contact@namiq.io · namiq.io

NamiQ

NAMIQ PLAYBOOK

The AI SDLC Playbook

Two ceilings, six practices, and the lifecycle around the code: from a team that ran it on itself

Full edition. How to keep quality while AI writes more of your code, with the failures that taught us each rule, a twelve-intervention pilot, and the templates to start on Monday.

PREPARED BY	Hung Le, NamiQ	DATE	2 November 2026
FACTS AS OF	28 October 2026 (scorecards), 30 October 2026 (ticket status), 31 October 2026 (Claude's logs)	EDITION	1.7
FOR	Engineering leads, owners, founders	INSIDE	60 findings, 1 pilot, 9 templates, 6 figures, 2 scripts

NamiQ © 2026 NamiQ. All rights reserved. This playbook may be shared unaltered, with attribution. Free to read and share. Web edition: namiq.io/resources/ai-sdlc-playbook

START HERE

The playbook on a page

AI makes code faster to write. Two ceilings stop that becoming faster delivery: a person can only review so much, and a confidently wrong change travels further as the product grows. Six practices answer them. None needs our software.

Six practices, each with the first move and the first number.

#	Practice	Defends against	First move	First measure	Ch.
1	**A person can finish the job** Add one check that a real person can complete the job on the running system, and let it block a release.	Confidently wrong	Write the sentence a customer would say when it works	Journeys defined, and journeys completing	4
2	**Gates tighten as the codebase grows** Make at least one threshold a function of size.	Confidently wrong; drifting	List what your coverage number excludes	Coverage of changed lines	5
3	**Idle is reported, not hidden** Put "idle, zero done" on the board in plain text when it is true.	Invisible	Add "Waiting" and "Blocked" states and move items where they belong	Active share of elapsed time, per stage	6
4	**Blocked is the loudest thing on screen** Blocked work shows the day it blocks, at the top, before anything green.	Invisible	Sort blocked items to the top, with their age	Blocked now, and the median age	7
5	**Metrics are recomputed, not remembered** Derive every number from the work record, and show the ones that do not flatter you.	Drifting	Recompute the number you would be most embarrassed to find wrong	Places a metric is typed (target: one)	8
6	**Multiply reviewers, not the reviewer** Build a ladder of checks that block on failure, then narrow human judgement where it is needed.	The first ceiling	Make one guard fail on purpose and see whether it catches it	Share of changes blocked by each rung	9

Where to start, for what you have today

You have ten minutes	Chapter 3.5: find your quiet failure, then read the practice it points to
You can change one thing on Monday	Chapter 7 (sort blocked to the top) and Chapter 6 (add Waiting and Blocked)
You want to know where you stand	Chapter 16, the self-assessment
You are planning the next quarter	Chapter 17, the 30, 60 and 90 day plan
You own the process end to end	Part III, Chapters 10 to 13
Model providers see client material	Chapter 11, which model may see what
You want to know what to believe	Chapters 14 and 15, and Appendix A with its dates

Edition 1.7, 2 November 2026. Scorecard figures are as at 28 October 2026; ticket status as at 30 October 2026; Claude's working time is read from its logs, 5 September to 31 October 2026; the board capture is 2 August 2026. Every figure carries its date.

Contents

How to read this playbook 4

I. THE PROBLEM

1. The two ceilings 6
2. The lifecycle around the code 8
3. Three ways it fails quietly 10

II. SIX PRACTICES

4. Practice one: a check that a person can finish the job 12
5. Practice two: gates that tighten as the codebase grows 14
6. Practice three: idle is a status to report, not hide 16
7. Practice four: blocked should be the loudest thing on screen 18
8. Practice five: recompute metrics, do not remember them 19
9. Practice six: multiply reviewers, not the reviewer 21

III. THE LIFECYCLE AROUND THE CODE

10. From a requirement to a ticket that is ready 25
11. Which model may see what 29
12. Building and reviewing inside a fence 30
13. Closing the loop: retrospectives, decisions and what the process learns 32

IV. WHAT BROKE

14. One real requirement, end to end: twelve interventions 34
15. What was wrong before, and what the pattern is 36

V. ADOPTING IT

16. Where are you? A self-assessment 39
17. A 30, 60 and 90 day plan, and who decides 40
18. The metrics, defined 43

APPENDICES

Appendix A. The numbers, and their dates 45
Appendix B. Templates 46
Appendix C. Glossary 48
Appendix D. Sources and what this playbook does not claim 49
Appendix E. The gates and the merge time, as scripts 50
Appendix F. Reverse-engineering our own build, and two exploit chains worked through 51
Appendix G. Secure development: the build checks what it depends on 55

How to read this playbook

AI writes more of your code every month, and you are still the one who has to read all of it. This is the full edition of the playbook: two ceilings that explain why the obvious fix does not work, six practices that do, the lifecycle around the code that most teams leave untouched, and an honest account of what broke when we renovated our own.

It is long because the short version (the free web page) tells you what to do and this one tells you how, with the failure that taught us each rule. Page 2 is the whole playbook on one page, with where to start for what you have today. Every chapter opens with a box saying what you will be able to do and the one idea to keep, and ends with something you can do in ten minutes. Read the six practices in Part II first if you are short of time. They stand alone. Part III is for whoever owns the process end to end. Part IV is the record of what went wrong, and it is the part we would read first.

What is in it

Part	Question it answers	Chapters
I. The problem	Why does faster code generation not make delivery faster?	1 to 3
II. Six practices	What do you change on the board, the gates and the metrics?	4 to 9
III. The lifecycle around the code	What has to exist between "a requirement arrived" and "a ticket is ready"?	10 to 13
IV. What broke	What went wrong when we ran it on ourselves?	14 and 15
V. Adopting it	Where are you, what do you do first, who decides?	16 to 18
Appendices	The numbers and their dates, templates, glossary, sources, the gate and merge-time scripts, a reverse-engineering trial with two exploit chains, and a secure-development check on the build's dependencies	A to G

Where the numbers come from, and how old they are

Three kinds of number appear, and they are never mixed without a date.

The published capture, as at 2 August 2026. Taken from our own delivery board for a live product past 165,000 lines, and shown with its screenshot at namiq.io/results. Coverage 80%, defects 2.64 per 1,000 lines, 43% of work later redone, 17 days to reach that scope. It is a dated capture, not a live feed, and the board it came from has moved since. We keep the capture because the board is also the evidence, and a figure and its screenshot should not drift apart.

The position on 28 October 2026. Read straight from the same boards' scorecards that day, for the factory itself and for that product. They are here because the August capture is three months old, and some of them are worse. The product's defect density is 3.41 per 1,000 lines, up from 2.64. Its line count fell after a refactor, which raises the density by shrinking the divisor, and we have not separated that effect from any real change. We would rather print the position, with what we got wrong, than the August figures alone.

Claude's working time, from its own logs, 5 September to 31 October 2026. Measured on 31 October 2026 from every Claude session log on the machine, for all our work, not only the work that is a ticket. The logs go back no further, so nothing earlier can be measured this way. Appendix A.2 has the figures. They are an order of magnitude with their ranges, not a benchmark: they count Claude's working time only, not time a person spends away from the keyboard.

> **A rule we keep for the whole playbook.** A number without a date is a claim about today. Every figure here carries its date and its scope, and where a number does not flatter us it stays in.

