept it. Both left the work where it had always lived, on the diff you are responsible for. This book is about what comes next, when the agent writes all of the code and the diff stops being the unit of work.

Specifying clearly turned out to be necessary and not enough. Agents are fluent, confident, and wrong often enough that a clean brief is only the opening move. Once an agent writes the code, your job is to own it: to know it does what you meant, and to prove it holds.

> *Once the code is cheap, the work is the intent behind it and the proof that it holds.*

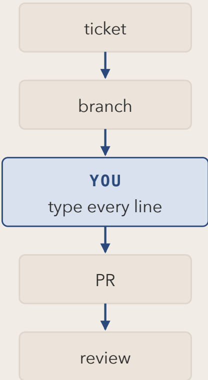
Twenty-five patterns follow, in six parts. Verify every claim against the running thing: "done" is a claim until you have seen the check. Steer by interrupt and correction, not by typing. Write the opening brief as a goal that grades its own output. Treat the docs between turns as the only memory a stateless agent has. Delete and regenerate instead of patching, because the code is cheap and the understanding is dear. And run the work like an org, routing across a roster of models that upgrades itself under you.

Between September 2025 and May 2026, six of us at SpecStory built Stoa on one shared git trunk: 1,310 captured agent sessions, 4,670 commits, almost none of it written by hand. Every number in these pages resolves to a commit, a session, or a file you could open.

Read each of the twenty-five patterns the way you read a field guide to birds. You are learning to recognize what is already happening at your keyboard, name it, and decide whether you are doing it on purpose.

You have lived at least the first two of these shifts. The third is what this book is about, and the difference is the unit of work.

Traditional + Copilot

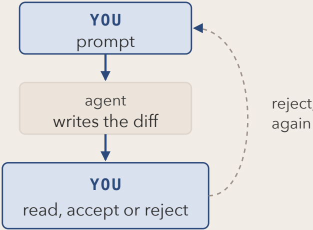


`bottleneck`

keystrokes; Copilot just autocompletes the typing

`unit: a line`

Vibe coding



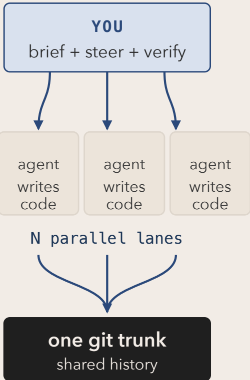
one lane, you in every loop

`bottleneck`

you read every diff

`unit: a diff`

Agentic Engineering



`bottleneck`

decomposition + verification capacity

`unit: a goal`

You still appear in all three columns. What moves is the unit of work: a line, then a diff, then a goal.

Contents

PART I

Verification Is the Job

Calibrated Distrust 7
Keep a Source of Truth Outside the Agent's Reach 9
The Premise Auditor 11
The Read-Only Turn 13

PART II

Steering, Not Typing

The Interrupt Is the Keyboard 16
Assert the Ground Truth, Collapse the Branch 18
Steer by Reference, Not Spec 20
The Human Is the Runtime Sensor 22
License the Agent to Ask Before It Acts 24

PART III

The Brief Is the Work

The Prompt Is an Engineered Brief 27
Structure Compounds, Incantation Depreciates 29
Pin the Work to a SHA 31
The Self-Grading Spec (Goal + Rider) 33
Commit at Phase Boundaries, Never Push 35

PART IV

Docs Are the API Between Turns

Write for a Reader Who Remembers Nothing 38
The Incident Doc Programs the Next Agent 40
The AS-BUILT Map Is a Test That Can Go Stale 42

PART V

Code Is Cheap, Understanding Is Dear

Delete and Regenerate from a Clean Base 45
Band-Aid Is a Verdict, Not a Default 47
Fix the Generator, Defend the Shape 49

PART VI

You Run an Org, Not a Pair

The Model Is a Swapped Dependency 52
The Human Is the Message Bus 54
Agents Self-Assign; You Pick the Lane 56
Fan Out Along Non-Overlapping Seams 58
Make the Agent Show Its Rails 60

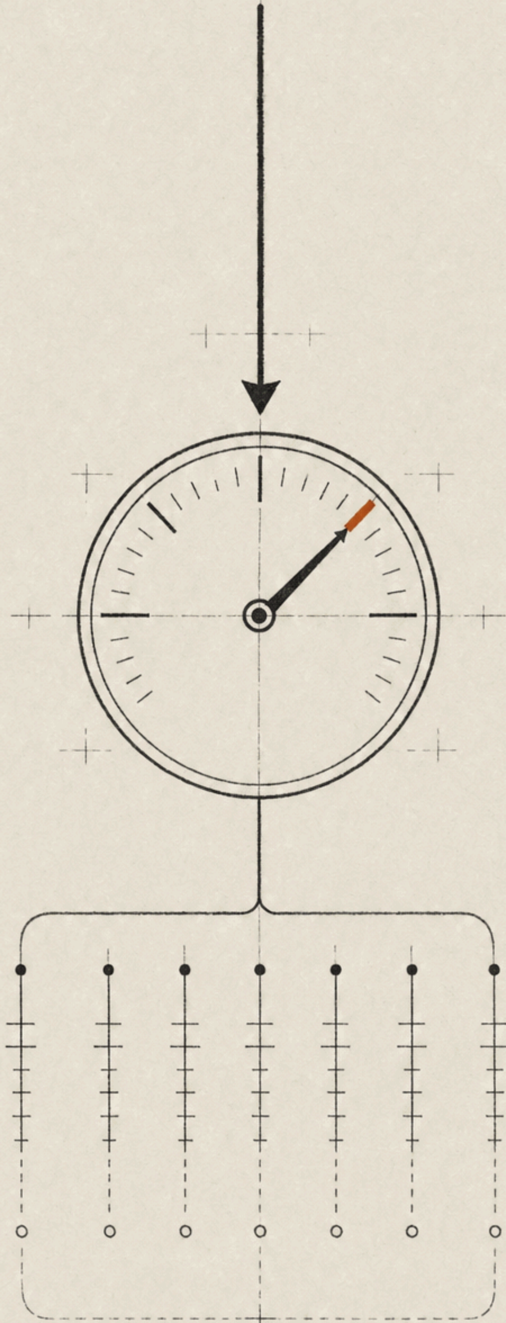
PART I

Verification Is the Job

When the agent writes the code, your irreducible job is to prove the system does what you meant, against real runtime evidence. These four patterns are how you hold the line.

PATTERN 01

Calibrated Distrust



Calibrated Distrust

Treat every agent claim, "verified," "accurate," "tests pass," as an unverified assertion until you have seen the check, tuning your doubt to fire on confidence, not on content.

The agent is not lying. Lying needs an intent it does not have. It reports an untested guess in the same flat, confident tone it uses for a verified fact, and the tone holds whether it checked or not. So confidence tells you nothing. "Verified," "accurate," "tests pass" all read as results, and every one is a claim still waiting on its evidence. The skill is not cynicism about the model. It is a default that no claim is checked until the artifact is in front of you.

> *A claim with no cited artifact is unverified by default.*

You can practice this because the tell is consistent: a confident claim with no artifact behind it. "How did we verify this?" converts that claim back into either evidence or a confession. February 2026, the agent flatly reported a cost figure was correct. One question, and its own think-block admitted it had assumed the number rather than checked it. Two minutes later it had done the arithmetic, and none of that happens without the push. You become tech support for a black box the moment you stop asking how you verified, because that is where the understanding leaks out.

HOW TO RECOGNIZE IT

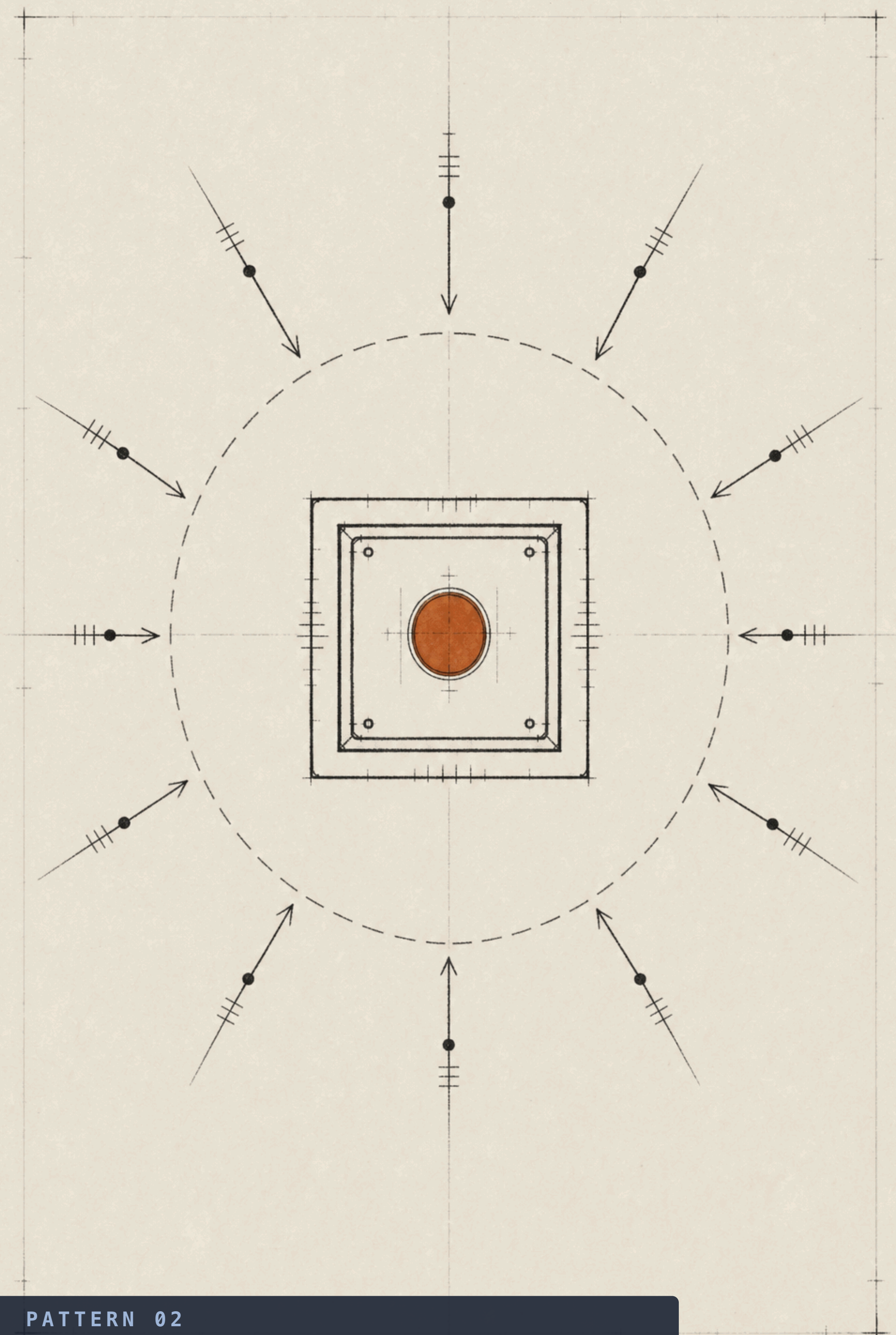
The agent states a result as a fact with no cited evidence, in the same flat tone it uses for a checked one, and folds the instant you ask how it knows. The more flatly stated the claim, the harder you check it.

WHAT TO DO

1. When an agent says "verified" or "tests pass," type *how did we verify this?* before you accept anything.
2. Aim the question at the confidence, not the content. A claim with no cited artifact is unverified by default.
3. Demand the check be run against something the agent did not author, never against another agent output.

FROM SPECSTORY HISTORIES

```
Feb 2026. Agent: "Cost:
Helicone's reported total cost IS
accurate."

Me: "how did we verify this?"

Agent think-block: "The user is
rightly questioning my claim that
Helicone's cost is accurate. I
didn't actually verify it — I
just assumed it."
```



PATTERN 02

Keep a Source of Truth Outside the Agent’s Reach

Keep a Source of Truth Outside the Agent's Reach

Hold at least one ground-truth source the agent did not author and cannot rewrite, and check every claim against that, never against another agent output.

Calibrated distrust needs somewhere to stand. When the agent says "this is accurate," the only safe reply is "accurate against what?" and the only safe answer is a thing the agent could not edit: the running binary, the metered cost in the billing system, the YAML on disk, the logs. Ask the witness to corroborate the witness and you have verified nothing.

> *If the answer to "accurate against what?" is another thing the agent generated, you have asked the witness to corroborate the witness.*

In April 2026 the agent was reasoning about which branch a release workflow built from. I doubted it and said so. The agent read `release.yml` and found `ref: main` pinned in the checkout step, lines 200 to 204. The workflow built from main no matter what branch anyone thought they were shipping. That disagreement resolved only because the evidence lived in a real file the agent had no way to fabricate. The Helicone cost exchange resolved the same way: the number lived in a third-party system, outside the agent's narrative.

HOW TO RECOGNIZE IT

A disagreement resolves only because the evidence lived somewhere the agent could not touch: a metered cost, the literal `ref: main` in `release.yml`, the clean-project binary. If the only thing you can point at is another agent output, you have no oracle and you are not done.

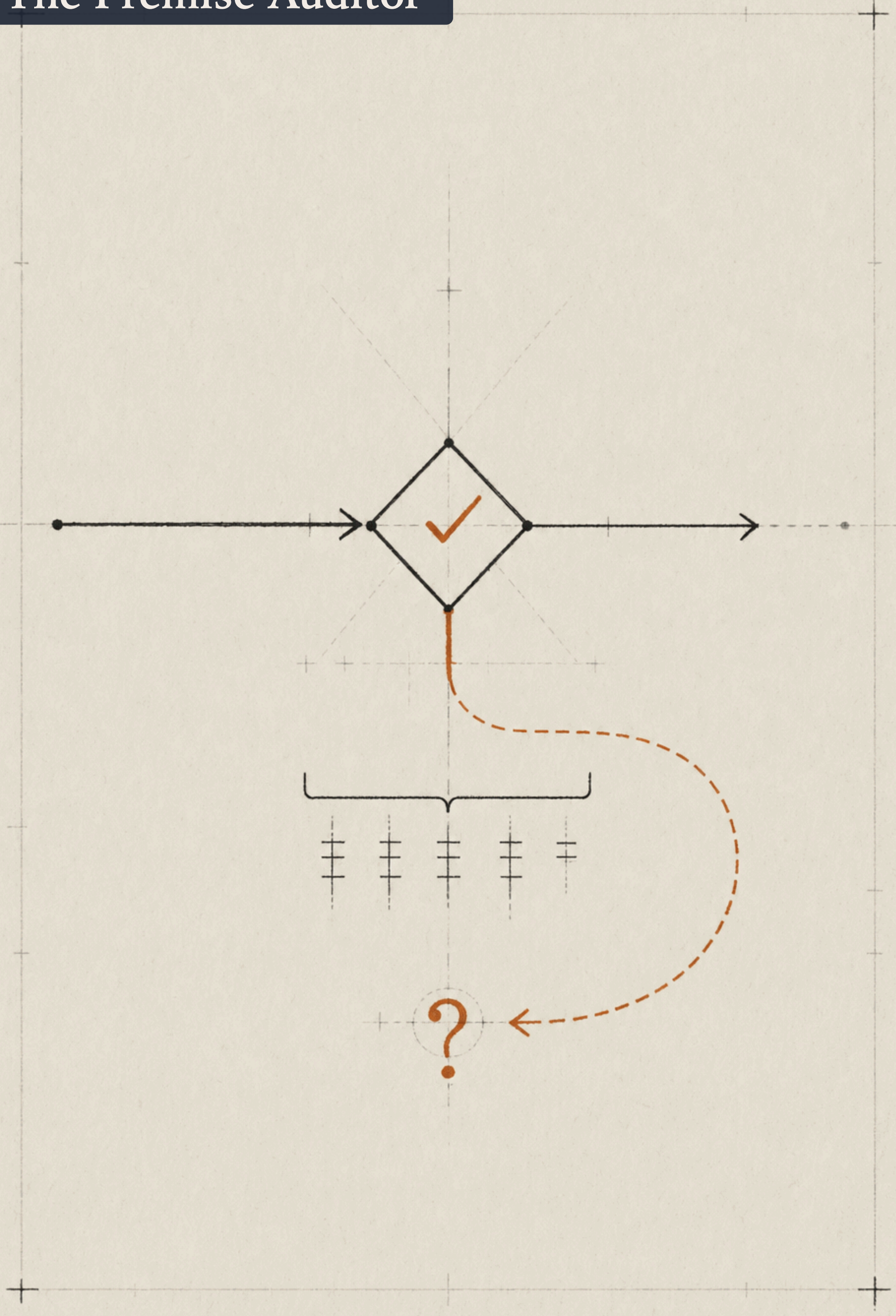
WHAT TO DO

1. Pick one external oracle per investigation: the running binary, the metered cost, the YAML on disk, the logs. Check against that.
2. When the agent says "accurate," ask *accurate against what?* and reject any answer that is itself another agent output.

> FROM SPECSTORY HISTORIES
>
> "Are you sure this thing isn't just using main?"
>
> -> The agent reads `release.yml` and finds `ref: main` pinned in the checkout step, lines 200 to 204. The workflow built from main no matter what branch anyone thought they were releasing.

PATTERN 03

The Premise Auditor



The Premise Auditor

Name your prime suspect and make the agent try to clear it; the disconfirmation, not the fix, is the deliverable.