What is new in edition 1.5

> **Edition 1.7, 2 November 2026.** Appendix G: secure development, one check on a project's declared dependencies and one on the host a release lands on, with the first run over our own seven projects, its numbers and its mistakes; chapter 12 gains a short section that points to it. **Edition 1.6, 1 November 2026.** Appendix F: a real REA trial run against our own compiled build (every figure is the tool's own output), and two exploit chains worked through from CISA's and the vendors' own published advisories, each with its source. ** The two board screenshots are removed from Appendix A.2; the figures in the tables stand on their own, and the share-of-time-working rows are removed from them. Edition 1.4, issued earlier the same day: Chapter 10.2 now has a diagram of the artifact chain, a table of what the AI does, what a person holds and what refuses a broken link at each link, and why it matters; the measure of working time has a per-ticket figure; and the line spacing is wider. Edition 1.3, issued earlier the same day: Appendix A.2 gained what Claude's own logs show of Claude's working time. Edition 1.2, issued earlier the same day, added a one-page start (page 2), a box at the head of every chapter, five figures, and the four appendices, A to D, in the front matter. The practices and every other number and date are unchanged.

What this is not

It is not a tool you must buy. Nothing in the six practices needs our software, and we would rather you did them yourself than hired anyone. It is not a claim that the process is finished: Chapter 14 lists twelve manual interventions that one real requirement needed on the way through ours. And it is not a benchmark. The product the numbers come from is our own, where we set the scope and chose the stack. That is the friendliest case there is, and you should discount accordingly.

Conventions

- Dates are written in full. "As at" always names the day a figure was read.
- IDs like ODSF-258 are tickets in our own work record. They are quoted so the claim can be checked against a number, not because you need to look them up.
- Where a practice is ours and not an industry standard, it says so. Where a figure comes from someone else, it is linked in Appendix C.
- "We" is NamiQ. The judgements marked as ours are ours.

I. THE PROBLEM

1. The two ceilings

YOU WILL	name the two ceilings that stop faster code generation becoming faster delivery, and tell which one you are under.
KEY IDEA	a person can only review so much, and a confidently wrong change travels further as the product grows.
TEN MINUTES	take the last 30 merged pull requests and divide lines changed by review minutes.

Coding was never the only process in building software. It was the one that got faster first. Requirements, design, security review, testing and release are most of the real work, and none of them sped up because generation did. Point AI at "write the feature" without touching the rest of the lifecycle and you have handed a slower pipeline more raw material than it can absorb.

Two ceilings follow. We wrote them down in July 2026, before we had seen anyone else measure them.

CEILING 1

Your reviewer is your throughput

AI can multiply how much code gets written by ten. It cannot multiply by ten how much a person can safely review.

Symptom: review gets shallower. The pull request read line by line in March is skimmed in September.

Answer: Chapter 9

CEILING 2

It degrades exactly where you are going

The blast radius of a confidently wrong change grows with the codebase. At 5,000 lines it fails where you can see it; at 165,000 it fails three modules away, a week later.

Symptom: defects are found by customers, or more than a sprint after the change that caused them.

Answer: Chapters 4 to 8

1.1 Your reviewer is your throughput

AI can multiply how much code gets written by ten. It cannot multiply by ten how much a human can safely review. If every line still passes through one person's eyes, that person's reading speed is your real throughput, however fast the model is.

The symptom is not a queue of unreviewed code. Teams are better than that. The symptom is review getting shallower. The pull request that was read line by line in March is skimmed in September, because there are three times as many of them and the same afternoon to read them in. Nobody decides to review less carefully. The volume decides.

How to tell if you are under this ceiling. Four questions, answered from the record, not from memory:

1. How many pull requests per reviewer per week now, against a year ago?
2. What is the median time a pull request waits for its first review, and is it rising?
3. What share of merged changes had a reviewer comment that changed the code?
4. What share of production incidents trace to a change that was approved in under five minutes?

If the first number is up, the second is flat or falling, and the third is falling, review is thinning. The fourth tells you what it costs.

1.2 It degrades exactly where you are going

AI is excellent on day one: a blank file and an afternoon get you a week of senior work. Real products cross into six figures of lines, where parts depend on each other in ways nobody wrote down. The model does not get worse. The blast radius of confidently wrong grows with the codebase, and nothing tells you unless something is built to.

A confidently wrong change in a 5,000-line product usually fails where you can see it. In a 165,000-line product it fails three modules away, in a path nobody looked at, a week later. The same mistake costs more because it travels further before anyone notices.

How to tell if you are under this ceiling. Look at where your defects are found and when. If a rising share is found by customers and not by your checks, or found more than a sprint after the change that introduced them, the checks are not growing with the codebase.

1.3 What the industry has measured since

Those two paragraphs were written in July 2026. Since then others have measured the same thing, with the caveats the authors themselves give.

Finding	Source	How to read it
1.7× more issues in pull requests containing AI-assisted code than in human-written code	arXiv preprint 2603.28592, March 2026	An association between two kinds of pull request. It is not a measurement of a rise over time, and it does not say the code was worse. It may reflect harder tasks being given to AI.
30 to 41% rise in technical debt within six months of widespread AI tool adoption	Innovative Group, 2026	Directional. A vendor analysis, not an audited study.
45% of AI-generated code introduced a security vulnerability	Innovative Group, 2026	The same analysis. The definition of "introduced" matters and is not ours.
4× maintenance cost by year two for teams that do not actively manage the debt	Innovative Group, 2026	The same analysis. A projection, not a measurement of a finished period.

Three of the four come from one analysis, and the first is a separate preprint. They are published figures, directional rather than audited. We list them because the direction agrees with what we saw, not because the magnitudes are settled. Every one is linked in Appendix C so you can weigh it yourself.

1.4 What we think the gap is

The gap between code generated and code safely reviewed is not idle capacity. It is code that ships having been read less carefully than the code before it. We think that is what the 1.7× figure is measuring, though the study reports the association and not the cause.

If that is right, the fix is not "review harder". A fixed amount of human attention cannot cover a rising volume by trying harder. The fix is to change what the human attention is spent on, which is the argument of Chapter 9

and, in a way, of the whole playbook.

1.5 Ten minutes: put a number on your ceiling

Pull the last 30 merged pull requests. For each, write down lines changed and minutes between "review requested" and "approved". Divide. If the median lines-per-minute is above what a person can read with care (a few hundred lines per hour is a common working figure, and yours may differ), you have your first measurement of the first ceiling.

2. The lifecycle around the code

YOU WILL	see the whole path from a requirement to a live, understood change as eleven stages, each with an output and a decision holder.
KEY IDEA	the fence at each stage is a check the model cannot satisfy by producing more text.
TEN MINUTES	write your own lifecycle and mark every stage whose output is not a file or a record.

"AI SDLC" is used loosely. We mean something specific: the whole path from a requirement arriving to a change being live and understood, with AI doing a defined share at each step and a person holding each decision that needs judgement. The code is one step of eleven.

2.1 The eleven stages

This is the target lifecycle we designed in October 2026 and then ran our own work through. Each stage names its output, and each output names its parent. Chapters 10 to 13 take them in turn.

Stage	Name	What comes out	Who holds the decision
S0	Intake	A requirement, with an id	Product owner
S1	Understand what exists	An evidence pack and a current-state architecture	Architect
S2	Explore	A brief	Product owner
S3	Requirements	A specification, drafted then accepted	Product owner approves
S4	Solutioning	Design, and decision records marked Proposed	Technical lead accepts
S5	Backlog	Epics and stories, each with acceptance criteria	Product owner
S6	Readiness gate	Each story marked ready, or bounced with the reason	Fails closed
S7	Build	A branch, code and tests	The assigned role
S8	Review and test	Review findings and gate reports	Technical lead and QA
S9	Close and adapt	A retrospective and re-sliced stories	Technical lead and product owner
S10	Knowledge	What was learned, linked to what it came from	Knowledge curator

The eleven stages. Each names its output, each output names its parent.