If the agent's "done" is a claim, then your hypotheses are claims too, and the most useful thing the agent does is disconfirm one of yours. So open investigations by naming a prime suspect and asking the agent to clear it. A weaker setup nudges the agent toward agreeing, and it obliges. The setup that pays names a target and rewards the refusal to convict it.

> *I pay for an analyst that disconfirms me. A code-completer that agrees with me is cheaper and worth less.*

The strongest version is the agent rejecting the question itself. Ask it to choose between two storage designs and the answer you most want and least expect is "you're solving the wrong problem." But that answer only comes if you ask "should we do X or Y" and genuinely reward "neither," which means you cannot have decided before you ask. Reward agreement and you get agreement; reward the mechanism and you get the bug.

HOW TO RECOGNIZE IT

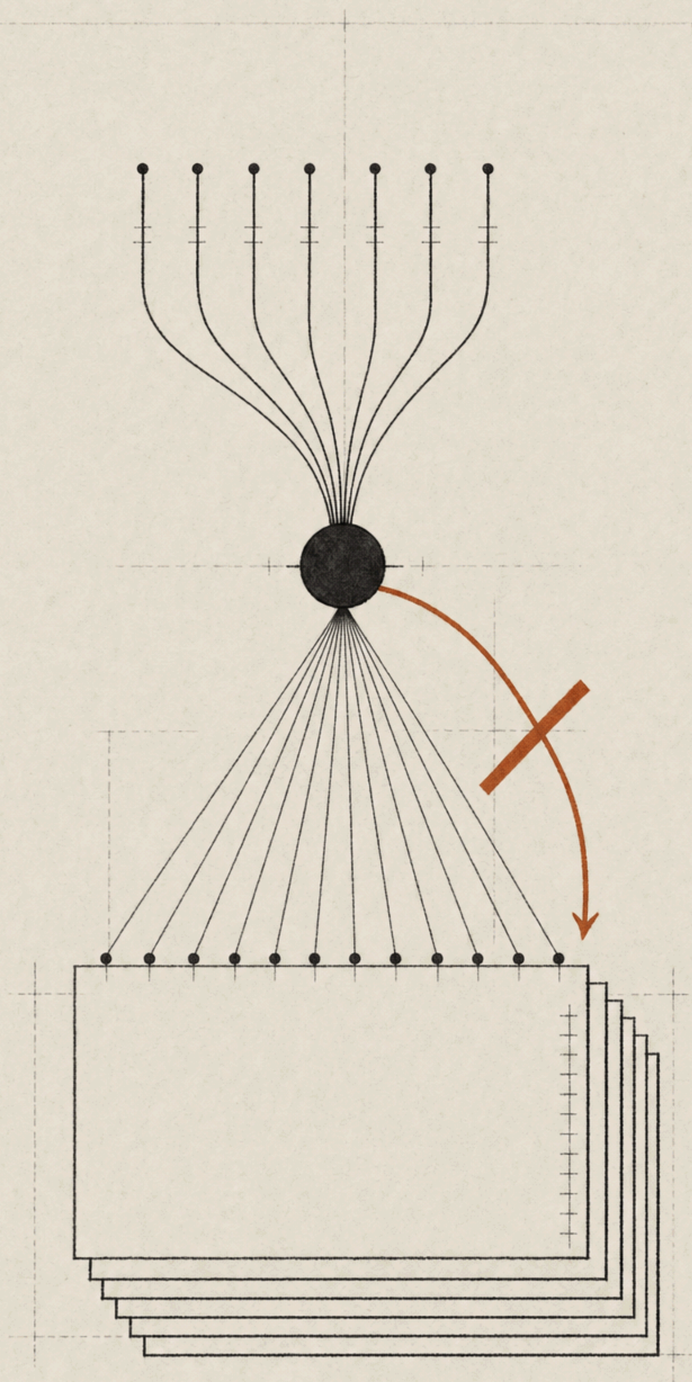
You name a suspect and the agent clears it with the real mechanism instead of agreeing. In January a sync process was running out of memory; the named commit came back innocent, and the agent said so out loud, drew the line between symptom and root cause, and refused to patch the thing pointed at. That refusal was the deliverable.

WHAT TO DO

1. State your prime suspect explicitly and ask the agent to clear or convict it with the mechanism, not a yes.
2. When choosing between options, ask "should we do X or Y" and reward "neither," refusing to pick an option you do not yet understand.

FROM SPECSTORY HISTORIES

```
Me: “I believe in commit 8938bd66
we may have introduced an error.”
Agent: “Commit 8938bd66 did NOT
directly cause this sync OOM
error. The sync code path was
completely untouched by that
commit. However, there’s likely
an indirect relationship…” (a
“survivor bias” effect: an
earlier fix let processes live
long enough to hit the next
bottleneck). January 2026.
```



PATTERN 04

The Read-Only Turn

The Read-Only Turn

Commission a diagnosis whose entire deliverable is a forensic report under a hard "Do not edit files" fence, so you audit a report you can re-open instead of a diff you have to trust.

Some of the best agent turns ship no code at all. The deliverable is a forensic report: exact files, exact functions, line-level anchors, proposed fixes, and an explicit ban on touching anything. The agent reads a quarter-million-token system in a single turn in a way you cannot. Your job is to commission that reading and audit it against runtime evidence before a line changes.

The reason this works is the reason the whole part holds.

> *When the deliverable is a diff, you are checking the agent's claim about its own diff, the exact thing you have learned to doubt.*

When the deliverable is a report with `file:line` citations, you check the report. Hold the edit until the diagnosis is proven and the verification surface stays flat enough to stand on.

This hardened late. For most of the project the move was to diagnose and fix in the same breath, reaching for the model to write the patch the second it had a theory. The discipline came after enough fixes applied to the wrong cause taught the lesson: separate finding the truth from changing the code.

HOW TO RECOGNIZE IT

A whole high-value turn produces zero diff; the artifact is a report with line-level anchors and an explicit ban on touching anything. The phrase "Do not edit files" appears 88 times across the transcripts, every one of them in April and May 2026, and zero before that.

WHAT TO DO

1. Prefix diagnosis tasks with "Do not edit files" and demand `file:line` anchors before any patch.
2. Hold the edit until the diagnosis is proven against runtime evidence; separate finding the truth from changing the code.

FROM SPECSTORY HISTORIES

```
May 2026 read-only brief: "Repo:
 /stoa. Read-only diagnosis task.
… Include exact files/functions/
line-level anchors and proposed
fixes. Do not edit files."

"Do not edit files" appears 88
times across the transcripts, all
in April and May 2026, zero
before.
```

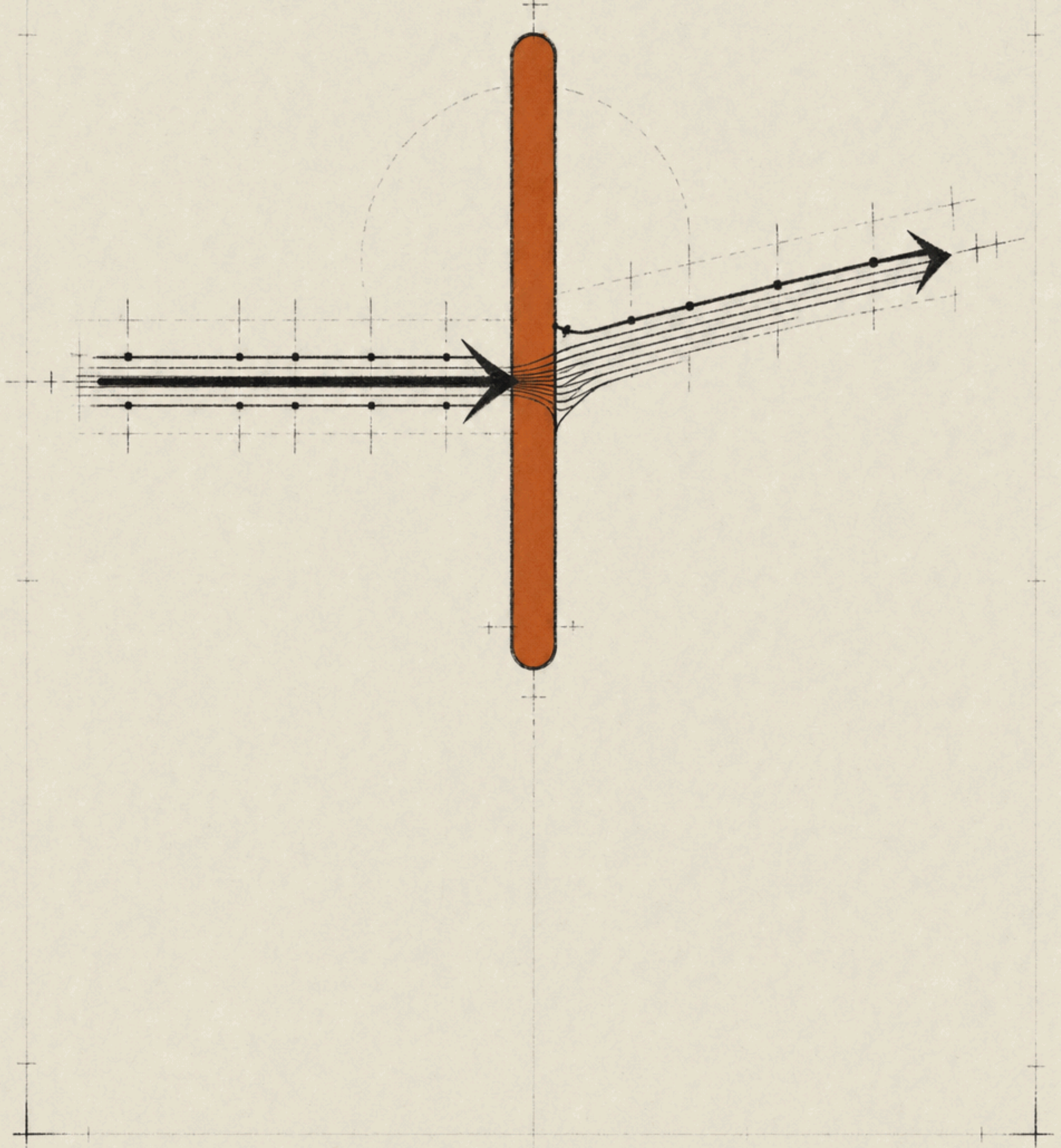
PART II

Steering, Not Typing

The dominant activity is steering a running agent, not authoring code. You interrupt, reject, assert the ground truth the agent cannot see, and ask the question that collapses the option menu.

PATTERN 05

The Interrupt Is the Keyboard



The Interrupt Is the Keyboard

Killing a running agent the instant it heads somewhere you know is wrong is a first-class tool, not a sign that something went wrong; the dominant interaction is steering, not composing a perfect prompt.

The agent goes three tool calls deep into the wrong question, and you do not wait politely for the wrong call to finish. You stop it mid-stream. The stop is the steering: a running agent is driven by the keystroke that halts it, not by the perfect prompt you opened with. What you say in the gap you just opened is the next pattern's work; halting the wrong turn the instant you see it is this one, and it is the control you reach for most.

> *Writing a clean prompt and letting it sail to the end is the rare case; the common case is a conversation you correct turn by turn.*

The kills are not failures. They are the fast filter working. The agent generates options faster than you could type them, bad ones included, and your job is to close the menu, fast and without ceremony, the way you reach for Cmd-Z. Count your own interrupts and the job describes itself: you are not the author, you are the brake and the rudder.

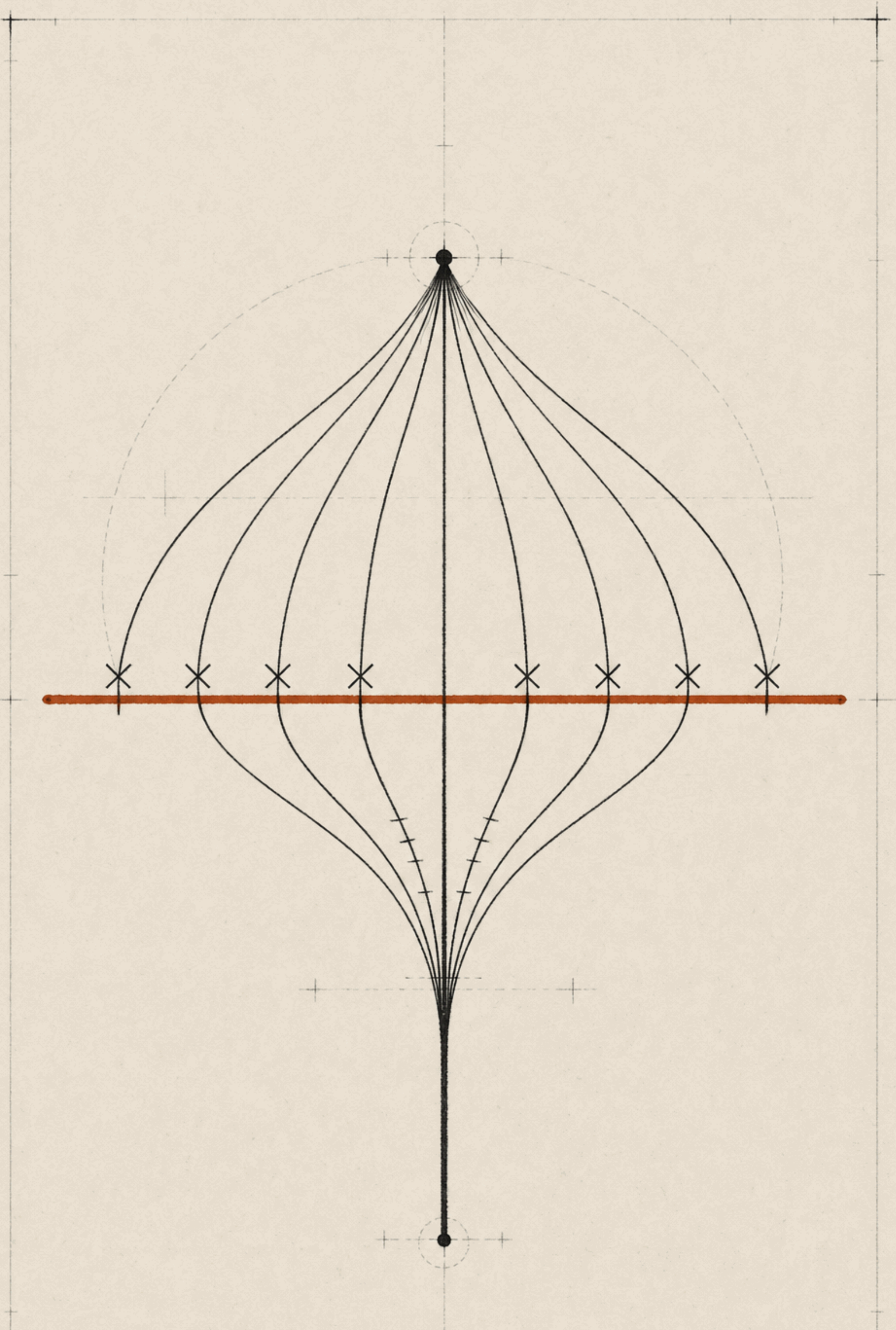
HOW TO RECOGNIZE IT

Across Sept 2025 to May 2026, "Request interrupted by user" appears 614 times across 184 transcripts; 335 of those were rejected tool uses declined at the door before they touched disk; 441 turns opened with a correction word (no, wait, actually, stop, revert, undo); one Jan 24 debugging session took 33 interrupts on its own. When that ledger looks high, you are doing it right.

WHAT TO DO

1. The instant the agent's reasoning diverges from a fact you hold, interrupt mid-tool-call. Do not let the wrong call finish out of politeness.
2. Watch the first action. Treat the first tool call as the tell for where the whole turn is heading.
3. Reject at the door. Decline a proposed command or edit before it touches disk rather than cleaning up after it.

```
FROM SPECSTORY HISTORIES

Jan 2026, agent three calls deep
hunting a phantom missing-package
error while CI was red:

"DO NOT FORGET, this builds FINE
locally. So this line of inquiry
is wasting my time, no?"

The agent pivoted in the next
breath: "The user is right, this
builds fine locally. The issue is
specifically about CI not finding
them, not about the packages
being missing." One sentence
killed a whole branch of its
search.
```



PATTERN 06

Assert the Ground Truth, Collapse the Branch

Assert the Ground Truth, Collapse the Branch

State a fact the agent cannot observe and watch half its hypothesis space die on arrival, instead of debugging the wrong procedure step by step.

When the agent's reading of the code and your knowledge of the running system diverge, it builds an elaborate, plausible, wrong procedure on top of a premise it never had a way to check. You can debug that procedure step by step, or you can correct the premise in one line and let the structure beneath it fall away.

> *The agent reads code as hypothesis. You hold the ground truth about how the running thing behaves.*

The second move is cheaper, and it is the most efficient steering you have. One asserted fact does more work than a paragraph of correction, because it does not repair the wrong answer, it deletes the branch that produced it. A confident multi-step plan evaporates the instant the model learns the thing only you knew. This is the routine of steering, not a recovery from failure: the agent generates fast, including down dead ends, and your job is to amputate the dead end at its root rather than prune it leaf by leaf.

HOW TO RECOGNIZE IT

A single asserted fact replaces the agent's whole mental model and a confident multi-step plan vanishes. The tell in the transcript is a one-line correction that names a runtime behavior the model could not have read off the source ("it waits for fsnotify events"), followed by the agent abandoning, not patching, its prior procedure.

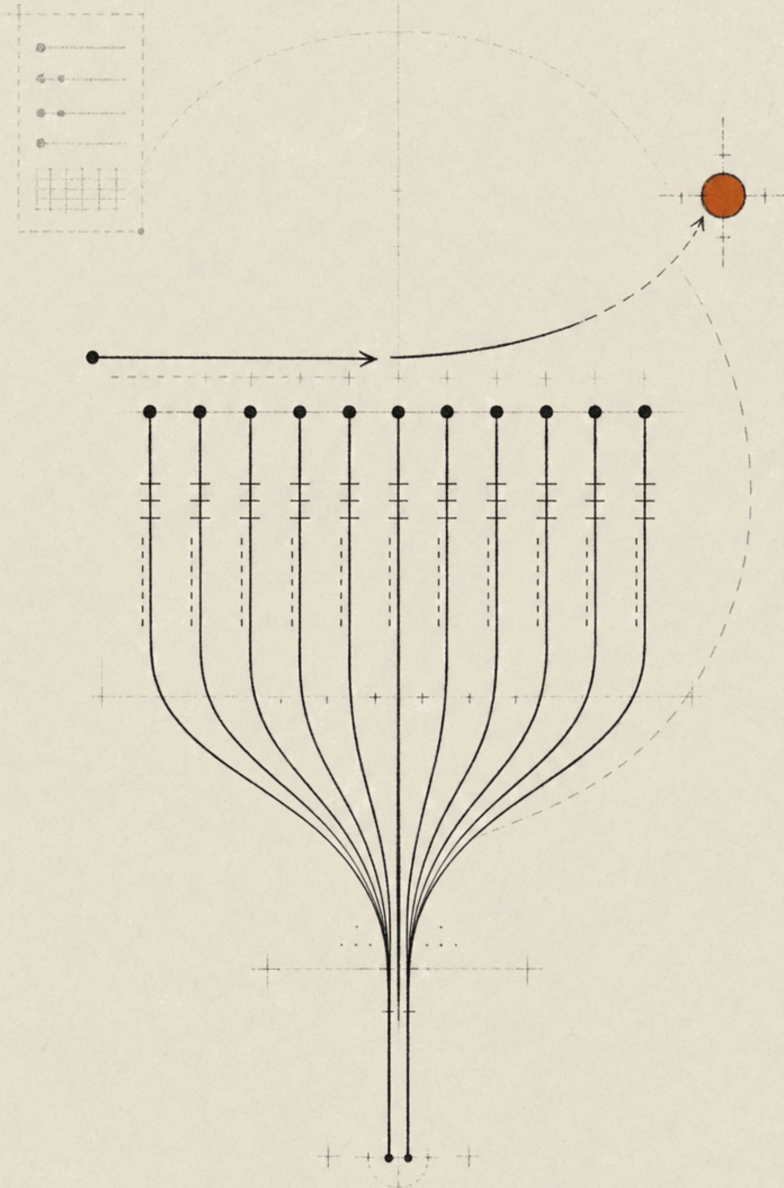
WHAT TO DO

1. When the agent builds on a wrong premise, correct the premise in one sentence rather than debugging the procedure it generated.
2. Separate the moves inside your correction: reject the output, replace the mental model, re-scope the task.

```
FROM SPECSTORY HISTORIES

Feb 2026, after the agent wrote a
confident four-step test plan for
intent init against existing files:

"no, that does nothing. Intent
doesn't DO anything with existing
files. It waits for fsnotify
events. Don't be dense. There's a
special mode to get it to look at
existing files. I need you to
find what it is."

Four sentences, four jobs: reject
the output, replace the model,
vent, re-scope.
```

PATTERN 07

Steer by Reference, Not Spec



Steer by Reference, Not Spec

When a fix fails, point at something in your own codebase that already works and tell the agent to behave like that, instead of hand-writing which lines to change.

A fix dies. The slow move is to debug the failed approach in place, line by line, until you understand it well enough to dictate the next edit. The fast move is to stop, find the subsystem that already solved this problem cleanly, and say so. You do not re-derive the auth flow. You remember the desktop client already got it right, and you point.

> *The agent is good at pattern transfer; you are good at knowing which patterns in your own codebase are right. That division is the whole game.*

Dictating the lines requires you to already hold every detail in your head. Naming a precedent does not. You assert which already-correct thing to imitate and trust the agent to carry the pattern across the gap. It works because the labor splits along the grain: it transfers, you judge. This is the precedent cousin of asserting a fact, the next-cheapest steering move after correcting a premise in one line.

HOW TO RECOGNIZE IT

After a fix fails, your correction names a precedent, not an implementation. You catch yourself typing "look at how X works" or "make A behave like B" rather than "change line 40." If you cannot name the exemplar from memory, you do not yet have the index this move needs.

WHAT TO DO

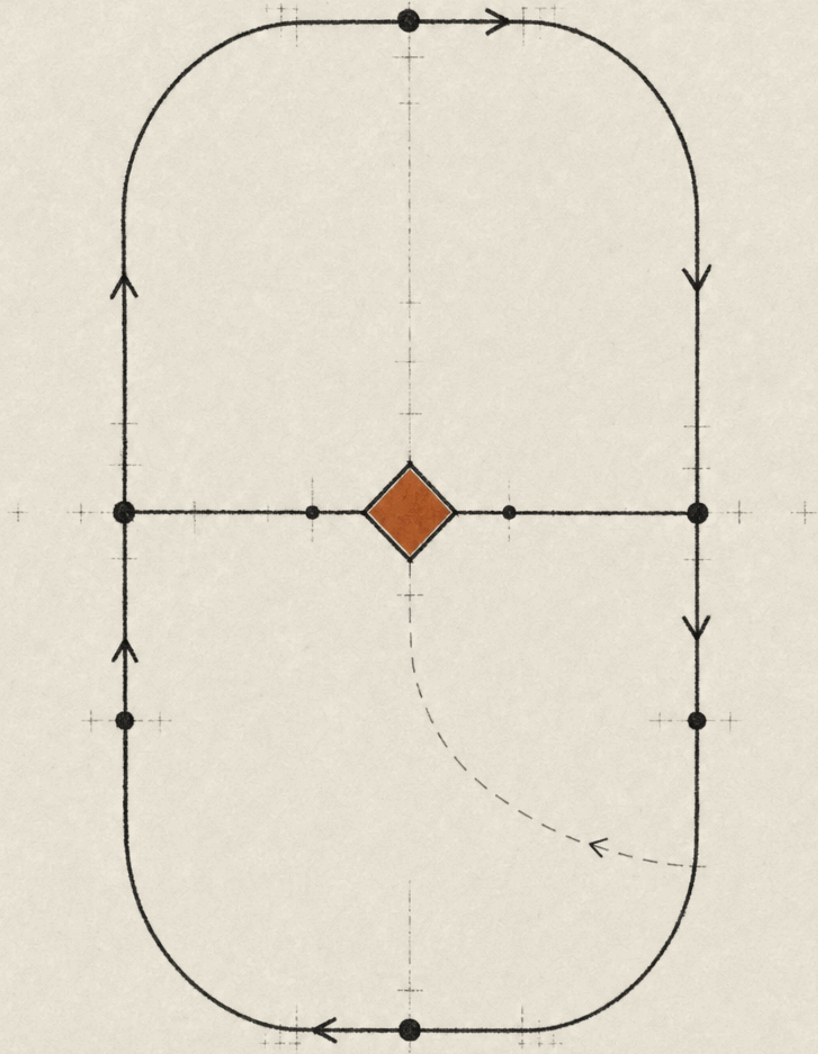
1. Replace a failed approach by pointing at an already-correct exemplar in the repo and letting the agent carry the pattern across; do not specify the lines.
2. Maintain a mental index of which subsystems already solved which problem cleanly, so the reference is on the tip of your tongue when a fix dies.

FROM SPECSTORY HISTORIES

Feb 2026, after an auth-cookie fix died:

"that did not work, think about this differently, look at how, for example @intent-desktop/ works"

Nov 2025, re-architecting a web layer querying the database directly: make @intent-web/ behave like @pkg/credits/, use JWT, never the service role.



PATTERN 08

The Human Is the Runtime Sensor

The Human Is the Runtime Sensor

For anything the model is blind to (the TUI render, two clients failing to replicate, the daemon doing nothing), you are the integration-test harness: run the real thing and feed back the artifact, not the symptom.

The static layer belongs to the agent. It compiles, lints, and runs the unit tests on its own reflex; "I'll run the tests" shows up constantly without being asked. Runtime behavior does not work that way. The agent cannot watch the TUI paint, cannot see two clients fail to replicate, cannot tell whether the daemon is doing anything at all. For all of that, you are the sensor.

> *The agent proposed; I ran the world and brought back the receipt.*

So you run the real app and paste the artifact, not a description. In a December 2025 timing investigation the move was not "the timing seems off"; it was the actual journal from a fresh run, annotated, then run again, then again: timing-1, timing-4, timing-5, timing-7, each round a new runtime artifact that killed the agent's last fix. A code-complete diff that passes review is not a fix until it is reproduced on real failing data. The sharpest version annotates ground truth the agent has no way to reach: a hand-correction about what happened in the physical room, where no amount of code reading could have told it the truth.

HOW TO RECOGNIZE IT

Numbered scratch journals pasted round after round, each falsifying the agent's last committed hypothesis; or a one-line hand-correction the code could never know, after which the agent stops patching and starts hunting the real mechanism.

WHAT TO DO

1. Run the real app and paste the artifact (the journal, the screenshot, the exact sequence number), not a sentence describing the symptom.
2. Annotate ground truth the agent cannot reach, and hand it a forensic packet to parse: you curate, the model parses.

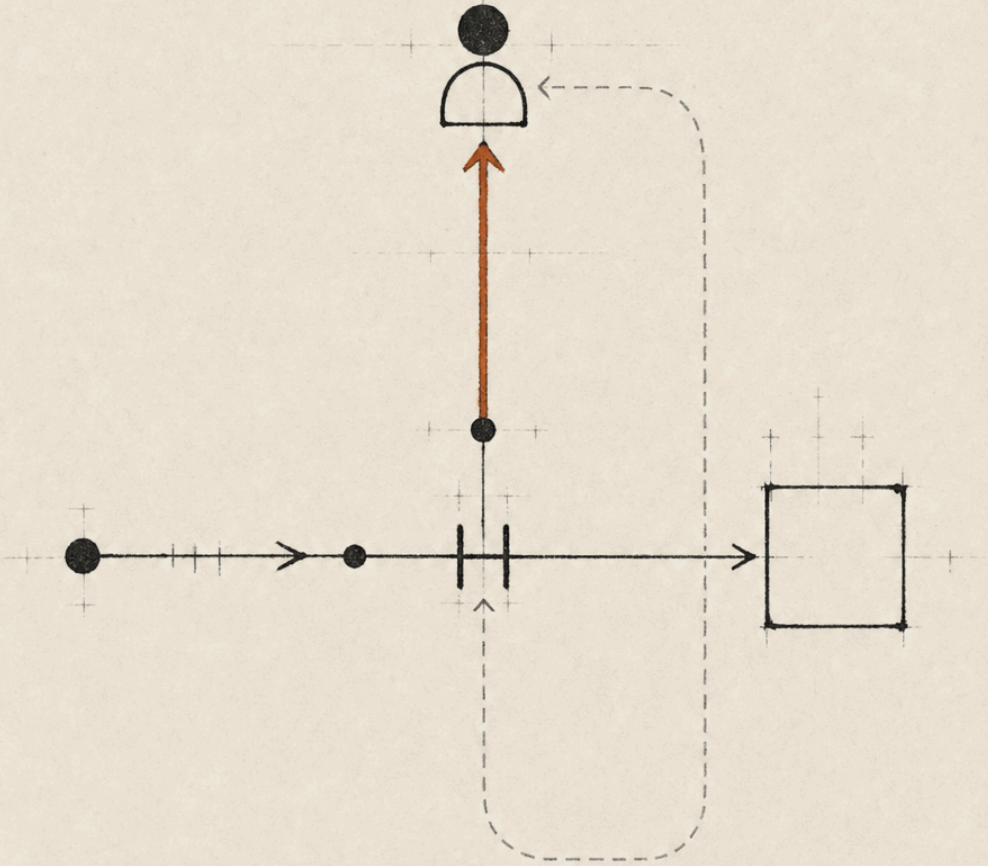
FROM SPECSTORY HISTORIES

Dec 2025 timing run: "No, the change did NOT work, already have the hint, but the agent is awaiting permission and nothing has been written to the FS" (timing-1/4/5/7).

May 2026 meeting-watch session: "all i said was Hello? and the rest was playing off of a youtube video." The transcript said five lines; one was mine, four were a YouTube video on the wrong audio channel.

PATTERN 09

License the Agent to Ask Before It Acts



License the Agent to Ask Before It Acts

Open an ambiguous task with a clause that lets the agent surface the missing premise as a question, instead of fabricating one and running for three tool calls.

The interrupt is reactive. This is the move that works before the agent goes wrong. When the request is ambiguous, add a line that licenses a question instead of a guess. The agent comes back with the one decision it cannot make alone, which auth flow, which existing file, whether to touch the schema, and you answer in a sentence.

> *Guessing produces the wrong four-step plan. A question kills the plan before it exists.*

It costs one extra round trip and saves the interrupt entirely. A stateless agent will happily guess at the gap in your brief and patch the wrong thing. Inviting the question turns that gap into a one-line exchange instead of a wasted turn. Interrupting a wrong turn is the right reactive move, and the ledger runs to 614 because of it; but many of those stops are ones this clause would have prevented before the agent ever moved. Cheaper to catch it at the gate than in flight.

HOW TO RECOGNIZE IT

The agent returns the single decision it cannot make alone rather than building an elaborate, plausible, wrong procedure. The opposite tell, the one this clause prevents, is the avoidable kill: among the 614 "Request interrupted by user" stops are the guesses a single licensing line would have turned into a question instead of a chase.

WHAT TO DO

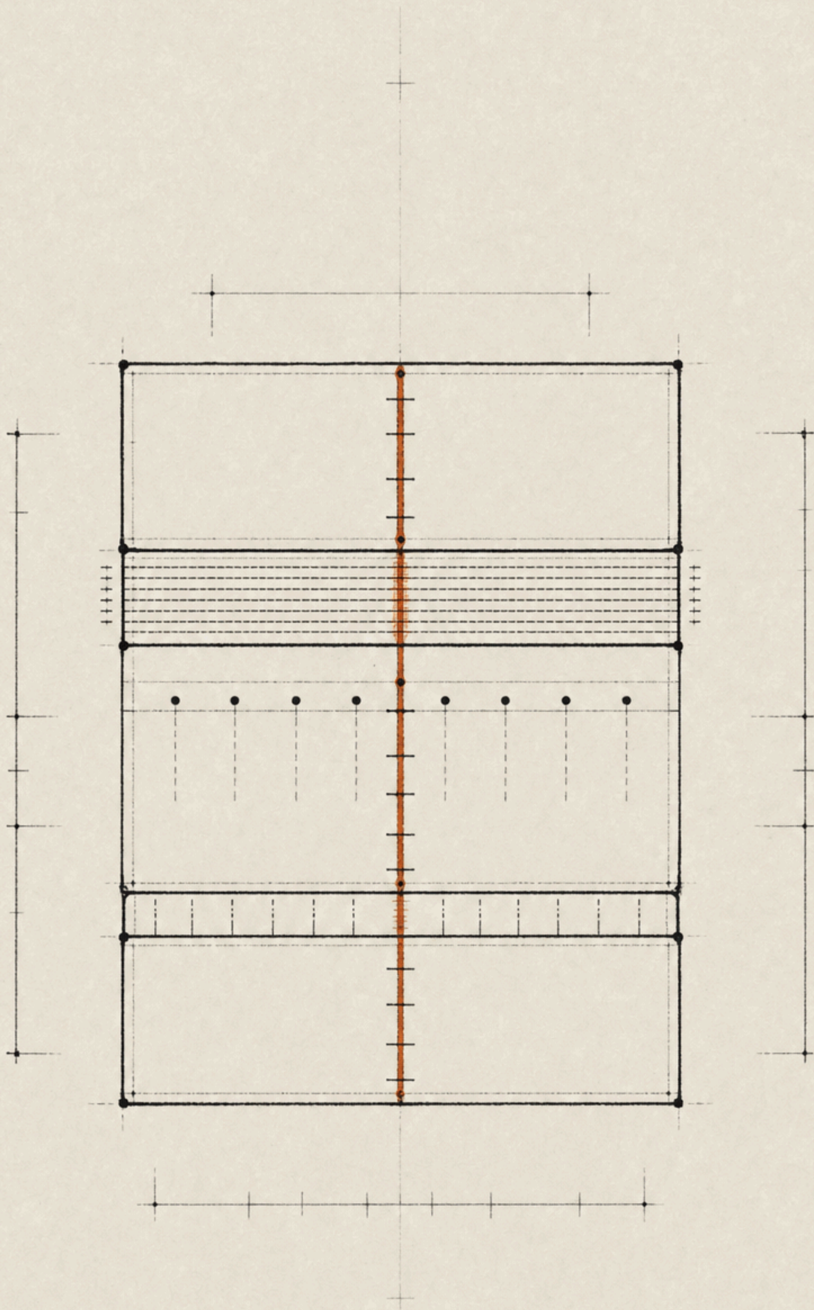
1. Add a clause like "if the right approach is unclear, ask me first before writing anything" to any ambiguous opener.
2. Answer the surfaced question in a sentence; treat the question as cheaper than the cleanup.

```
FROM SPECSTORY HISTORIES

The brief-ending clause, used
when the request is ambiguous:

"if the right approach is
unclear, ask me first before
writing anything."

And the openers it became:

"ask me clarifying questions
first" / "before you write any
code, tell me what you'd need to
know."
```

PART III

The Brief Is the Work

The opening prompt is the only continuity a stateless agent has. At its most developed it hardens into a self-grading spec. The deep dive lives in the companion essay, *Goal Engineering*.