S0 Ch. 10 **Intake** Product owner	**S1** Ch. 10 **Understand what exists** Architect	**S2** Ch. 10 **Explore** Product owner	**S3** Ch. 10 **Requirements** Product owner approves
S4 Ch. 10 **Solutioning** Technical lead accepts	**S5** Ch. 10 **Backlog** Product owner	**S6** Ch. 10 **Readiness gate** Fails closed	**S7** Ch. 12 **Build** The assigned role
S8 Ch. 12 **Review and test** Technical lead and QA	**S9** Ch. 13 **Close and adapt** Technical lead and product owner	**S10** Ch. 13 **Knowledge** Knowledge curator	

Chapter 11 (which model may see what) applies at every stage.

Three things about the shape matter more than the detail.

Every link in the chain names its parent by id, so a story can be followed up to the requirement that justified it and a requirement down to the tests that prove it. The readiness gate refuses a story whose chain is broken.

There are three tracks, so a one-line fix does not get a requirements document. Quick for a bug or a small change in one component. Standard for a new feature in an existing product. Brownfield for importing a system, replacing a legacy feature or any cross-component initiative. Choosing the track is part of the analysis output, and the product owner can overrule it. Ceremony sized to the change is what keeps people from routing around the process.

Human gates are by design, and they are few. The product owner accepts a specification. The technical lead accepts a design and reviews a build. Everywhere else a person is not in the way, and where a person is in the way it is because the decision cannot be written as a rule.

2.2 Where AI changes each stage

Not every stage gains the same. This table is our reading, from running it, of where AI helps, where it needs a fence and where it should not be left alone.

Stage	What AI does well	What it needs fencing	What stays human
Intake	Classify, deduplicate, ask clarifying questions	Inventing requirements it was not given	Whether the requirement is worth doing
Understand	Read a large codebase and summarise structure	Narrative over a thin graph: confident prose with no evidence behind it	Judging whether the picture is true
Requirements	Draft, find gaps, check citations	Citing a source that does not say that	Scope and acceptance
Design	Propose options and trade-offs	Choosing the fashionable option	The decision, and the record of why
Backlog	Slice into stories	Slices that cannot be tested	Priority
Build	Write the code and tests	Touching files outside its remit	Nothing, if the gates hold

Review	Multi-lens first pass	Agreeing with the author	The decision to merge
Close	Summarise, draft the retrospective	Declaring done for work that did not land	What the retrospective changes

A pattern runs down the middle column. The fence is almost always **a check that the model cannot satisfy by producing more text**: a citation that must resolve, a path that must exist, a test that must run, a boundary that is enforced before and after. That is the idea Chapters 4 to 9 turn into practices.

2.3 Ten minutes: draw your own

Write your actual lifecycle as a list of stages, with the output of each and who holds the decision. Mark every stage where the output is not a file or a record. Those are the places the chain is held in someone's head.

3. Three ways it fails quietly

YOU WILL	name the three ways AI-assisted delivery fails quietly, and which practice answers each.
KEY IDEA	confidently wrong, drifting and invisible. You do not notice because nothing is built to tell you.
TEN MINUTES	pick the last defect you found late and say which of the three it was.

The two ceilings explain why more generation does not mean more delivery. Three failure modes explain why you do not notice. Each practice in Part II is the answer to one of them.

3.1 Confidently wrong

A model produces fluent, plausible output whether or not it is right, and fluent wrong output passes a skim. In a codebase this looks like a test that asserts what the code does and not what it should do, a citation to a requirement that does not say that, or a change that is correct in the function and wrong in the caller. In our own work record this showed up as a requirements citation that did not resolve, a fix applied to the two places a test happened to find and not the five others that shared the pattern (DF-52), and a retrospective that stated a gate wrote to a log when it did not (DF-58, found an hour later by checking the claim).

The defence is a check the model cannot pass by producing more text. Practices 4, 5 and 9.

3.2 Drifting

Anything maintained by hand moves away from the truth at a rate slow enough to keep trusting and fast enough to be wrong when it matters. Two examples from our own site: one defect-density figure printed as 2.64 on a page and 2.1 on another, one scroll apart, because one was typed and one was computed. And a coverage gate that measured only one directory, so that new code in other directories was tested but never counted (DF-05) and coverage could fall in exactly the code being added without any gate noticing.

The defence is to derive numbers from the work record and to compare the thing you measure with the thing you changed. Practices 5 and 8.

3.3 Invisible

Idle and blocked work are the two most expensive states in a pipeline, and they are the states people have least reason to raise. Raising a blocker means admitting it. Idle time looks like the absence of an event. A process that does not show them does not have less of them; it has the same amount plus a reporting layer that hides where it went. We found a ticket marked DONE for work that never landed (DF-17) and a build cut short by a turn limit that was committed as if complete and reported success (DF-54). Neither was visible until someone looked.

The defence is to make the bad states the loudest things on the board. Practices 6 and 7.

3.4 How the six practices map

Practice	Defends against	Chapter
A check that a person can finish the job	Confidently wrong	4
Gates that tighten as the codebase grows	Confidently wrong; drifting	5
Idle reported, not hidden	Invisible	6
Blocked as the loudest thing on screen	Invisible	7
Metrics recomputed, not remembered	Drifting	8
Multiply reviewers, not the reviewer	The first ceiling	9

3.5 Ten minutes: find your quiet failure

Pick the last production defect you found late. Ask which of the three it was: wrong with confidence, a measure that had drifted from the truth, or a state nobody could see. The answer tells you which chapter to read first.

4. Practice one: a check that a person can finish the job

YOU WILL	build one check that a real person can finish the job on the running system, and let it block a release.
KEY IDEA	every other check can be satisfied by writing more code; a completed journey cannot.
TEN MINUTES	write the one sentence a customer would say when your product worked.

The practice. Add one check that asks whether a real person can complete the job on the running system, and let it block a release.

If you do nothing else in this playbook, do this.

4.1 What happened to us

Our test suite passed. Coverage was at 80%, the figure on our published board as at 2 August 2026. Every gate reported success. And two screens were failing to load, with exactly one of nine customer journeys completable end to end.

This is not a criticism of testing. It is the definition of it. A test encodes an expectation, and an expectation is something a person had. The failures that hurt are the ones nobody expected, which is precisely the set no test covers. More tests push the boundary outward, and the boundary never closes.

4.2 Why this one check is different

Every other check in a pipeline can, in the limit, be satisfied by writing more code to satisfy it. A unit test can be made to pass by changing what it asserts. A linter can be quieted. A coverage figure can be raised with tests that execute lines without asserting anything. When the author is a model that optimises for the check, this is not hypothetical.

A completed journey is different. "A new customer can sign up, pay, and see the thing they bought" is true or false of the running system. It does not matter how the code got there. This is what makes it the signal least vulnerable to the way generation fails.

4.3 How to build your first one

1. **List the journeys that pay the bills.** Not every path. The five to ten without which the product has failed: sign up, log in, the one action customers came for, pay, get the result, get support. Write each as a sentence a customer would say.
2. **Rank them and start with one.** The path where a failure costs most. Doing one properly beats sketching ten.
3. **Run it against the running system, not a mock.** A deployed environment, real services, a real browser or client. A journey run against stubs proves the stubs.

4. **Define done as an outcome, not a page.** "The order appears in the customer's order list with the right total" and not "the checkout page returned 200". A page that loads and a job that finishes are different things, and the gap between them is where two broken screens hid.
5. **Make it block.** A check that warns is a check that is overruled the day a release is late. Wire it into the release step so that red stops the release, and make the exception a recorded decision (Chapter 17), not a skipped step.
6. **Keep it small and fast enough to run every release.** If it takes an hour, it will be skipped. A journey that finishes in a few minutes can run on every merge to the release branch.

4.4 What it will tell you the first time

Expect it to fail. When we first counted, one of nine journeys was completable. That figure is the point: it was the first honest measurement we had of whether the product worked for a person, and no other figure on the board said so.

The first run also tells you things you did not ask:

- **Journeys nobody owned.** A path that spans three teams and belongs to none of them.
- **Setup you had forgotten.** A test account, a seeded record, an email inbox, a payment sandbox. The effort to keep these alive is part of the cost of the check, and it is worth paying.
- **Flakiness you cannot yet separate from failure.** Retry once, record that you did, and count retries as a metric. A journey that passes only on retry is telling you something.

4.5 Anti-patterns

Pattern	Why it fails
The journey runs against a mock of the payment step	It proves the mock
Done is "no error on screen"	A silent failure is no error
It runs nightly and a human reads the result	Reading it is the step that gets skipped
One giant journey that covers everything	One failure tells you nothing; split by outcome
The author of the change also writes the journey it must pass	The check inherits the author's blind spot; have the journey written from the requirement

4.6 What to measure

- **Journeys defined, and journeys completing.** The headline ratio. Report both.
- **Time since each journey last passed.** A journey that has not run is not a check.
- **Releases blocked by a journey, and releases overridden.** Overrides are the number to watch.
- **Defects found by a journey that no other check found.** This is the proof the check earns its place.

4.7 Ten minutes: write your first journey sentence

Write the one sentence a customer would use to say your product worked ("I signed up, imported my data and saw my first report"). If you cannot say whether that sentence is true today, that is the finding.

5. Practice two: gates that tighten as the codebase grows

YOU WILL	make at least one threshold a function of the codebase's size, and measure the right coverage.
KEY IDEA	a fixed quality bar is a falling bar. It fails quietly, which is the worst way to fail.
TEN MINUTES	list your directories and write which ones the coverage tool does not count.

The practice. Make at least one threshold a function of size.

5.1 Why a fixed bar is a falling bar

What was good enough at 5,000 lines is not good enough at 165,000. The number of ways a system can be broken grows faster than the system does, because the combinations of parts grow faster than the parts. A fixed quality bar applied to a growing product is therefore a falling one, and it fails quietly, which is the worst way to fail. No one decided to lower the bar. The product grew and the bar stayed put.

5.2 What to make a function of size

Four thresholds are worth tying to the size of the change or the size of the system:

Threshold	Fixed version	Size-linked version
Test coverage	"80% overall"	A ratchet: the figure may not fall below its last accepted value, and a change that adds code must carry its own coverage
Review depth	"one reviewer"	More scrutiny as a change touches more files or more sensitive paths
Blast radius	"any change may touch any file"	A change may touch a limited area without extra approval
Checks per release	"the usual suite"	The set that runs grows with the components affected

The ratchet is the cheapest and the most useful. It turns "coverage should not fall" from a hope into a gate.

5.3 A ratchet that measures the right thing

A ratchet only works if the number it ratchets measures what you changed. Ours did not, for a while, and the way it failed is instructive.

Our second-line gate measured coverage on one directory only. The renovation described in Part III added most of its new code to other directories. Tests existed and passed, but those lines were never counted, so coverage could fall in exactly the code being added without any gate noticing (DF-05). Later, a gate run raised the uncommitted baseline from 78 to 79 and the next run warned against its own write (DF-53). Later still, a baseline shared by every working copy meant that one branch's improvement raised the bar for all the others, and parallel branches blocked each other with false warnings (DF-55).

Three lessons, each cheap once you know them:

1. **Measure the code you are changing.** A ratchet on a number that excludes the new code ratchets nothing.
2. **Keep the baseline in the repository, committed, and change it only in a reviewed change.** A baseline that a gate writes to is a gate that compares itself with itself.
3. **Scope the baseline to the branch.** A shared mutable baseline in a parallel workflow is a race.

5.4 A worked threshold

Here is a threshold written as a rule, not a preference. It is an example shape, not our configuration.

```
coverage_floor   = last accepted coverage on the integration branch, and never under 60% overall
new_code_floor   = 80% of changed lines covered, measured on the diff
test_scripts     = a change that touches code must touch a test, and the tests run as a script, not as
a promise
security_check   = OCO's gate 1 (code health and secrets) and gate 4 (whole system) pass on every
change
merge_time       = reported, with the share of changes that carried tests
review_depth     = 1 reviewer if the diff is under 200 lines and touches one area
                   2 reviewers if it crosses areas or touches a restricted path
whole_system     = full rebuild and dependency scan on every integration, not only on release
```

Each line is a command. Appendix E has them as a script (`gate_check.sh`) and the measurement as another (`merge_time.py`):

```
$ pytest --cov=src --cov-report=xml --cov-fail-under=60     # the 60% floor on the whole system
$ diff-cover coverage.xml --fail-under=80                   # the 80% floor on the changed lines
$ make gate-1 && make gate-4                                # OCO's security gates
$ python3 merge_time.py . main 14                           # tests carried, and merge time
```

Two properties matter. Every line is checkable by a command. And none of them is a number someone has to remember to raise; the floor moves because the thing it follows moves. The 60% is a floor for the whole system and the 80% a floor for new code, so the old code cannot hide a weak change and a strong change cannot hide old code that nobody tests.

5.5 The objection, and the answer

The objection is that people want a stable definition of done. They do, and it is a reasonable thing to want. The answer is that the definition of done can be stable as a *rule* ("the floor never falls; new code carries its own tests") while the *number* it produces moves. What you are protecting is the rule. A team that insists on a fixed number is choosing predictability over honesty, and as the product grows it will find out which it needed.

5.6 What to measure

- **Coverage of changed lines,** not only overall coverage.
- **Gate failures per week and why.** A ratchet that is never red is either perfect or measuring nothing.
- **The share of code changes that carry tests, and the merge time,** from `merge_time.py` (Appendix E). A change that arrives with no test is the one the coverage number cannot see yet.
- **Time to run the gates.** A gate that takes five minutes is tolerated; fifteen is routed around. We recorded a doubled gate time from a duplicated scope (DF-59) and ticketed it for exactly that reason.

Ours, read on 31 October 2026 for the last 30 days, from git history only: of the changes that touched code, 95% of the factory's carried a test (103 of 108), 97% of OCO's (33 of 34) and 81% of the website's (42 of 52). The merge time, counted from the oldest commit that came in with a change to the merge, had a median of minutes in all three (the 90th percentile was 0.4 hours for the factory, 5.0 for OCO and 0.1 for the website). That figure says the work is committed and merged in one sitting, and nothing about how long a ticket waited before someone picked it up: git does not record that, and the ticket system only records created and resolved. It also does not say the tests were good, only that a change did not arrive without any.

5.7 Ten minutes: find out what your coverage number excludes

List the directories in your repository. Next to each, write whether the coverage tool counts it. The ones it does not count are where the next defect lives.