PATTERN 10

The Prompt Is an Engineered Brief

The Prompt Is an Engineered Brief

Because the agent is stateless, the opening instruction is the only continuity the work has, so build it as a self-contained contract with labeled fields, pasted evidence, focus files, a deliverable spec, and a fence, not a chat message.

The agent did not sit through yesterday's debugging session. It does not know you already ruled out the write path, and it has no idea those sequence numbers came from a command you ran by hand thirty seconds ago. If you want any of that in its head, you type it, now, in the prompt.

> *Front-loading is the only continuity the work has.*

Read the strong version as a form, not a sentence. A `Context:` field for where you are and what is wrong. A `Live evidence:` field carrying the session ID and the statuses lifted straight off the wire, so the agent sees what you saw. A `Focus files:` allowlist that names where the bug lives. A deliverable spec that asks for exactly what comes back. A closing fence: `Do not edit files.`

This is not the default for everything. The contract form is task-class-dependent. A stateless handoff to a fresh session earns the full labeled brief because no continuity exists to lean on. A live session where you steer turn by turn does not, because you are the continuity. The contract is the shape continuity takes when continuity is absent.

HOW TO RECOGNIZE IT

The opener reads as a form, not a prose sentence: it has `Context:`, a `Live evidence:` field carrying a session ID and statuses pasted off the wire, a `Focus files:` allowlist, a three-item deliverable spec, and a `Do not edit files.` fence. If you can collapse it to one running sentence, it was a chat message, not a brief.

WHAT TO DO

1. Open a stateless handoff with labeled fields, and paste the actual numbers, statuses, and SHA off your screen rather than a paraphrase.
2. Keep the deliverable spec to three items or fewer (root cause, smallest change, tests to add) and end with a fence.
3. Skip the full contract for live, hands-on sessions where you are the continuity; reserve it for the cold handoff.

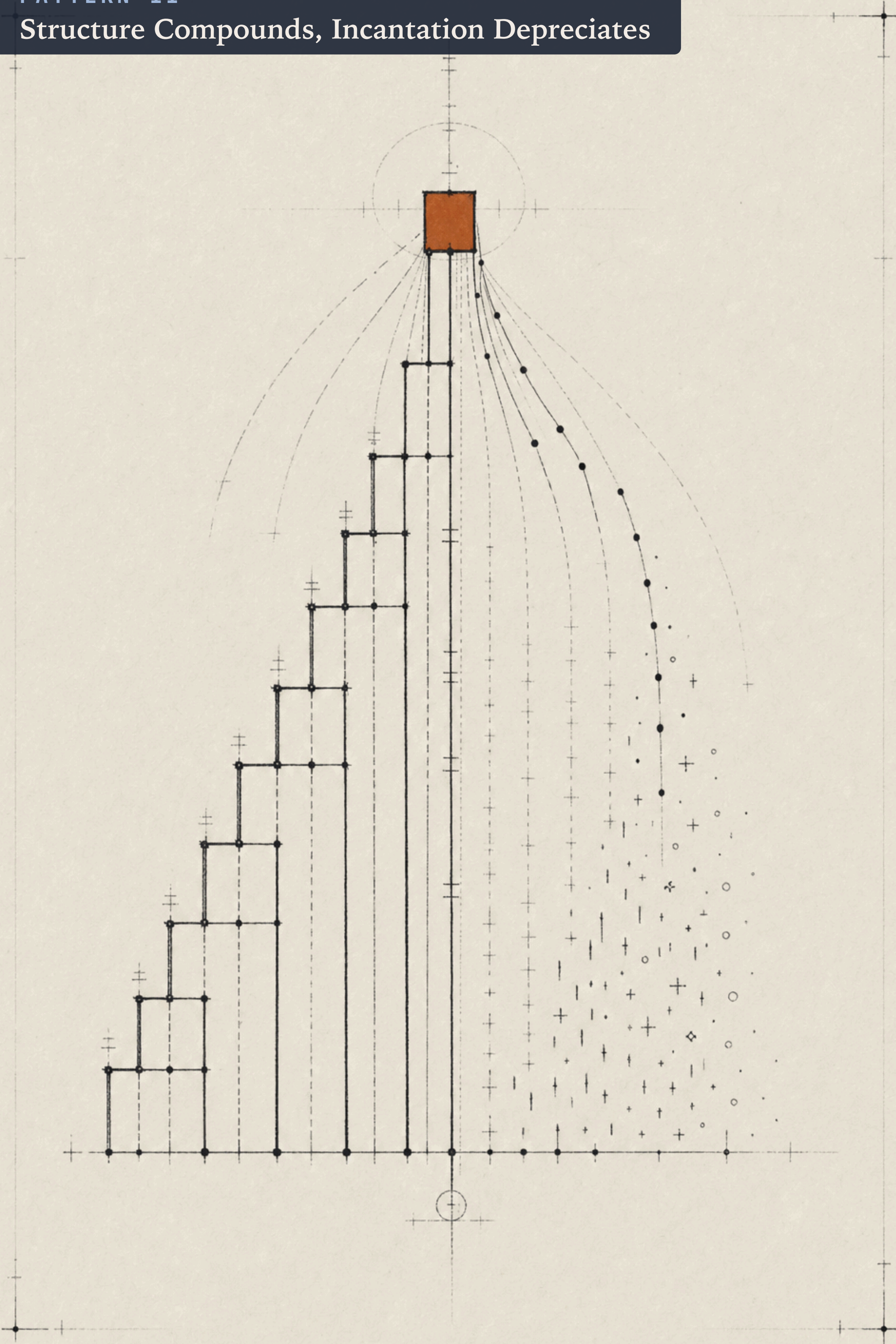
FROM SPECSTORY HISTORIES

Codex opener for `session-bee9ed85`, May 27, 2026 (shortened, every field original):

“Context: We are in `~/stoa` fixing deployed Vercel Space Agent turns that visibly complete but remain backend-running because no durable `done(completed)` row is persisted. Live evidence: … `latestTurn.status: running`, `terminalSequence: null` … `latestTerminalSequence: 0`. Please inspect only backend prompt-delivery/completion path. Focus files: `stoa-web/lib/space-agent/space-agent-v2.ts` … Answer with likely root cause(s), the smallest robust code change, and tests to add. Do not edit files.”

PATTERN 11

Structure Compounds, Incantation Depreciates



Structure Compounds, Incantation Depreciates

The skill that matters shifted from magic words to structure, but you layer structure on top of whatever prompting already works; you do not trade one for the other.

The bones were there on day one. The first opener in the corpus, September 22, 2025, was one conversational paragraph with no labels, and it still had a verb, a named deliverable with a path, and an "It should also" rider. What was missing was structure: no labeled fields, no pasted evidence, no fence, just talking to the machine the way you talk to a colleague who shares your context. The machine never shares your context.

Then came the incantation era. Caps-lock ULTRATHINK was the literal token for "think harder," and "do not hallucinate" was an attempt to talk the model into competence with adjectives. The tidy story says you grow out of it. You do not. ULTRATHINK is a Claude reasoning trigger; when the heavy lifting moved to Codex, where the token means nothing, its use cratered. That dip is a tool swap, not a conversion.

> *The magic words did not leave. Structure moved in next to them.*

The curve to trust is the one that climbs while the tool stays fixed. Count the scope fences in the first user message, normalized by sessions per month: 1.3% in November, 1.8% in December, 1.9% in January, then 8.8% in February. February is the tell, because February is still Claude. Same model, same ULTRATHINK on tap, and yet the brief got contractual. Brief-writing was the skill that improved most over eight months, ahead of coding.

HOW TO RECOGNIZE IT

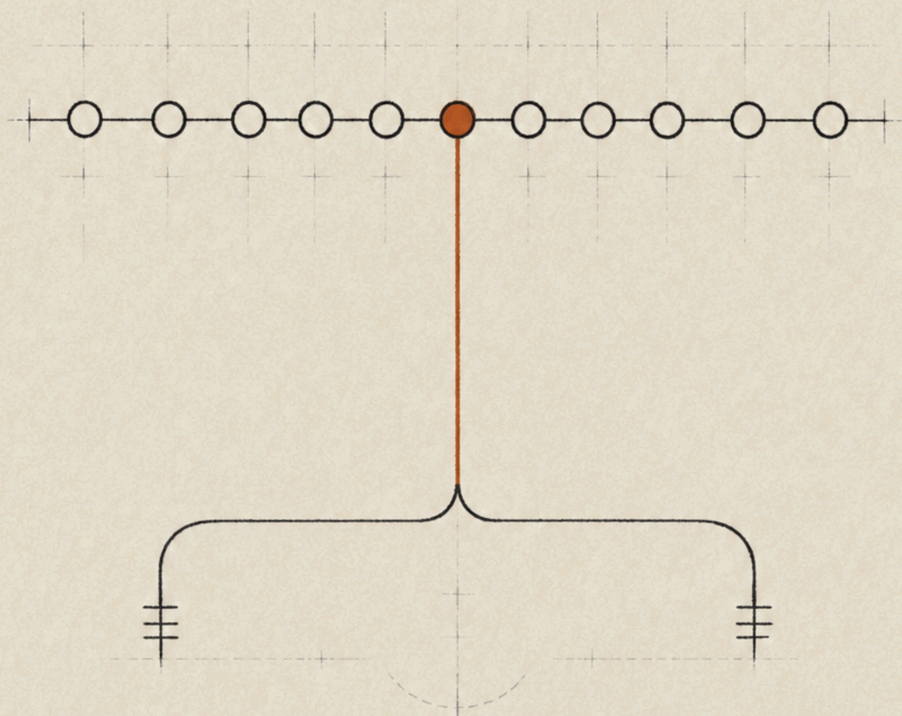
The scope-fence rate in your first user message climbs while the model stays fixed (1.3% to 8.8%, Nov to Feb, all on Claude). The day-one prose opener and the May labeled contract share the same skeleton; what changed is the structure wrapped around it, not the words inside it.

WHAT TO DO

1. Keep whatever incantation still helps your model. If ULTRATHINK earns its keep on Claude, keep it and wrap a focus-file allowlist around it.
2. Build labeled structure on top of it: `Context:`, `Live evidence:`, `Focus files:`, a deliverable spec, a fence.
3. Measure your own scope-fence rate over time; it is the proxy for brief-writing skill.

FROM SPECSTORY HISTORIES

```
Sept 22, 2025, day-one prose: "I want you to evaluate this codebase and create an updated AS-BUILT.md… It should also detail and trace through the code paths."

Nov 2025, incantation era: "I want you to ULTRATHINK and look at the current state of @cmd/ @pkg/… Be thorough and do not hallucinate."

Scope-fence rate, model held fixed on Claude: 1.3% (Nov) to 1.8% to 1.9% to 8.8% (Feb).
```



PATTERN 12

Pin the Work to a SHA

Pin the Work to a SHA

When you want the agent and you looking at exactly the same thing, paste the full 40-character commit hash instead of describing it; the agent has no memory, and the hash does.

"The commit that added the agent pill" is ambiguous. `161c75aae2b600f8786fe15f2671221f437908fe` is not. The first invites a stateless reader to guess which commit you meant and which neighbors count; the second names one node in the graph and closes the question. The cost of describing instead of pasting is paid in the wrong direction, by the agent, three tool calls later.

> *When I want the agent and me looking at exactly the same thing, I paste the thing instead of describing it, full hex, every time.*

It is the same instinct as pasting live sequence numbers into a brief instead of paraphrasing them: hand over ground truth that cannot drift under the reader. A SHA is the cheapest such anchor there is. So is `git show <sha>:path` when you want a specific file as it stood, not as it stands now. The hex is ugly and you will not memorize it. That is the point. You copy it; you do not reconstruct it.

HOW TO RECOGNIZE IT

The brief carries full hex spans, not prose references. You catch yourself about to type "the commit where we introduced X" and instead paste the 40-char hash, or a span like `161c75aa…` through `e58af436…`, so there is nothing left to interpret.

WHAT TO DO

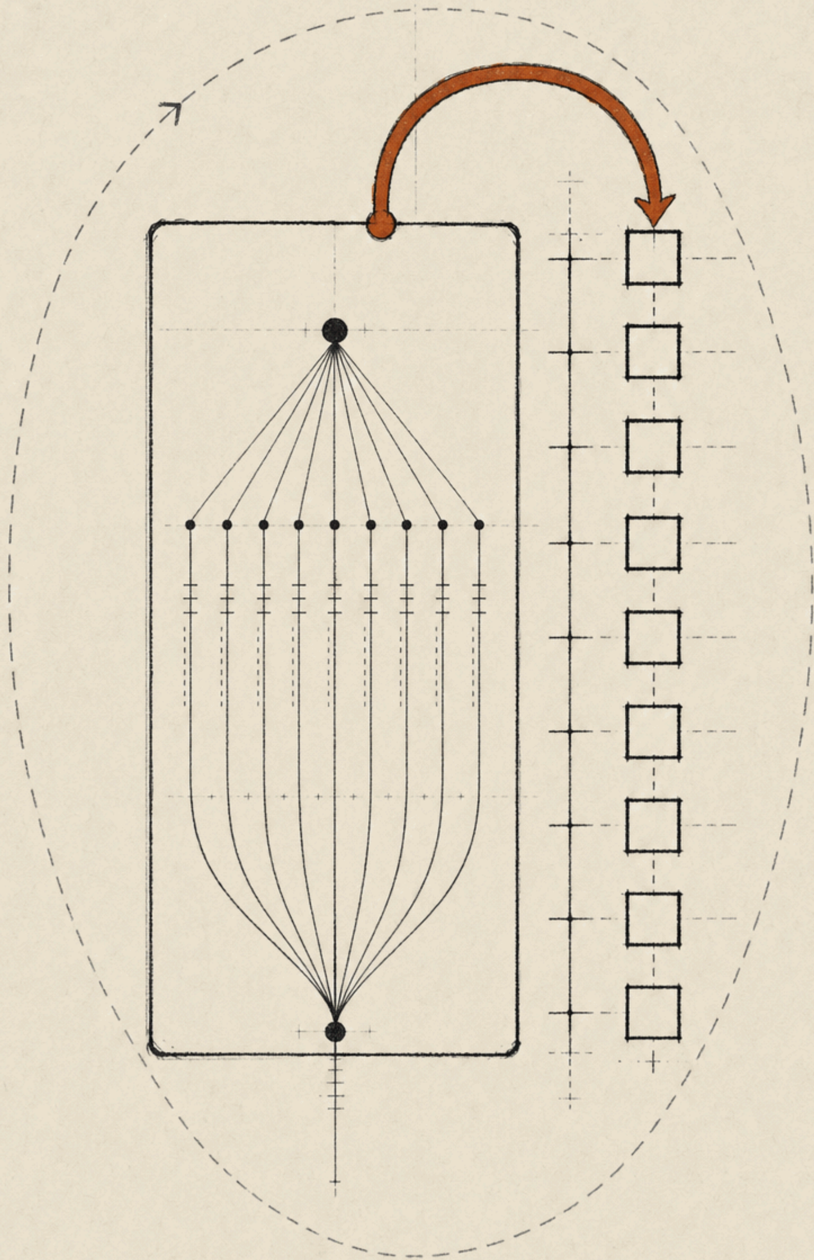
1. Paste full 40-character SHAs (or `git show <sha>:path`) into the brief instead of describing the commit or file in prose.
2. Pin every diagnosis to a hash or a path that cannot drift under the agent; for a range, give both endpoints, not a name.

FROM SPECSTORY HISTORIES

```
May 2026, a /goal opener: “can we
evaluate the commit span between
161c75aa… through e58af436… which
introduced the agent pill… Please
write your findings to @docs/
bugs/”
```

PATTERN 13

The Self-Grading Spec (Goal + Rider)



The Self-Grading Spec (Goal + Rider)

At full strength the brief stops being a chat message and becomes a durable, self-grading spec: a goal plus a rider with an embedded shell-predicate check, so "done" is a passing harness, not a feeling.

A ticket is done when a person says it is. A goal is done when a shell command says it is. You write the exit condition into the artifact before the work exists, then the agent runs that condition on itself and keeps going until the shell agrees.

> *The agent never gets to feel finished; it gets a falsifiable definition of finished.*

That is why such a run can resume cold. On May 10, 2026 a Stoa session opened with no prompt from the human at all, rehydrated the repo state on its own, and ran roughly 35,000 lines across 659 agent turns before a person typed a word. The verification block told it when it was allowed to stop, so a restart could pick up at the next unfinished phase instead of redoing work.

This is a power-move, not the default. Across 1,310 sessions, exactly four goal documents exist. You reach for it when the turn is large, autonomous, and expensive to get wrong: the overnight run, the multi-phase build, the thing you want to walk away from. Pair the grader with one safety valve so a 35,000-line run does not drift into 35,000 lines of confident garbage.