6. Practice three: idle is a status to report, not hide

YOU WILL	define active, waiting, blocked and done, and report idle in plain text when it is true.
KEY IDEA	processes that hide idle time have the same amount plus a reporting layer that obscures where it went.
TEN MINUTES	add "Waiting" and "Blocked" to your board and count what is in each.

The practice. Put "idle, zero done" on the board in plain text when it is true.

6.1 Every process has idle time

Every delivery process has idle time. Processes that hide it do not have less of it. They have the same amount plus a reporting layer that obscures where it went. The question is not whether work waits. It is whether you can see which work waits, where, and for how long.

6.2 Define the states before you measure them

"Idle" is meaningless until you say what active means. Ours are deliberately few:

State	Definition

Active	Someone or something is working on the item right now
Waiting	The item is ready and nobody is working on it
Blocked	The item cannot proceed until something outside it changes (Chapter 7)
Done	Merged and verified, not merely closed

Elapsed time for an item is the sum of the four. The share spent active is the figure most people have never seen for their own process.

6.3 Rules for the board

1. **Idle appears as a state, in plain text.** "Idle, zero done" at the top of a lane is more useful than a green lane.
2. **No quiet tidying.** An item is not moved to waiting to make the lane look busy. States change when the facts change.
3. **Time is measured, not estimated.** Timestamps on transitions, not a person's recollection of when something stopped.
4. **Show it per stage.** A single figure hides where the wait is. Waiting in review and waiting in deploy are different problems.

6.4 Why it changes behaviour

It changes behaviour faster than any stand-up, because it is visible without anyone having to raise it. Raising it is exactly what people avoid doing. When the board shows a lane idle, the question "why is this idle" arrives without anyone having had to be the person who admitted it.

6.5 Reading the number without fooling yourself

- **Do not act on it for two weeks.** Look at it. The first reading is dominated by things you already knew.
- **Separate waiting you chose from waiting you did not.** A deliberate batch window is not waste.
- **Watch the trend per stage, not the total.** A total can improve while the worst stage worsens.
- **Do not turn it into a target.** A measure that becomes a target is gamed. Use it to ask questions.

6.6 What to measure

- **Active share of elapsed time,** per stage. This needs the state changes recorded with a timestamp; a ticket system that keeps only created and resolved cannot give it (6.3).
- **Longest wait in each stage this week,** with the item named.
- **Items idle with zero progress for more than N days,** where N is yours.

6.7 Ten minutes: add one state

If your board has only "To do, Doing, Done", add "Waiting" and "Blocked" and move the items where they honestly belong. Count how many are in each. That count is the first measurement.

7. Practice four: blocked should be the loudest thing on screen

YOU WILL	make blocked work the loudest thing on the board, and know what to measure about it.
KEY IDEA	blocked is the most expensive state and usually the least visible, because people hide blockers for reasons you can remove.
TEN MINUTES	sort blocked items to the top, with their age, and see who notices in a week.

The practice. Blocked work shows the day it blocks, at the top, before anything green.

7.1 The most expensive state

Blocked work is the most expensive state in any pipeline and usually the least visible. It surfaces at the next stand-up, or in a status report, or three weeks later when someone asks why a feature never shipped. By then the cost is not the delay. It is that everything built on top of the blocked item was built on an assumption that it would land.

7.2 What counts as blocked

A taxonomy keeps the word from meaning "I am busy":

Kind	Example	Who clears it
Decision	A choice only the technical lead or product owner can make	The named decider
Dependency	Another item must land first	The owner of that item
Access	A credential, an environment or a permission	Whoever grants it
Information	A fact the builder does not have and cannot find	Whoever knows
Defect in the process	A gate, tool or script is failing for a reason unrelated to the work	The owner of the process

The last kind is the one that hides. When a gate fails for a reason that has nothing to do with the change, the builder is blocked by the process, and a process that does not record that learns nothing.

7.3 Blocked, properly: a real example

During our pilot (Chapter 14) a story stopped. The builder had reached a point where a decision belonged to the technical lead, recorded it as blocked and asked, instead of guessing. The story waited for one decision, then resumed and finished in 17 more turns. That is the practice working: the stop was visible, it named who had to decide, and it cost one decision.

The counter-example is as useful. A run cut off by a turn limit was committed as though it were complete and reported success (DF-54). A ticket read DONE for work that had never landed, because its own commit had been refused by a gate (DF-17). Both were blocked states reported as finished ones. Nothing was loud, so nobody looked.

7.4 Rules for the board

1. **Blocked sorts to the top.** Above in-progress, above ready, above anything green. This is a sort-order change and costs nothing.
2. **A blocked item names its kind and its clearer.** "Blocked: decision, technical lead" is actionable. "Blocked" is not.
3. **It shows the day it blocks,** not the day someone mentions it.
4. **Blocked is reported with its age.** A blocker three days old is a different thing from one three hours old.
5. **Refuse to report success for an unfinished run.** An automated step that stops early must report that it stopped.

7.5 Why people hide blockers, and what removes the reason

People hide blockers because raising one costs the person something: attention, a conversation, an implied admission. The board cannot remove those costs, but it can stop making the person pay them alone. When the system raises it, "it was blocked" is a fact on the screen and not a confession.

A second reason is that a raised blocker often gets no response, so the person learns not to bother. The practice is therefore two-sided: blocked is loud, and blocked is *answered*, with an owner and a response expectation.

7.6 What to measure

- **Blocked items now,** by kind.
- **Median age of blocked items, and the oldest.**
- **Share of blockers cleared within one working day.**
- **Blockers caused by the process itself,** as a separate line. It should trend down.

7.7 Ten minutes: sort your board

Change one thing: sort blocked items to the top, with their age. Leave it for a week and see who notices.

8. Practice five: recompute metrics, do not remember them

YOU WILL	define each metric once, with source, owner and known limit, and recompute it on read.
KEY IDEA	a number updated last Friday is a story. Show the ones that do not flatter you.
TEN MINUTES	recompute the number you would be most embarrassed to find wrong.

The practice. Derive every number from the work record, and show the ones that do not flatter you.

8.1 A remembered number is a story

A number updated last Friday is a story, not a fact. Any metric maintained by hand drifts from reality at exactly the rate that makes it dangerous: slowly enough to keep trusting, fast enough to be wrong when it matters.

We have the scar. Our own site once printed defects per 1,000 lines as 2.64 in one place and 2.1 in another, one scroll apart. 2.64 was what the work record produced, because its scorecard counted archived closed tickets as well as live ones. 2.1 had no source at all. Both were typed. Now both places read one constant, and a build check fails if a figure reappears that the record does not produce.

8.2 The metric dictionary

The cure is to define each metric once, in one place, with its source and its formula, and to let everything else read it. Here is the shape.

Field	Example
Name	Defects per 1,000 lines
Definition	Closed defect tickets (live and archived) divided by thousands of lines of code in the product's own directories
Source	The work record, recomputed on read
Scope	The named product, not the factory
Owner	The person who answers when it looks wrong
Known limits	Falls mechanically when lines are removed
Last recomputed	Date

The "known limits" line is the one that earns its place. A figure with its limits attached cannot be misused as easily.

8.3 Show the unflattering ones

The August capture reports that 43% of work was later redone, a figure we publish with the caveat that there is no published industry norm to compare it against. Read it as our own watch item, not as a benchmark we have beaten. The same board shows queueing time rising as more work ran in parallel. We can show the direction and not a magnitude we would defend, so take it as a direction.

Publishing both is not humility. It is self-interest. A board that can only show good news tells you nothing on the day something goes wrong, which is the only day you urgently needed it.

8.4 A number that moved, and why we cannot fully explain it

The honest case is one we are living through. Between the August capture and 28 October the product's defect density moved from 2.64 to 3.41 per 1,000 lines, and the quality index from 80.6 to 77.3. In the same period a refactor removed a large block of lines, which shrinks the divisor and raises the density mechanically. Some of the rise is therefore arithmetic, and some may be real. We have not separated the two. Refreshing the headline

to today's number would have printed a worse-looking density caused partly by an improvement, and keeping the old one would have hidden that anything had moved.

What we do instead: keep the August figure labelled as a capture, show the October position beside it with its date, and say that we cannot yet split the effects. That is what recomputing metrics looks like in practice. You will often find that the number you would have remembered is not the number the record gives.

8.5 Rules

1. **One definition per metric,** in one file, read by everything.
2. **Every figure carries its date and its scope.**
3. **Recompute from the record on read.** If the figure is on a slide, the slide is a copy and the copy is never the source.
4. **A figure and its evidence move together.** If a number is refreshed, the screenshot or the query behind it is refreshed in the same change.
5. **Put the limits next to the number.**
6. **When a figure and the record disagree, the record is right.** Fix the copy.

8.6 What to measure about your measuring

- **Number of places a given metric is typed.** The target is one.
- **Metrics with no named owner.**
- **Metrics with no stated limit.**
- **Times a figure was corrected after publication.** Count them. They are the cost of remembering.

8.7 Ten minutes: pick the number you would be most embarrassed to find wrong

Recompute it from source today. If it matches, you have lost ten minutes. If it does not, you have found the practice.

9. Practice six: multiply reviewers, not the reviewer

YOU WILL	build a six-rung ladder of checks that block on failure, so a person reviews a much smaller surface.
KEY IDEA	multiply reviewers, not the reviewer. Rung five, scoring a sample of what the AI produced, is the one most teams lack.
TEN MINUTES	make one guard fail on purpose and see whether it catches it.

The practice. Build a ladder: automated checks that block on failure, then narrow human judgement only where judgement is genuinely required.

9.1 The answer to the first ceiling

If every line passes through one person, that person is your throughput, and hiring another of them is slow, expensive, and the reason you reached for AI in the first place. The way out is not more reviewers doing the same job. It is to change what a reviewer is asked to do.

A reviewer reading every line is spending human attention on things a machine can settle: does it build, do the tests pass, are secrets absent, does it touch what it should. Spend that attention instead on what a machine cannot settle: whether this is the right change, whether the design holds, whether the risk is acceptable.

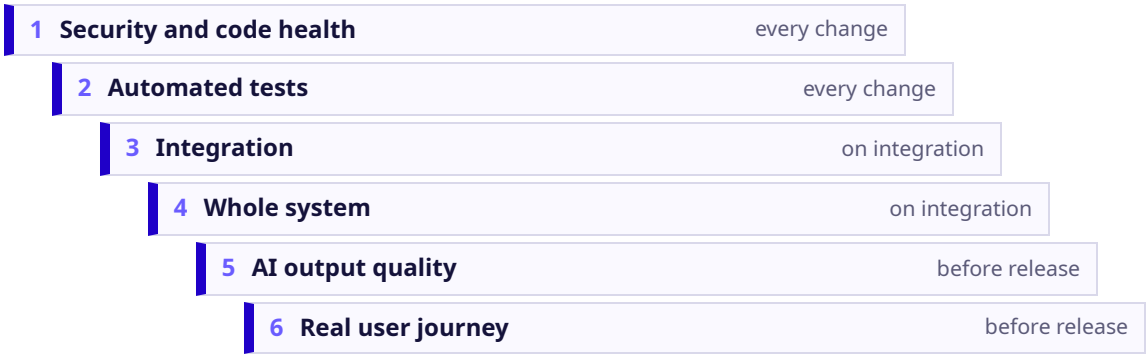
9.2 The ladder

Ours has six rungs. Each blocks on failure. Together they leave a human a much smaller surface.

Rung	What it establishes	When it runs
1. Security and code health	Insecure code, leaked credentials and unreadable work stopped at the door. In our pipeline this is OCO's gate 1: ruff, mypy, bandit and a secret scan	Every change
2. Automated tests	Every change proved against the suite, and the bar rises as the product grows	Every change
3. Integration	The parts still work together, not just on their own	On integration
4. Whole system	Everything rebuilt from scratch and the full stack scanned for vulnerabilities. In our pipeline this is OCO's gate 4: semgrep and trivy among its checks	On integration
5. AI output quality	What the AI produced is sampled and scored; nothing is assumed	Before release
6. Real user journey	Whether a person can finish the job, end to end, on the running system	Before release

Rungs 1 to 4 are standard practice done properly. Rung 5 is the one most teams lack. Rung 6 is Chapter 4.

The reviewer ladder: each rung blocks on failure, and leaves a person a smaller surface.



Rungs 1 to 4 are standard practice done properly. Rung 5 is the one most teams lack. Rung 6 is Chapter 4.

9.3 Rung five: sample and score the AI's output

You cannot read everything the model wrote. You can read a sample, and a sample scored consistently is enough to see a trend. Ten changes a week is enough to start.

For each sampled change, record:

Field	Question
Correct	Does it do what the requirement says?
Complete	Does it cover the acceptance criteria, including the unhappy paths?
Fits	Does it match the codebase's conventions and boundaries?
Claims	Does anything it states (in a comment, a commit message, a ticket note) turn out to be false?
Verdict	Accept, accept with changes, reject

The "claims" row matters more than it looks. In our own record a retrospective stated that a gate wrote to a log when it did not, and the claim was wrong for an hour before checking it (DF-58). Models state things confidently, and a sample that includes the statements, not only the code, catches this.

Track the reject rate and the claims-wrong rate over time. A rising rate is the early warning no other check gives you.

9.4 Roles with boundaries that are enforced

A "specialist" role that is only a prompt saying "act as a specialist" is not a boundary. A boundary is a list of paths a role may write and a list it must not touch, enforced by a check.

Ours are enforced twice: a deny list applied before a run, and a check of what the run actually changed, after. Having both matters, because we learned the check after the run had been enforcing nothing. The post-run check parsed only a particular heading and a fenced path list, and every role profile had written its restrictions as prose. For a period the boundary check passed because it found nothing to enforce (DF-12). The fix was a few lines. Finding it required an adversarial test: give a role a file it must not touch and confirm it is refused.

Two corollaries. **Test your guards with a thing they should stop.** A check that has never failed has not been shown to work. And **a role that cannot reach a directory cannot break it at three in the morning,** which is what makes a specialist different from an instruction.

9.5 Human gates: few, and by design

Where a person is in the loop, say why:

Gate	Why a person	What they decide
Accept a specification	Scope is a judgement about value	What we are building
Accept a design	Trade-offs are a judgement about risk	How
Review a build	The merge decision is accountable	Whether it lands
A blocked decision	Only the decider can	The choice, recorded

In our pilot, two of the 12 manual interventions were human gates working as designed: the product owner accepting the specification, and the technical lead making a decision a build had stopped to ask for. The other ten were things the process should have done itself (one of them only partly). The measure of a good ladder is not zero humans. It is that every human step is one only a human can do.