HOW TO RECOGNIZE IT

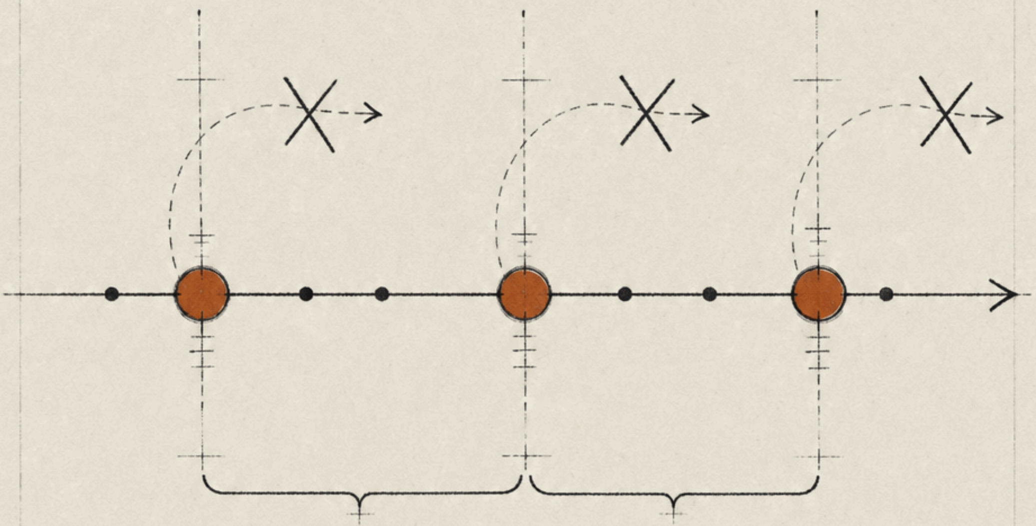
A session opens with no human prompt and runs tens of thousands of lines and hundreds of agent turns before the first human turn lands. In the May 10 run that turn arrived at line 35,283 of a 42,883-line transcript, six human turns total. The agent does not stall waiting for approval, because the spec already encodes when it is allowed to stop.

WHAT TO DO

1. For the rare big autonomous turn, write the exit condition into the artifact as shell predicates the agent runs on itself, not as prose a human has to judge.
2. Add the safety valve verbatim: "If the rider conflicts with reality, stop and surface."
3. Exile the prescriptive detail to a second rider file so it never bloats the executor's context; keep the goal short.

FROM SPECSTORY HISTORIES

```
May 10, 2026 cold–resume goal,
verification section:
“Verification — all must pass
before stopping. REPORT.md exists
at the path above. grep -c "^\#\#
Need: " REPORT.md ≥ 15. Each \#\#
Need: block has ≥ 5 distinct
source citations across ≥ 3
distinct platforms…” Plus the
safety valve: “If the rider
conflicts with reality, stop and
surface.”
```



PATTERN 14

Commit at Phase Boundaries, Never Push

Commit at Phase Boundaries, Never Push

Grant the agent everything up to the deploy edge, including committing at phase boundaries, but keep the push to a branch, the seam where velocity meets review, for yourself.

The flagship run had a second boundary, and it was about hands, not done-ness. Its self-grading spec said when the agent could stop; this is about how far it was allowed to go. It woke cold, rebuilt its own ground truth, and committed across hundreds of turns before a human typed a word, the whole way through. It still had a hard ceiling, and the ceiling was the push.

That is the line worth drawing on purpose. Everything before the push is reversible inside the repo, so you can let the agent own it: the diff, the phase commits, even the spec it graded itself against. The push is the act that puts code on a path to staging, and staging is where other people's work starts depending on yours.

> *A commit is a save point. A push is a promise to the rest of the org.*

Note what the human's first turn actually was. Not a correction to the code. A request to commit the goal file itself to `dev`, which put the spec the agent had been grading against on the same tracked footing as the code it produced. The intervention was about what gets tracked and what gets pushed, not about whether the work was right. The edge is the checkpoint; the keyboard comes back at the edge.

HOW TO RECOGNIZE IT

The agent commits repeatedly across a long, autonomous run and stops dead at the push.

When the human finally intervenes, the first turn is about what gets committed or pushed, not about the code itself, because the code was already trusted up to that line.

WHAT TO DO

1. Tell the agent to commit at phase boundaries and never push, so a long run leaves a trail of save points you can inspect and unwind.
2. Keep the push to a branch as your checkpoint; treat it as the moment you ratify the work onto the path to staging.
3. Put the spec under version control too: have the agent commit the goal file so the contract it graded itself against is tracked alongside the code.

FROM THE FIELD NOTES

“The agent commits at phase boundaries and the agent never pushes. The first human turn in that session, when it finally arrived at line 35,283, was a request to commit the goal file itself to `dev`… The push to a branch, the thing that puts code on a path to staging, is the seam where agentic velocity meets human review. Everything before it the agent can own. The edge, I keep.”

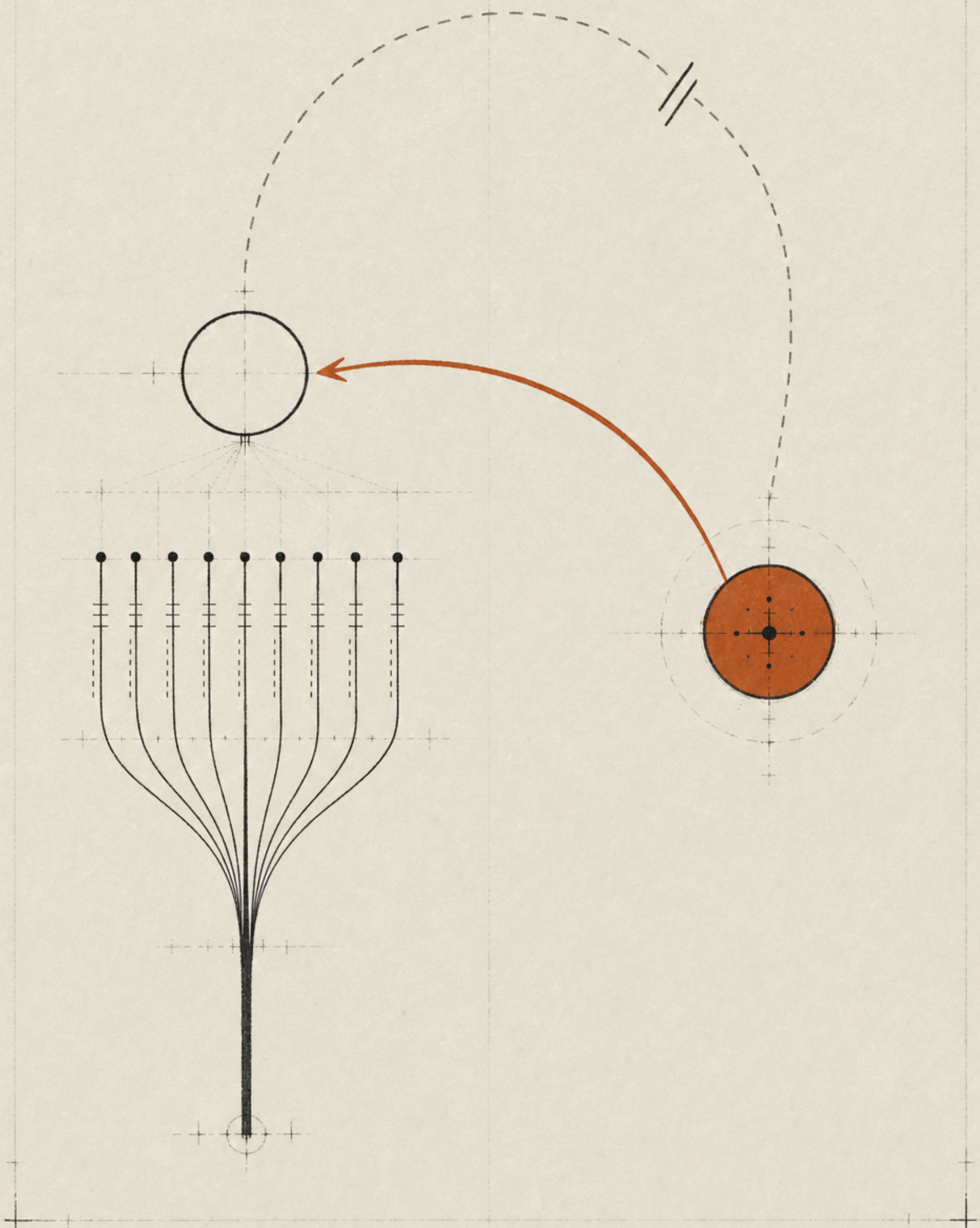
PART IV

Docs Are the API Between Turns

The agent that finishes a task and the agent that picks it up next share nothing but a file. Write the file like it is the only thing that survives.

PATTERN 15

Write for a Reader Who Remembers Nothing



Write for a Reader Who Remembers Nothing

Put a `file:line` citation behind every claim and front-load a source-of-truth table, so the next agent re-opens and re-verifies each finding instead of trusting it blind.

The agent that finishes a task and the agent that picks it up next share nothing but a file. The one that reads it did not sit through the session that wrote it. So every claim in the doc that does not rest on a citation is a claim the next agent either re-derives from scratch or swallows whole, and swallowing whole is how a reader with no memory drifts.

> *A human review says "the plugin system looks risky." The doc is built so a second agent can re-open each cited line and re-verify it without re-deriving.*

The reason is mechanical. The context window is the bottleneck, and telling the agent `git show v0.1.8:path` is cheaper and surer than letting it search the tree and hope. Docs get written to cut exploration. This is also how you pay down the cognitive debt that clean, machine-written code otherwise leaves: the citation and the source-of-truth table are the shared model, written where the next human and the next agent both read it.

HOW TO RECOGNIZE IT

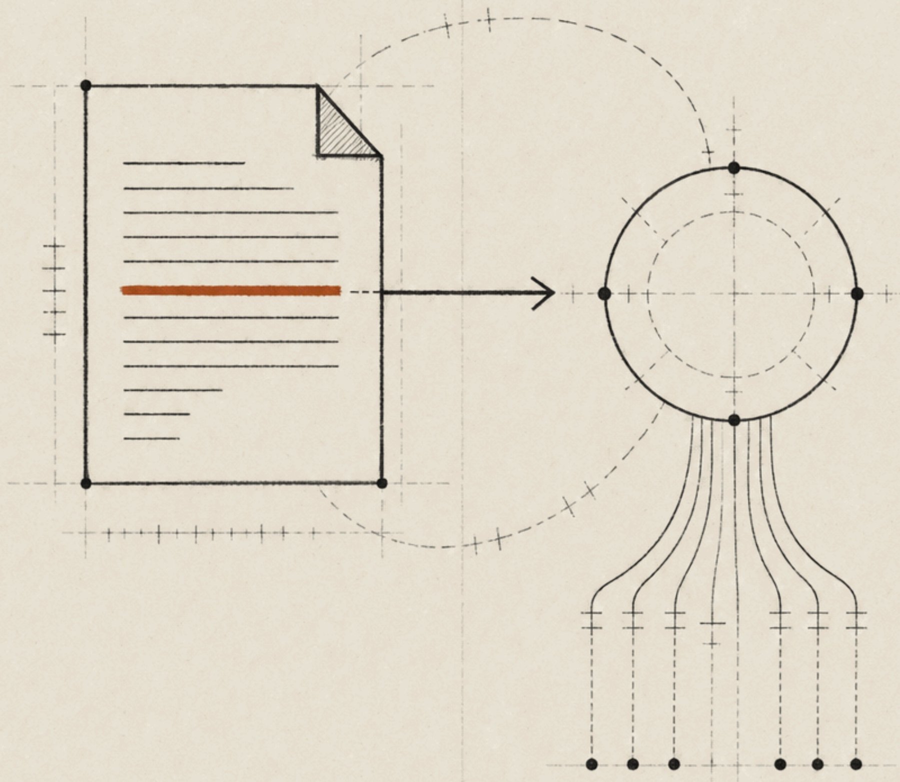
A security audit doc carries 129 distinct `file.ext:line` citations and nineteen explicit `Confidence:` ratings, with a stated standard up front (no exploit traffic; source review and line-level evidence only). Migration docs front-load a "Source of truth" table that hard-codes git refs like `git show v0.1.8:...` so the agent never guesses where ground truth lives.

WHAT TO DO

1. Put a `file:line` citation behind every claim so each finding is independently re-openable by the next turn.
2. Front-load a source-of-truth table that hard-codes the paths and git refs, instead of letting the agent search the tree.
3. Treat the AS-BUILT and design docs as the team's shared model on disk; that is how you pay the cognitive debt down.

FROM SPECSTORY HISTORIES

```
docs/analysis/2026-04-23-stoa-web-critical-high-security-audit.md: 129
file:line citations, 19
Confidence ratings:

Evidence:
• stoa-web/app/api/plugins/execute/route.ts:35 reads caller-supplied source.
• stoa-web/app/api/plugins/execute/route.ts:63 says the source manifest is the permission authority.

Mar 2026 migration "Source of truth" table:
| v0.1.8 source | git show v0.1.8:server/packages/sandbox-agent/src/ |
```



PATTERN 16

The Incident Doc Programs the Next Agent

The Incident Doc Programs the Next Agent

Write the postmortem for the stateless reader who picks it up cold: mark the boundary of what you could not observe, keep the dead ends on the record with the reason they were rejected, and pin every diagnosis to a count, not an adjective.

In traditional engineering the dead ends evaporate. They live in the debugger's head for an afternoon, then they are gone. Here they are the payload. The next agent has no memory of the session and will otherwise walk straight back down the same plausible-but-wrong branches. So the doc persists each negative result alongside the reason it was rejected, and the next reader inherits the elimination instead of re-running it.

> *The postmortem stops being a story about what happened. It becomes a contract for what the next turn is allowed to believe.*

Three moves show up in the careful ones. Mark what you could not observe. Record the false starts as content, not embarrassment. Pin the diagnosis to a number a reader can recompute. A stochastic reader cannot argue with a count it can re-grep, and it cannot be allowed to treat an inference as a measurement. The doc refuses to let it.

This is a learned style, not a house style. The earliest incident doc, from September 2025, raced to a fix and recorded no dead ends. The moves cluster in the April 2026 daemon investigations, in a handful of roughly twenty docs. That it emerged at all is the finding.

HOW TO RECOGNIZE IT

The doc says "strongly implies," not "was." It names the wall it hit instead of papering over it. It splits "Confirmed" from "Not yet fully explained" rather than racing to one root cause. And the diagnosis is tied to a reproducible string and a count, not an adjective like "the log was flooded."

WHAT TO DO

1. Mark the boundary of the unobservable. Write "we cannot confirm X" where you hit a wall, even when a confident sentence would read better.
2. Keep each false start with the reason it was rejected, under a heading that names what the evidence did not settle.
3. Tie the diagnosis to a string an agent can grep and a count it can reproduce, not to an adjective.

FROM SPECSTORY HISTORIES

April 2026 orphan-daemon disk-fill postmortem:

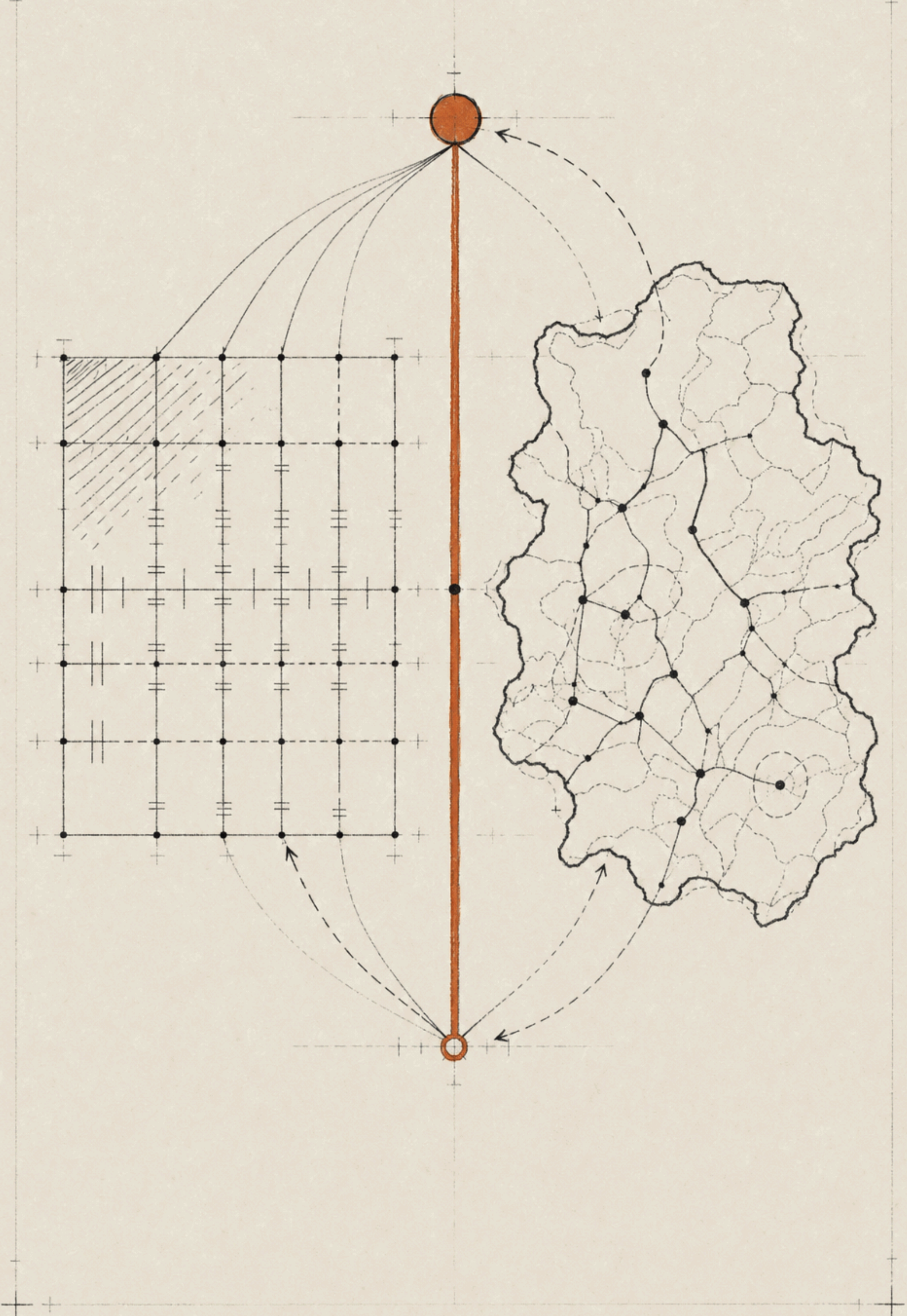
"Note: We could not read the actual contents of the deleted multi-gigabyte log file (macOS SIP blocked dtrace, and there's no /proc/PID/fd on macOS)… we cannot confirm the exact log contents."

A sibling daemon postmortem, split into Confirmed and "Not yet fully explained," then pinned to telemetry:

"=== sql: database is closed === count=29632 This is not incidental noise. It is a primary signal."

PATTERN 17

The AS-BUILT Map Is a Test That Can Go Stale



The AS-BUILT Map Is a Test That Can Go Stale

Keep one AS-BUILT-ARCHITECTURE doc per subsystem that the next agent reads first to inherit your orientation, dated and stamped with the SHA it was last trued against, and re-verify it on a cadence because it rots like any test.

The very first prompt in the corpus was already reaching for this artifact, an AS-BUILT map of the codebase. By 2026 it had hardened into a habit. An agent traces the real code paths end to end, writes down how the system actually works, and that becomes the shared map. The first instruction in a diagnosis task is then to go read it.

> *Read the map first, inspect only as needed.*

That single line turns a focus-file allowlist from a stab in the dark into a believed claim on the agent's end. It now shares your orientation, so "look only at auth and middleware" lands as a fact instead of a guess, and the context window goes to the work instead of re-derivation.

The catch is the one in the pattern name. A map is only true the day you drew it. Twelve of these docs live on disk now, one per subsystem, each dated and each carrying the commit SHA it was last trued against. They rot like any other test, so a recurring agent role re-verifies line counts with `wc -l`, fixes stale paths, and prunes packages that no longer exist. The doc is treated as an assertion that can fail, not a record that sits.

HOW TO RECOGNIZE IT

A diagnosis brief opens with "Read stoa-web/AS-BUILT-ARCHITECTURE.md first" instead of describing the system inline, and a scheduled agent role re-trues the docs against `git log` and `wc -l` rather than waiting for a human to notice the map drifted from the territory.

WHAT TO DO

1. Maintain one AS-BUILT doc per subsystem. Date it and stamp it with the SHA it was last trued against, and tell every diagnosis task to read it before it inspects anything.
2. Schedule a recurring agent role to re-verify line counts and prune dead paths. Treat staleness as a failing test, not a cosmetic chore.

FROM SPECSTORY HISTORIES

April 2026 security-audit opener:

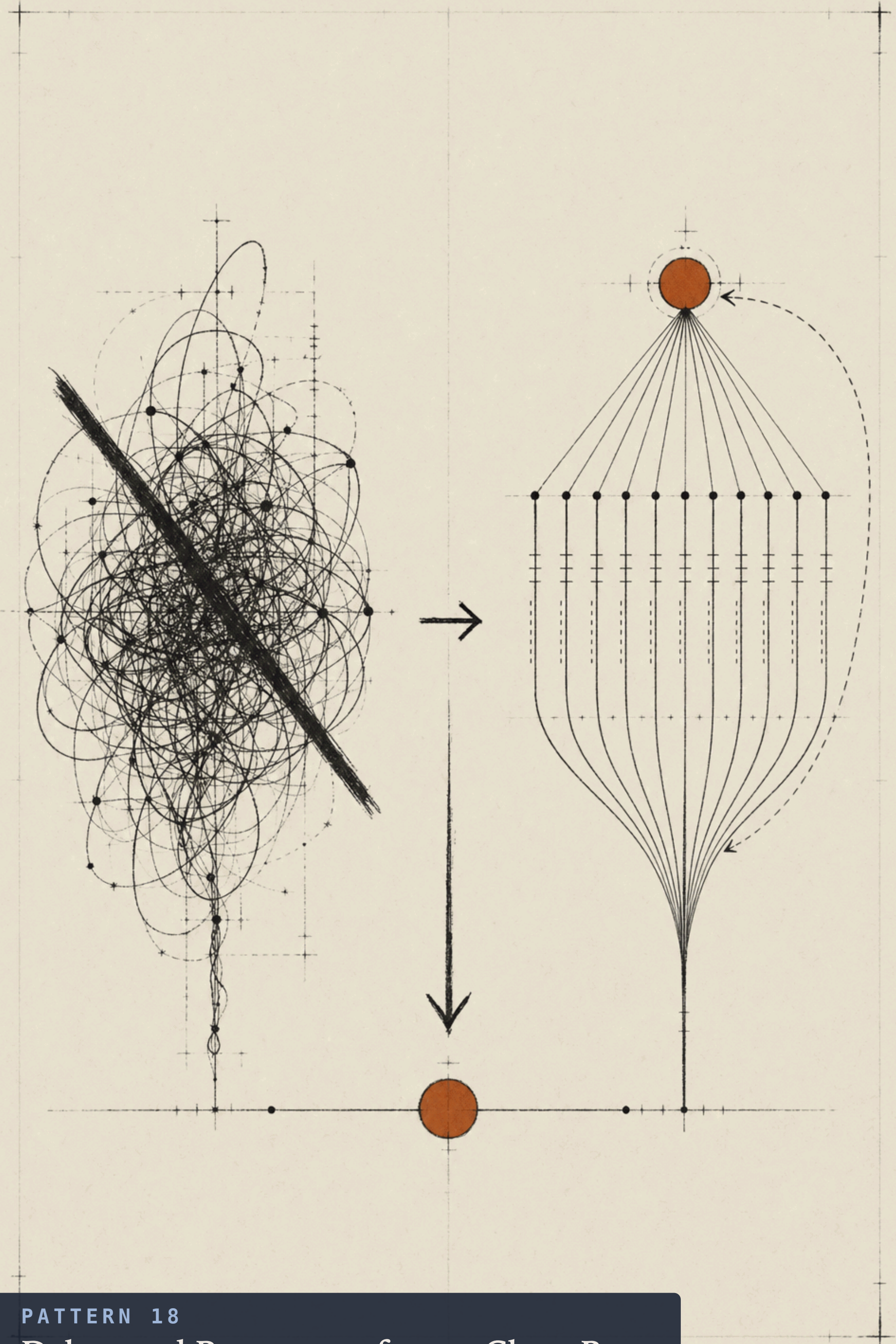
"Read stoa-web/AS-BUILT-ARCHITECTURE.md first, especially auth, API routes, middleware. Inspect … only as needed … Return concrete findings with file:line evidence."

Twelve such docs on disk, one per subsystem, each dated and SHA-stamped, re-verified with `wc -l`.

PART V

Code Is Cheap, Understanding Is Dear

Once regenerating from a clean base takes an afternoon, the cost model inverts. The diff is disposable; the intent you can restate is the asset.



PATTERN 18

Delete and Regenerate from a Clean Base

Delete and Regenerate from a Clean Base

When a branch turns into a pile of incomplete optimizations, find the last clean commit, write down what you learned, and rebuild fresh, keeping the understanding and throwing the diff on the floor.

In January 2026 there was a branch called `oom-phase3-work`: days of commits, all green, written while chasing an out-of-memory bug through Automerge. Working code, the kind a sane person protects. It was deleted on purpose, and that was the rational move.

> *The diff went on the floor. What came with us was the learning.*

What survived was one sentence, "those felt like incomplete optimizations," and the knowledge of which root cause to hit. Not one line of code.

This is the inversion. When the agent writes the code, the code stops being the expensive thing. What costs you instead is understanding the system plus a clean generator to rebuild from, whether that generator is a prompt, a design doc, or a base commit. The moves that look like vandalism to a traditional engineer, abandoning a branch of green commits and regenerating instead of patching, are the cheap ones now. The band-aid you keep is the expensive code.

HOW TO RECOGNIZE IT

The vocabulary gives it away. Across 1,310 transcripts, "throw away," "start over," "from scratch," "rewrite," and "rip it out" recur the way a person talks about a draft, not a crisis. Throwing code away is not a bad day; it is Tuesday. The concrete tell: a days-old all-green branch deleted on purpose and rebuilt from an earlier SHA.

WHAT TO DO

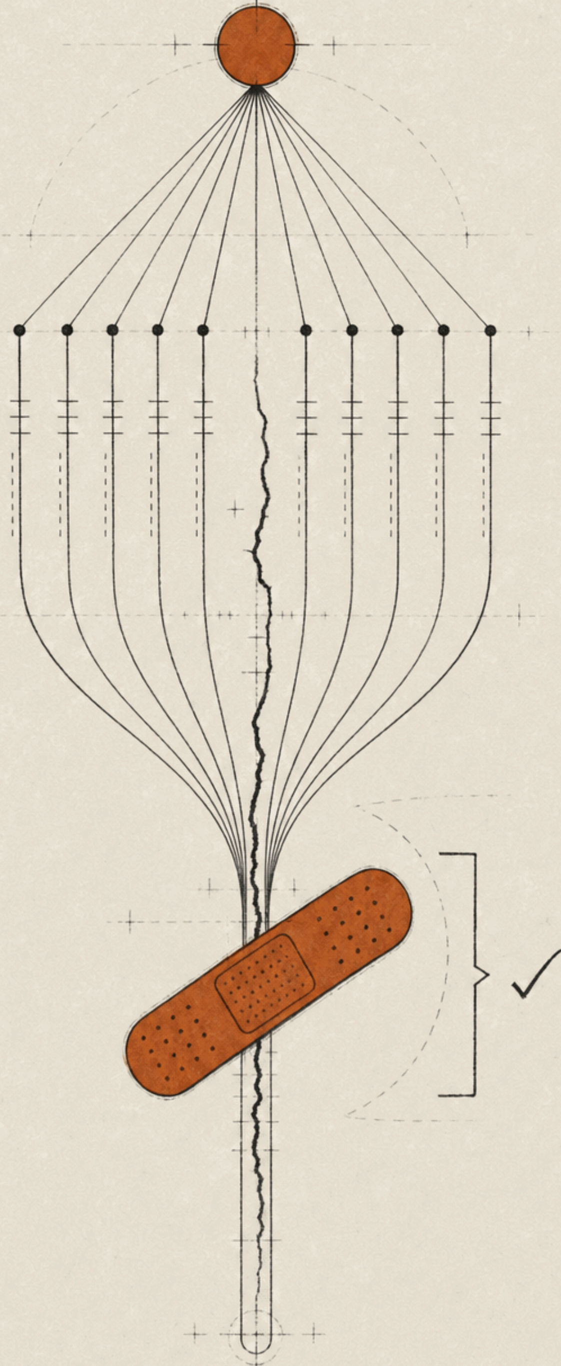
1. When a branch feels like band-aids, locate the last clean commit, record what you learned, and rebuild from there.
2. Treat committed, working code as a provisional sketch. Keep the root-cause doc, not the diff.
3. Ask the agent which commits were workarounds versus real fixes before you decide where to restart.

FROM SPECSTORY HISTORIES

```
Jan 2026, on oom-phase3-work:
"should we actually work from
this point of the branch in git
OR should we work from
1c05c3a61aa0e876e1d5e392968c8669c0cdf4b8
since we learned a lot from all
of our incremental changes and
scripting? Some of those other
changes felt like incomplete
optimizations"

Agent: "Start fresh from 1c05c3a6.
The commits on oom-phase3-work were
band-aids … Workarounds for
symptoms, not the root cause."
```

PATTERN 19

Band-Aid Is a Verdict, Not a Default



Band-Aid Is a Verdict, Not a Default

Once the agent can regenerate the clean fix in an afternoon, the workaround you carry forever becomes the most expensive code there is, so “band-aid” stops being the pragmatic choice and becomes the word for the choice you reject.

In the deadline-driven world this trade ran the other way. The band-aid was the cheap path: ship the workaround, file the tech-debt ticket, promise to fix it after the quarter. The clean fix was the expensive one you never got to. That calculus inverts the instant an agent can regenerate the clean fix cheaply.

> *A workaround you carry forever is the most expensive code there is.*

The workaround is no longer the rational shortcut. It is the line you will have to read, own, and keep alive long after the agent that could have replaced it has been swapped out from under you. The honest read of the field notes is that this work is mostly correction, not new features. On a shared trunk the fix commits outrun the feat commits, and “harden” shows up as a named lifecycle phase, not an apology.

The evidence piles up in one direction. Every time the word “band-aid” appears in the transcripts, the workaround loses, because the real fix costs less than the workaround once the agent does the typing.

HOW TO RECOGNIZE IT

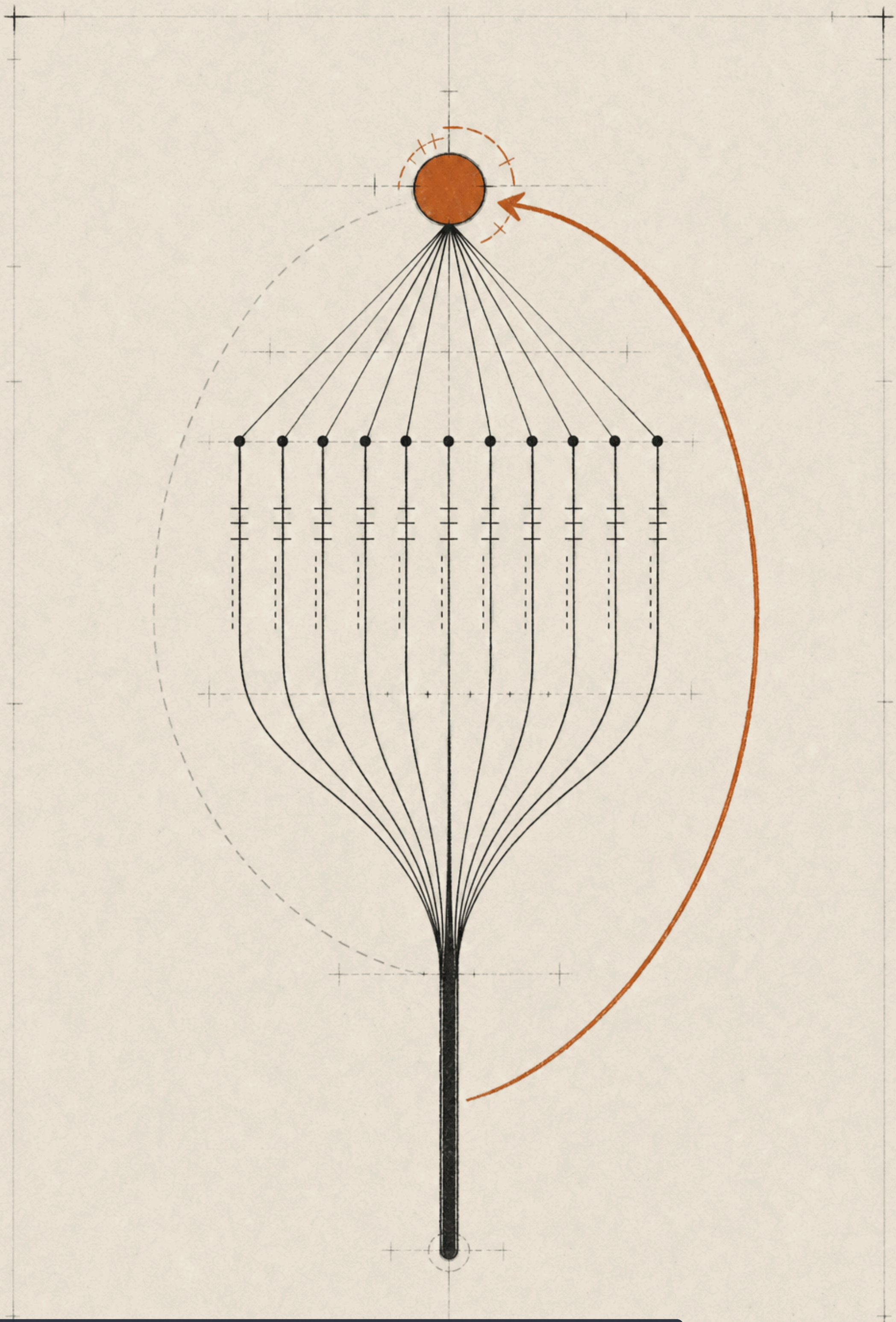
The tell is in the vocabulary: “band-aid” recurs as a verdict that loses, not a category you ship. A dedup proposal gets rejected before a line of it is written, because it would “still accumulate duplicates”; a design doc closes with “Verdict: Quick band-aid, but fragile.” If you catch yourself about to file a tech-debt ticket you both know you will not pick up, you are in this pattern.

WHAT TO DO

1. Treat “band-aid” as a verdict, not a default. When you reach for a workaround, price the clean regenerate first; it is probably an afternoon now.
2. Do not file the tech-debt ticket you both know you will not pick up. Either fix it now or name it as the choice you are rejecting.
3. When you reject a workaround, write the verdict and the reason into the doc, so the next agent inherits the decision instead of re-proposing it.