9.6 What to measure

- **Share of changes blocked by each rung,** and what they were blocked for.
- **Reject rate of sampled AI output,** and the claims-wrong rate.
- **Human review time per change,** and what it was spent on.
- **Boundary violations caught before and after a run.** If the second is non-zero, the first is leaking.

9.7 Ten minutes: test one guard

Pick one automated check you rely on. Make it fail on purpose: commit a fake secret, break a boundary, delete a required file. If it does not fail, you have found a check that was passing because it was looking at nothing.

10. From a requirement to a ticket that is ready

YOU WILL	build the chain from a requirement to a ticket that is ready, with every link naming its parent and a gate that refuses a broken chain.
KEY IDEA	the largest gain was in the stages before the code, not at it.
TEN MINUTES	follow a ticket that shipped last week upward using only links, and count where you had to ask a person.

Most AI-assisted teams have put AI at one step and left the rest as it was. The largest gain was in the stages before the code: the chain between "a requirement arrived" and "a ticket exists that someone could build without asking". This chapter covers stages S0 to S6.

10.1 What was wrong before

Before the renovation, our lifecycle had a requirement turn straight into flat tickets. There was no specification, no architecture document, no epic or story as an entity, so a requirement's rationale could not be followed down to its stories and re-slicing meant editing tickets by hand (L3 in our as-is review). Brownfield understanding was shallow: an imported project's "discovery spec" was mostly narrative from a language model over a thin graph (L4). And nothing stood between planning and build: tickets reached "in progress" with unresolved unknowns and no design (L9).

Each of those is a thing you can check in your own process.

10.2 One artifact chain, every link checkable

The target is a chain where each artifact names its parent by id:

```
requirement -> evidence -> brief -> specification -> design/decision record
            -> epic -> story -> code and tests -> review -> retrospective
```

What makes it a chain and not a folder of documents is that a **machine can follow it.** A story that does not name the specification it implements is refused. A specification that is still a draft cannot be implemented. If you cannot follow a story to its requirement with a command, your traceability lives in people's heads.

One artifact chain: each link names its parent, and a machine can follow it.

1
Requirement
AI Classifies, deduplicates, asks
PERSON Product owner: is it worth doing

2
Evidence
AI Reads the code, summarises
PERSON Architect: is the picture true

3
Brief
AI Drafts options
PERSON Product owner

4
Specification
AI Drafts, finds gaps, checks citations
PERSON Product owner accepts

5
Design and decision
AI Proposes options
PERSON Technical lead accepts

6
Epic and story
AI Slices into stories
PERSON Product owner: priority

7
Code and tests
AI Writes both
PERSON The gates, if they hold

8
Review
AI A first pass, several lenses
PERSON Technical lead and QA: merge

9
Retrospective
AI Summarises, drafts
PERSON Technical lead and product owner

The readiness gate sits between the story and the code: no parent, no accepted specification, or fewer than three falsifiable criteria, and the story is bounced with the reason.

The mission: AI does the drafting, a person holds the decision, a machine refuses a broken link

Every link has the same three parts. The AI produces the next artifact, a named person decides whether it is true, and a check that cannot be argued with refuses the chain if the parent is missing. The middle column is what the model is good at. The right-hand one is the decision we do not hand over (Chapter 2.2).

Link	AI does	A person holds	What refuses a broken link
Requirement	Classifies, deduplicates, asks clarifying questions	Whether it is worth doing	It has an id
Evidence	Reads a large codebase and summarises its structure	Judging whether the picture is true	A summary with no evidence behind it is not accepted as one
Brief	Drafts options and trade-offs	The product owner chooses	It names the requirement
Specification	Drafts, finds gaps, checks citations	Scope and acceptance: the product owner accepts	A draft cannot be implemented
Design and decision record	Proposes options	The decision, and the record of why: the technical lead accepts	It names the specification; it is marked Proposed until accepted

Epic and story	Slices into stories	Priority	The readiness gate: a parent, an accepted specification and at least three falsifiable criteria, or the story is bounced with the reason
Code and tests	Writes both, and the test scripts that run them	Nothing, if the gates hold	A boundary check before and after the run, then OCO's security gates (code health and secrets, then semgrep and trivy) before the change can merge
Review	A first pass through more than one lens	The decision to merge	The gates that block on failure (Chapter 9)
Retrospective	Summarises and drafts	What the retrospective changes	It names the decisions as lines

Why it matters

It is what lets AI work without asking. The test of a ready ticket, at the start of this chapter, is that someone could build it without asking a question. Each question a builder has to ask is a link that was missing from the chain.

It removes the largest hidden cost. Four of the fourteen findings in our own as-is review were this one: no specification, epic or story as an entity, no readiness gate, traceability only half built, and scope changes as untracked prose (Chapter 15.1). Most pipelines have the same gap, because the chain lives in the heads of the people who happen to remember it.

It keeps the human load small and aimed. A person who has to judge every link reads far more than one who judges only where judgement is needed. The machine refusal takes the rest, so the human attention goes to scope, to the decision and to the merge, which is where it cannot be replaced (Chapter 9).

It makes a failure findable. When a defect appears, you follow the chain upward from the code to the requirement and downward to the test that should have caught it. Without the chain, an incident starts with reconstructing it.

10.3 Evidence before opinion

On an existing system, planning starts from evidence, not from a summary. The two rules we hold:

1. **Every claim about the system cites something observable,** a file, a route, a run, a capture, with a confidence: observed, derived or inferred.
2. **An unknown stays an unknown until a person or a spike settles it.** It does not become a plausible sentence. A residual-unknowns list travels with the evidence, and a story whose scope touches an open unknown is not ready.

The second rule is the one that matters. A language model will fill a gap with fluent text. The process has to make the gap visible and refuse to proceed over it.

10.4 The readiness gate

Between the backlog and the build sits a gate that fails closed. A story is ready only if all of these hold. They are mechanical, and each is reported per story:

Check	Rule
R1	The story has a parent epic that exists and is not closed
R2	It implements a specification whose status is accepted
R3	At least three acceptance criteria, each a command or Given/When/Then
R4	If a design is needed, the design artifact or decision record named in the story exists
R5	No open unknown in the plan or in a referenced evidence pack covers this story's scope
R6	The "where to start" paths in the story exist
R7	The assignee has a role profile

On the pilot in Chapter 14, the gate refused both stories until the specification was accepted, which is it doing its job: it fired before either story went into progress.

10.5 Tickets written for the person who will kill them

The ticket standard is strict, and the point is that a ticket is read by the person who will resolve it, not the person who wrote it. Ours require an outcome-shaped title, who is blocked and why, a reproducible evidence command, the expected terminal state, a scope boundary, where to start, why this owner, and falsifiable acceptance criteria. The tool that creates tickets refuses anything below the bar, and a refused ticket consumes no id.

That sounds bureaucratic until you see the alternative: stories written as "improve X", each of which needs a conversation before anyone can start. A model asked to build "improve X" will build something. A ticket that cannot be built wrongly is a ticket with falsifiable criteria.

10.6 Three tracks

Right-sized ceremony is what stops people routing around the process:

Track	When	What runs
Quick	A bug or small change in one component	Intake, one story, build, review
Standard	A new feature in an existing product	Intake, evidence if stale, requirements, design if needed, backlog, readiness, build, review, close
Brownfield / large	Importing a system, replacing a legacy feature, or a cross-component initiative	Everything, with full evidence

The track is part of the analysis, and the product owner can overturn it.

10.7 Ten minutes: follow one ticket upward

Pick a ticket that shipped last week. Follow it to the requirement that justified it, using only links. Count the steps where you had to ask a person. Each is a link missing from the chain.