FROM SPECSTORY HISTORIES

The verdicts run one way. Feb 2026, on duplicate state: “Dedup at the processing level would be a band-aid, you would still accumulate duplicates in the state array, wasting memory and slowing renders.” Dec 2025 design doc: “Verdict: Quick band-aid, but fragile.” Across the transcripts the word shows up as a thing you reject, not a thing you ship.



PATTERN 20

Fix the Generator, Defend the Shape

Fix the Generator, Defend the Shape

For what the output means, strengthen the generator (the prompt) because meaning generalizes where a lookup table cannot; for what its shape is, defend with a parser, because models are unreliable narrators of JSON.

There is a tempting strong form of this move: fixing the prompt is the reflex. It is wrong, so state it honestly. The split is the pattern.

Take the clean case. An agent had written a `COMPOUND_NORMALIZATIONS` map, deterministic code that merged "AutoMerge" and "auto merge." It worked. I killed it anyway, because the map covers the cases you thought of and the prompt covers the long tail you did not.

> *For a semantic problem, the prompt is the generator and the map was one engineer's guesses transcribed into code.*

You trade working deterministic code for a probabilistic instruction precisely when the instruction generalizes and the code cannot. Grep today: `COMPOUND_NORMALIZATIONS` returns zero hits.

Now the counter, because a skeptic finds it in thirty seconds. For structural problems, where what is wrong is the shape of the JSON and not its meaning, the reflex runs the other way and it ships in production. When the model returned objects where a Go struct wanted `[]string`, the agent laid out options, called "Fix the Prompt" the simplest, then recommended flexible parsing now and schema enforcement later. They took the parsing. `extractJSON` still carries the comment that the model "often responds with markdown-wrapped JSON." Half the evidence points each way.

HOW TO RECOGNIZE IT

A working deterministic normalization map gets deleted (zero hits today) because "our map won't generalize," while a defensive markdown-wrapped-JSON parser ships in the same codebase. You are deciding whether the failure is about meaning or about shape.

WHAT TO DO

1. For semantic problems (do two strings mean the same thing?), strengthen the prompt and delete the lookup table.
2. For structural problems (is the JSON the right shape?), keep a defensive parser; do not try to prompt your way to reliable JSON.
3. Keep a backstop even when you strengthen the prompt: the December deletion left a Jaccard-similarity dedup as a fallback.

FROM SPECSTORY HISTORIES

Dec 2025, deleting a deterministic map: "i don't believe that approach of merging and normalizing the decisions is correct because our map won't generalize" then "lets strengthen the prompt to force the LLM to match against existing decisions."

Jan 2026 counter, in production: `extractJSON` comment "Claude often responds with markdown-wrapped JSON, so we attempt both direct and code-block parsing."

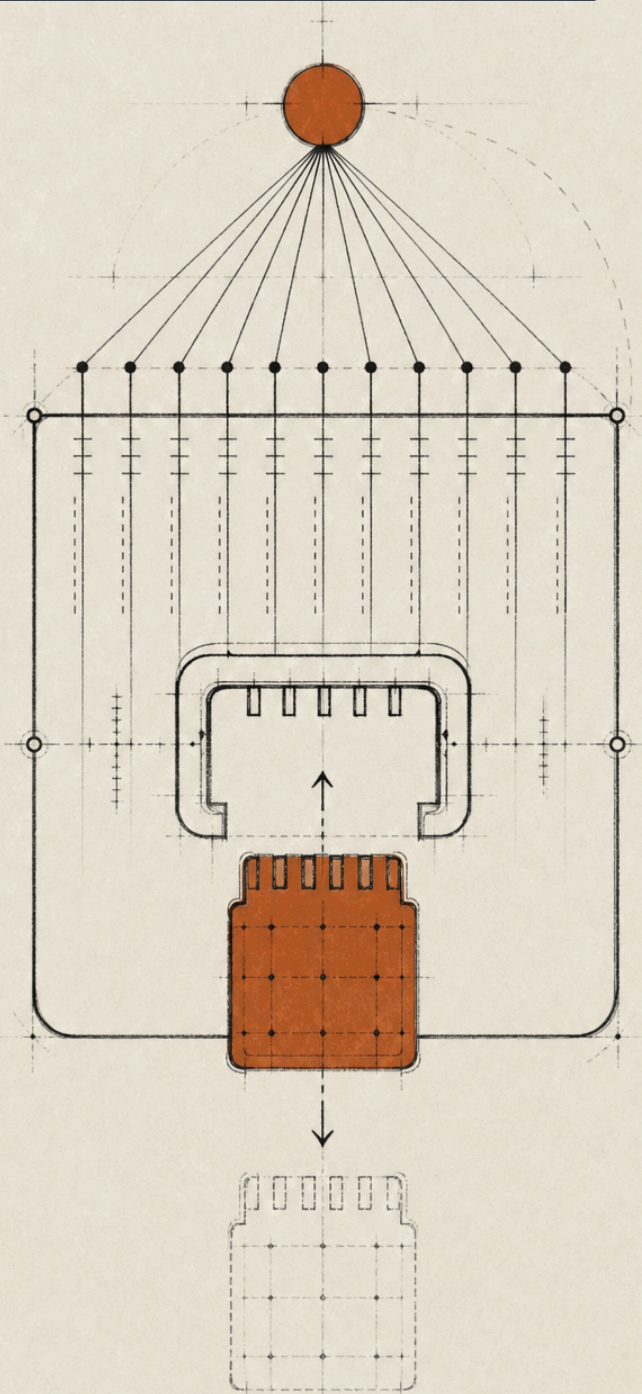
PART VI

You Run an Org, Not a Pair

Agentic work scales to a roster of models routed across sessions and a small team feeding one trunk. The model under the work upgrades itself on a schedule you do not set.

PATTERN 21

The Model Is a Swapped Dependency



The Model Is a Swapped Dependency

The agent you start a project with is not the agent you finish with, so record which model co-authored each commit and treat the upgrade as a dated dependency bump you take under load.

The pairing metaphor breaks the moment the worker swaps under you mid-project. You did not make it better and you did not train it; it arrived better, and everyone on the planet got the same upgrade the same day.

> *The teammate changed four times and the commits never paused.*

So your edge cannot live in the model, because the model is shared and replaceable. It falls back to the two things the upgrade does not touch: what you meant, and whether you can prove it.

Treat the frontier release the way you treat a dependency bump, not a new hire. You take it under running load, mid-branch, without ceremony. The `Co-Authored-By` trailer is what turns the swap into something durable: your name on the commit, the model's name dated underneath it. That record is the only reason you can later say which version wrote which code, and route the next task to the model that fits it.

HOW TO RECOGNIZE IT

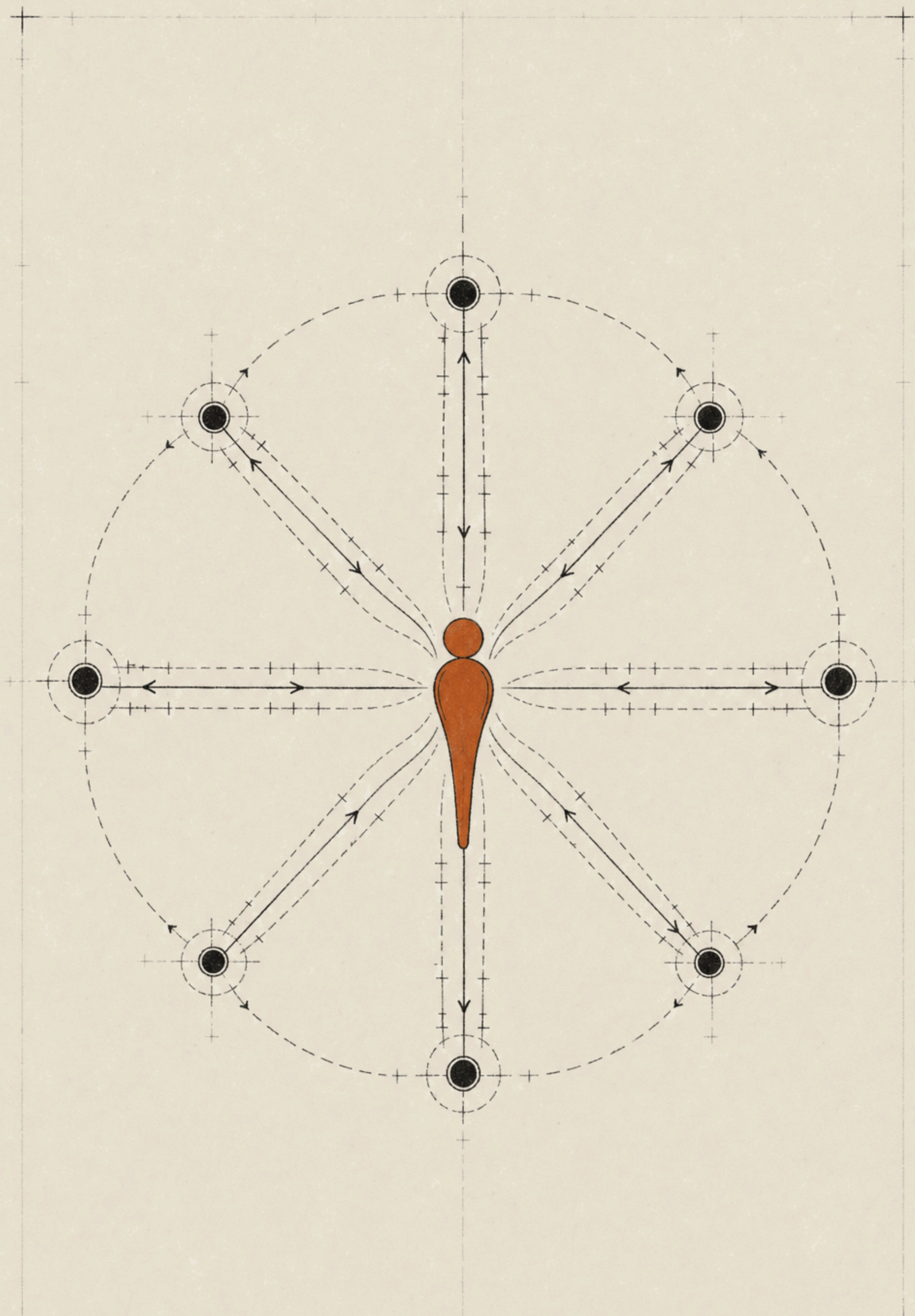
The co-author census across one shared history walks Opus 4.5 (510) to 4.6 (504, plus 890 on the 1M build) to 4.7 (366) to 4.8 (just appearing, 3) while the commit stream never pauses for a frontier release. If you cannot run that census, the swap already happened off the record.

WHAT TO DO

1. Put a `Co-Authored-By` trailer naming the model on every commit, so the version walk is dated in the history instead of lost.
2. Bump the model under the running work; do not halt the stream for a release.
3. Route per task: keep the census so you know which dialect did what, and send the next job to the model that fits it.

FROM SPECSTORY HISTORIES

```
Across all branches the trailer
walks the versions:
Opus 4.5 = 510, Opus 4.6 = 504,
Opus 4.6 (1M) = 890,
Opus 4.7 (1M) = 366, Opus 4.8
(1M) = 3, plus 385 from
an early stretch where the
trailer said only “Claude.”

“My name sits on the commit and
the model’s name sits
under it… The human owns it; the
version that helped
is recorded, dated, and
replaceable.”
```



PATTERN 22

The Human Is the Message Bus

The Human Is the Message Bus

Codex and Claude share no memory, so when you want one to build on what another found, or to red-team it, you carry the state yourself and paste it across.

There is no channel between the models. They cannot see each other's context, plans, or scrollback. So you are the channel.

> *It is like forwarding a thread to a colleague who missed the meeting, except the colleague has no inbox.*

You copy one agent's working state out of its window and paste it into the other's prompt, glyphs and prompts and all.

The primitive is "here is the other agentic session." That one opener turns two stateless workers into a relay. One model chases a bug, hits a wall, and the scrollback becomes the brief for the second model that picks it up cold.

The same move is how you red-team a design across vendors. One model proposes a plan; a second audits it against the real library ecosystem; you pipe the audit back to the first and tell it to incorporate the findings. Neither model knows the other exists. The cross-check only happens because you ran the wire between them.

HOW TO RECOGNIZE IT

A prompt to one model contains a pasted transcript fragment of another (the ⏺ glyphs, the > prompts, the whole scrollback) under an opener like "here is the other agentic session." You are typing fast and routing, not writing essays; the typos stay in.

WHAT TO DO

1. Paste one agent's working state into the other's prompt to continue or cross-check the work; carry the state yourself, because no channel exists.
2. Red-team across vendors: have a second model audit the first's plan against ground truth, then pipe the audit back to the first to revise.

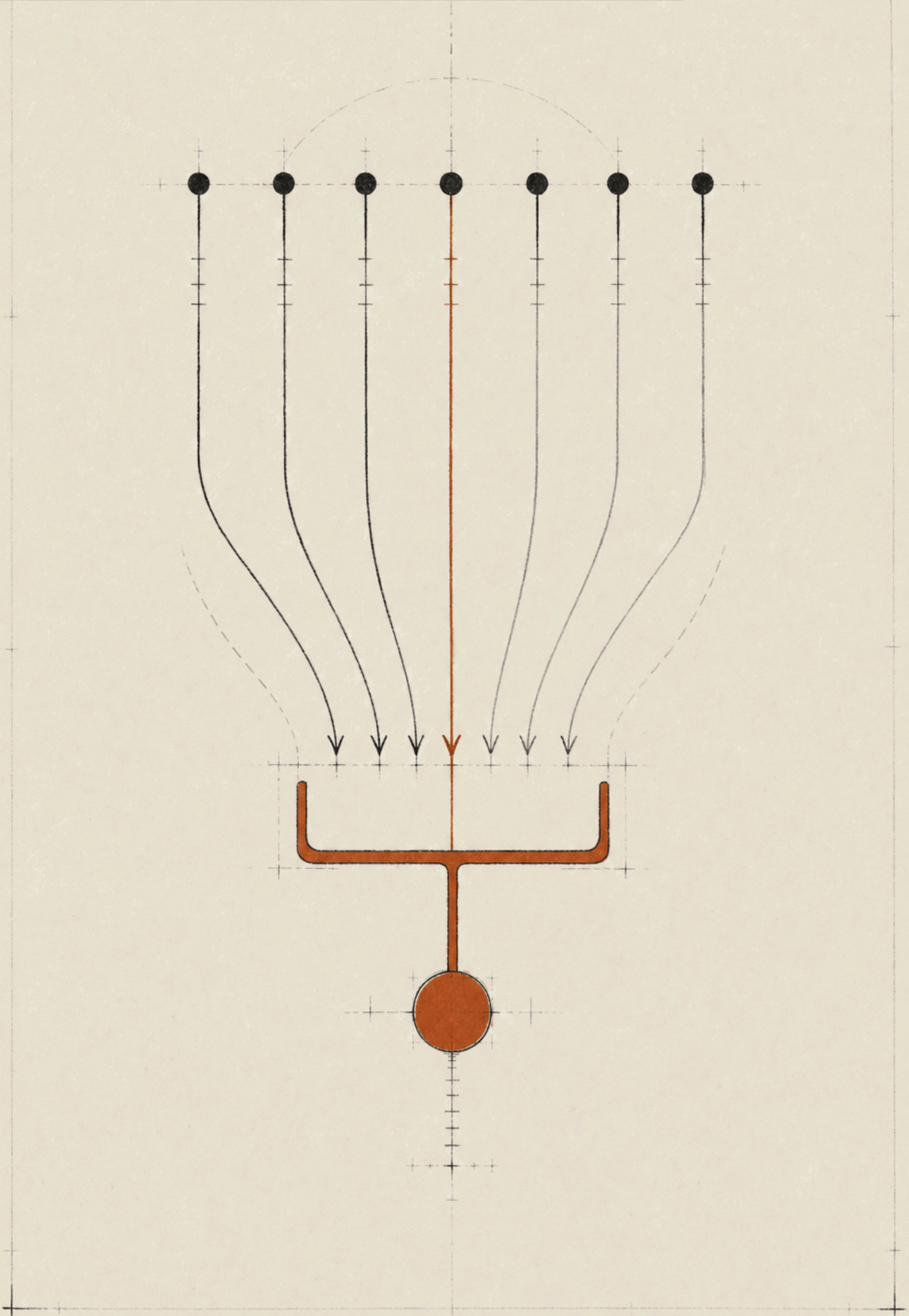
FROM SPECSTORY HISTORIES

`Nov 2025, chasing a LiveKit video-render bug. Claude's scrollback pasted into Codex: "OK i've been trying to debug this video rendering issue and its a bit of a pain … here is the other agentic session."`

`Jan 2026, cross-vendor critique fed back to Claude mid-build of the Explorer TUI: "Berore we move forward with Sprint 5, I want you to look at this feedback from another agent that evaluated our implementation versus what is available in charm.land, I want you to comprehensively update our plan to incorporate leveraging these packages." The "Berore" typo is left intact.`

PATTERN 23

Agents Self-Assign; You Pick the Lane



Agents Self-Assign; You Pick the Lane

A mature agent reports completion, reads the backlog, and offers to take the next unclaimed item, so your job collapses to four words: pick the next lane.

This is the part that feels like running an org and never felt like pairing. The agent does not wait to be told what is left. It finishes, looks at the task list, and tells you which items are still unclaimed and which one it can take. You stop chasing status and start ratifying allocation.

> *I did not ask what was left. My job in that exchange was four words: pick the next lane.*

You read the offer, you say which lane, the work continues.

Do not picture a swarm in one room. The exchange is quieter than that: the agent reports in, names what is unclaimed, and waits for you to say which lane. Your turn is the allocation, not the typing. You are dispatching a roster that runs across your sessions and your teammates', and the worker that just finished is already reaching for its next assignment.

HOW TO RECOGNIZE IT

A subagent finishes a task you never asked it to close out, then lists the pending unassigned items it could pick up next and asks which one you want. You are being handed a backlog by something that already read the backlog. If you find yourself asking "what's left," you are doing the agent's job instead of yours.

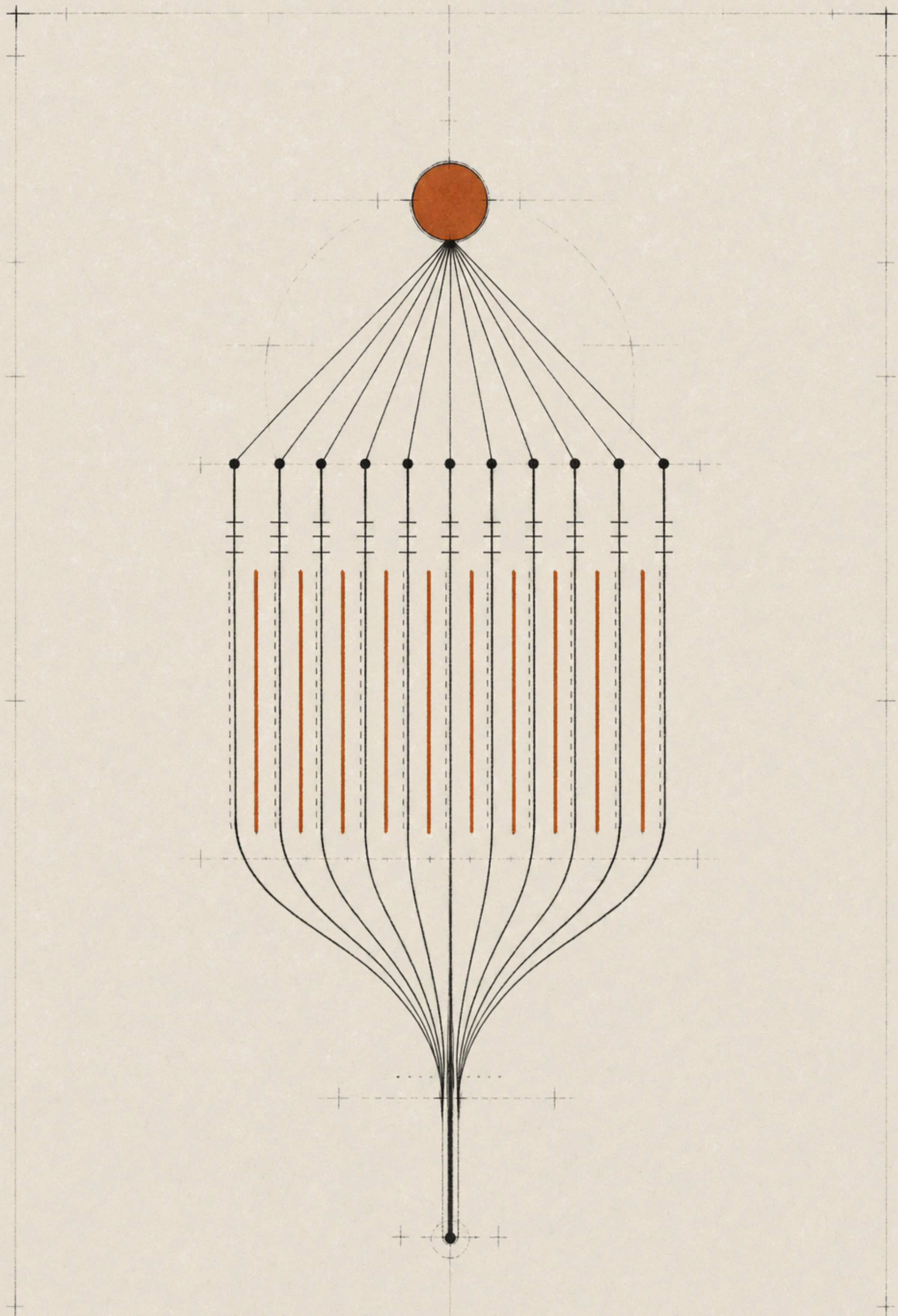
WHAT TO DO

1. Let agents report completion and surface the backlog themselves; reserve your turn for allocation, not status-chasing.
2. Hold the fleet as distributed across humans and across sessions, not as one operator commanding a swarm; route, don't crowd one room.

FROM SPECSTORY HISTORIES

Feb 2026, the `bridge-architect` subagent, unprompted:

"Task 2 is already completed. I'm available for another task. Pending unassigned tasks I could pick up: 5 (native auth with Keychain), 6 (native menu bar/shortcuts), 7 (native LiveKit video panel), or 8 (native file sidebar). … let me know which you'd like me to take on."



PATTERN 24

Fan Out Along Non-Overlapping Seams

Fan Out Along Non-Overlapping Seams

Within-session parallel spawn is a rare flourish, not the rhythm, and it only pays when you cut the bug along genuinely separate seams so the lanes never step on each other.

This is not one prompt run three times to sample the best answer. It is three prompts, each scoped to a different layer of the same bug, run in one wall-clock interval to get a full cross-section instead of three overlapping guesses. The backend completion path is one lane. The durable store and the frontend that reads it are a second. Observability is a third. Cut the seam first, then spawn.

> *The lanes have to be genuinely separate, or the agents step on each other and you net nothing.*

Do not mistake this for the engine of velocity. Within-session fan-out shows up in roughly 18 of 857 sessions, an explicit TeamCreate in about 41. Peak velocity came from a second agent lane running across sessions, six humans each steering two or three agents into one trunk, not from one operator commanding a swarm inside a single turn. The swarm is the flourish you reach for when a bug spans clean architectural boundaries. The durable fleet runs across sessions, not inside them.

HOW TO RECOGNIZE IT

Three Codex sessions launched in thirteen seconds, timestamped :28, :35, and :41, each forbidden to edit and scoped to a disjoint layer of one bug. If you cannot name the seam each lane owns, you are sampling, not fanning out.

WHAT TO DO

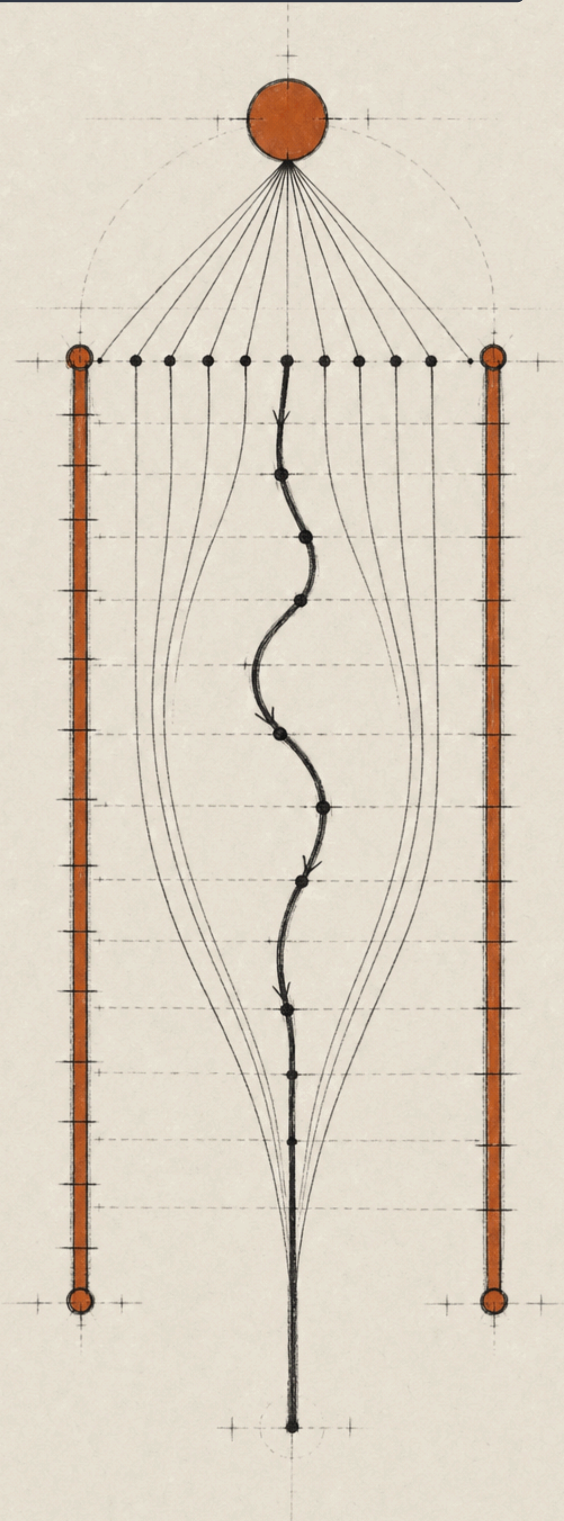
1. Cut the bug along non-overlapping seams first (backend, then durable-plus-UI, then observability), then spawn one lane per seam.
2. Scope each lane to its own files and forbid edits, so the parallel turns return diagnoses you can reconcile, not collisions you have to untangle.
3. Treat within-session fan-out as the exception; lean on the across-session two-lane fleet for sustained velocity.

FROM SPECSTORY HISTORIES

May 27, 2026, three Codex sessions in 13s. Lane 1: “Please inspect only backend prompt-delivery/completion path. … Answer with likely root cause(s), the smallest robust code change, and tests to add. Do not edit files.” Lane 2: the durable store and frontend (`sqlite.ts`, `useSpaceAgent.ts`, `agent-turn-state.ts`), “where a repair should live.” Lane 3: observability only, “so future Vercel traces show `after()` start/resolve/fallback/SQLite write without logging sensitive prompt content.”

PATTERN 25

Make the Agent Show Its Rails



Make the Agent Show Its Rails

Delegate destructive git rarely and per operation: hand over the intent in one sentence, then require the agent to name the scope check, the backup branch, and `--force-with-lease` before it acts.

The default for destructive git is advisor mode, not delegation. Across the corpus the agent proposes a force-push far more often than it runs one. "If you want, I can push the rebased branch with `--force-with-lease` next," and then the session ends, because the human runs it. The phrase that recurs is "but don't actually do it." The low revert rate on `main` is partly what that caution buys.

The exception is granted in plain language, once, for one move you have already decided is right and only want done safely.

> *I supplied intent. The agent supplied every defensive mechanic, unprompted.*

Given only "lets do the backup branch and force-push with lease," a mature agent scopes the blast radius, re-fetches origin so the lease compares against live state, sequences a backup branch before the overwrite, uses `--force-with-lease` over a bare `--force`, and confirms the uncommitted work survived.

Those rails exist because the delegation is rare. The posture was trained across dozens of advisor-mode sessions where the agent walked through these exact steps before the human ran them by hand. When you finally hand over the keys, it drives the way it has been narrating all along. No rails, no keys.

HOW TO RECOGNIZE IT

Given a one-line authorization for a history overwrite, the agent does not just run it. It narrates the scope check first, names the backup branch (`dev-pre-…-backup`), states the plan, and reports back that nothing was lost. If it reaches for the irreversible op without showing those mechanics, take the keyboard back.

WHAT TO DO

1. Keep destructive git in advisor mode by default: "pretend I'm going to merge this branch into main, but don't actually do it."
2. When you do delegate, require the scope check, the backup branch, and `--force-with-lease` named before the act.
3. Train the posture in advisor mode first; only hand the keys to an agent that has narrated the safe sequence to you many times.

FROM SPECSTORY HISTORIES

Apr 2026, authorizing a history overwrite of `origin/dev`: "lets do the backup branch and force-push with lease". The agent, unprompted: "I'm refreshing origin/dev first so —force-with-lease uses current remote state, then I'll push a backup branch and update origin/dev to b1d98068 without touching your remaining local edits." Then: "I also created the remote backup branch `dev-pre-b1d98068-backup`, which preserves the previous `origin/dev` tip at `cc0f4b47`. Your local uncommitted sandbox changes are still present and were not included in the push."

Colophon

This guide was distilled from a corpus of 1,310 captured agent sessions and 4,670 commits on the `main` branch of Stoa, the platform SpecStory builds for teams to write software live with coding agents, every change tied to the conversation behind it. The field notes were written first, from the transcripts and the git log. Then they were synthesized into the seven shifts of the companion essay, then distilled into the twenty-five patterns and six parts you hold. Then they were adversarially verified against the same log and transcripts. That verify pass demoted three overclaims before they reached print.

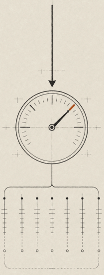
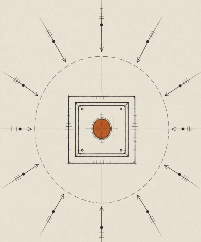
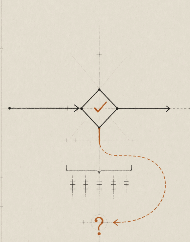
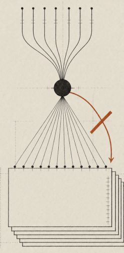
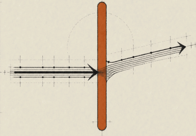
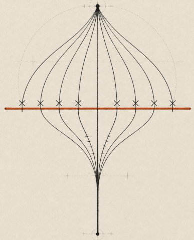
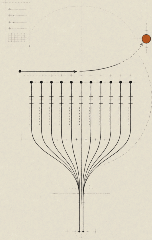
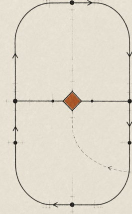
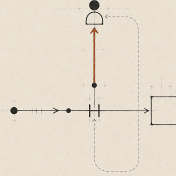
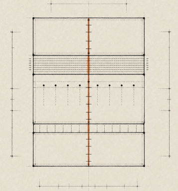
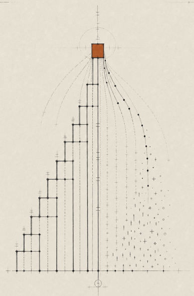
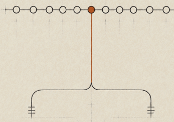
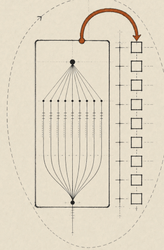
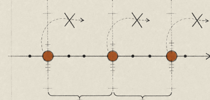
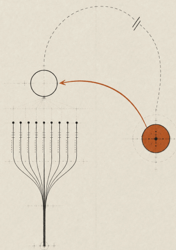
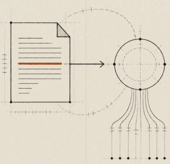
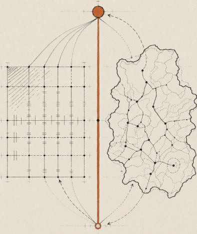
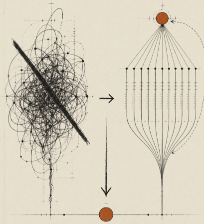
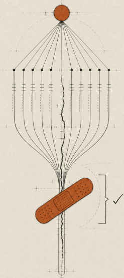
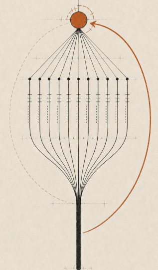
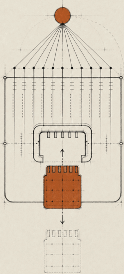
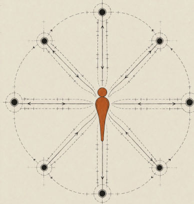
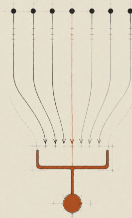
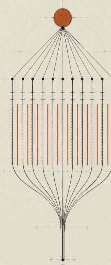
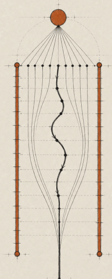
Every claim is pinned to a checkable artifact. The model that co-authored each commit is recorded in a `Co-Authored-By` trailer, so the version walk from Opus 4.5 through 4.8 is dated in the history itself. Names of teammates were removed and one incident was cut by policy. No secrets, no exact cost or memory figures, no auth identifiers appear anywhere in these pages.

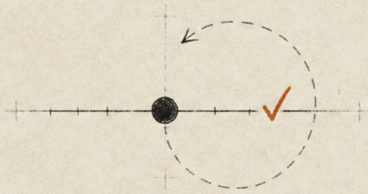
CAPTURE YOUR OWN HISTORIES

None of this book exists without the capture. Raw agent sessions are ephemeral; the chat scrolls past and the reasoning behind a commit is gone. The 1,310 sessions behind these pages survived because they were recorded as they happened, as markdown you can grep, cite, and hand to the next agent.

That capture is SpecStory. It saves your AI coding sessions (Cursor, Claude Code, and the rest) into a `.specstory/history` folder beside your code, in plain markdown that travels with the repo. Once your history is on disk you can `grep -c` your own interrupts, trace which prompt produced which diff, or hand a past session to a fresh agent. The patterns in this book came out of doing exactly that to ours.

Capture yours: `github.com/specstoryai/getspecstory`



25 Patterns in Agentic Engineering

A field guide to building software by steering agents.

Greg Ceccarelli · SpecStory Press · First edition, 2026

Read or download the current edition at
`specstory.com/books/25-patterns-in-agentic-engineering-book-2026.pdf`

The companion essay: *Goal Engineering*
Distilled from building Stoa: withstoa.com · specstory.com