11. Which model may see what

YOU WILL	classify each project by data class and put every model call behind one decision point.
KEY IDEA	where the material goes is a separate question from quality, and a guideline is not a control.
TEN MINUTES	search your repositories and jobs for the model providers you use and list the calls.

AI in the lifecycle raises a question that has nothing to do with quality: where does the material go? A specification, a codebase or a log may belong to a client, and "we used a model" is not an answer to "who saw it".

11.1 What we found

In our as-is review, the house rule said language-model calls stay on a local endpoint, while in practice every job run by the factory's background daemon used a cloud-hosted command-line tool (L5). The rule could not be audited or enforced, and client data had no stated boundary. A written policy that the code does not follow is worse than no policy, because it is believed.

11.2 The rule we settled on

LLM use is tiered by the class of the data, and the router enforces it.

Data class	What it covers	Where it may go
Internal	The factory's own code and documents	A hosted tool may be used for development-time work
Client-confidential	A client's material, and anything third-party	Only a local model, by default
Third-party binary	A compiled artefact we do not own	Only a local model

Four properties make this a control and not a statement:

1. **The class is declared once, per project,** in the project's own configuration. A project that has not declared one is treated as the most restrictive class, not the least.
2. **One router decides every call.** No stage picks its own model; it asks. Every decision, allowed or refused, is logged.
3. **A refused job does not run.** The background daemon declines a job the router refuses, without starting anything.
4. **Exceptions are written down.** Where a local model is not good enough for a use case, the owner may grant one backend for named job purposes of one project, recorded with a reason, a grantor and a date. A grant never changes the class, and every use is logged as an exception. We have granted one.

11.3 Why a router, and not a guideline

A guideline relies on each person and each script remembering. A router gives you a log you can read. When someone asks "did any of this client's material reach a hosted model", the answer is a query over the log, not a

recollection, and the exception grants are the only place the answer can be yes.

For a client this is the useful form: not "we are careful", but "here is the rule, here is where it is enforced, here is the log, and here is the one exception, with who approved it and why".

11.4 What you can do on Monday

1. **Classify your projects.** Write down, for each, whether its material is yours, a client's or a third party's.
2. **Find every place a model is called.** Scripts, plugins, IDE tools, background jobs. Most teams discover calls they did not know about.
3. **Put the decision in one place,** even if it begins as a short function and a log file.
4. **Make the default the safe one.** An unclassified project should be refused, not allowed.
5. **Write the exceptions down,** with a name and a date.

11.5 A limit to be honest about

A router protects what goes through it. It does not stop a person pasting a document into a browser. That is a training and policy matter, and no tool closes it. The control we are describing is for the automated paths, and it is only as complete as your list of them.

11.6 Ten minutes: list your model calls

Search your repositories and scheduled jobs for the names of the model providers and tools you use. The list you get is your starting inventory, and it is nearly always longer than the list in your head.

12. Building and reviewing inside a fence

YOU WILL	build inside a fence: one isolated copy per ticket, an envelope of what the builder may touch, and review through more than one lens.
KEY IDEA	what matters is what the builder may touch, what it must prove, and what happens when it stops early.
TEN MINUTES	make one automated step stop halfway and read what it reports.

The build stage is where most teams start and where the least needs inventing. What matters is the fence around it: what the builder may touch, what it must prove, and what happens when it stops early.

12.1 One isolated copy per ticket

Every ticket builds in its own working copy of the repository, on its own branch. Two builds cannot step on each other, a failed build leaves nothing behind in the main tree, and "what did this change touch" has a precise answer: the diff of that branch.

Isolation also exposes your assumptions. Several of our findings only appeared when builds ran from working copies other than the main checkout: a test that derived the factory's home directory from its own path and

found the wrong one inside a nested copy (DF-50), a baseline shared by every copy so that one branch's improvement raised the bar for all (DF-55), and a stale untracked file in the parent directory that hid a bug on one machine and exposed it on another. Parallel, isolated builds are a stress test of your process. They find the places where your tooling quietly assumed there was one copy.

12.2 The envelope

Around each build:

Element	What it does
Role profile	States what the assignee may write and must not touch
Pre-run deny list	Refuses paths the role may not reach
Post-run boundary check	Confirms the diff stayed inside the remit
Gates	The ladder in Chapter 9, run before the branch can land
Budget	A turn limit per run, with resumption recorded
Ledger	What was attempted, how many times, with what result

The envelope is the same for every ticket and every role, so the question "what is this run allowed to do" never depends on who started it.

12.3 Things that went wrong in the envelope

These are real findings from running our own work through it, each fixed or ticketed:

- **A run cut off by its turn limit was committed as complete** and reported success (DF-54). The fix: an incomplete run reports that it is incomplete.
- **A small change skipped the build procedure's planning step,** and the run staged a stray symlink along the way (DF-49).
- **An automated build staged an orchestrator event file and rewrote the coverage baseline,** a mistake we had already met by hand (DF-40). Automation repeated the human error faster.
- **A tool update changed behaviour between our tests and our runs** and nothing noticed (DF-57). We now pin the version and fail when it differs.
- **Giving a run enough turns mattered.** Runs with a 150-turn budget finished in one go. Runs with 40 needed repeated resumption, each costing more than a larger budget would have (DF-60).

Several of those are the same shape: the system stopped for a reason and reported something else. A pipeline's honesty is how it behaves when it fails.

12.4 Review: more than one lens

A reviewer is more useful with a checklist of lenses than with a blank page. We run a multi-lens first pass (does it meet the acceptance criteria; is it secure; does it match the architecture; are the tests meaningful; does it respect the boundary) and then a technical lead reads the result with the findings in front of them. The lead decides. The first pass narrows what has to be looked at; it does not replace the look.

One lesson from using a model as a reviewer: **it agrees with the author more than you want.** Give it a specific job ("find a way this change breaks an existing caller") and not "review this".

12.5 Landing

Landing a change should be one command that does the mechanical part: merge, rename the ticket, update the board, clean up. And it should refuse when the underlying work did not land. We had a landing helper that renamed a ticket to DONE even though no feature commit existed, because the commit had been blocked by a gate (DF-17). The fix was a guard: refuse to mark a ticket done without proof the change is on the branch.

12.6 What the build depends on

The envelope above governs what the run does. It says nothing about what the run adds. A model writes a package name and a version into a requirements file from memory, and no step in the loop reads that package's advisories. Put one check where the dependency is declared: read the project's manifests and compare them with the exploited-vulnerability list, then keep the answer with the change. The same tool, without its project subcommand, checks the server a release lands on, so the deployment step has a check too. Appendix G shows both, the first results over our own seven projects, and the mistakes in it.

12.7 Ten minutes: ask what a stopped run says

Take one automated step in your pipeline. Make it stop halfway (kill it, exhaust its budget). Read what it reports. If it says "ok", you have found where success is declared without being checked.

13. Closing the loop: retrospectives, decisions and what the process learns

YOU WILL	close the loop with retrospectives that change something, findings that have a home, and replanning that keeps the record.
KEY IDEA	a lifecycle that never examines itself accumulates its defects.
TEN MINUTES	extract every decision from your last retrospective as a line beginning "Decision:".

A lifecycle that never examines itself accumulates its defects. This chapter covers the last two stages and the habit that holds the rest together.

13.1 Retrospectives that change something

We close each epic with a retrospective kept beside its record, and each sprint with the usual one. Two rules turned them from ritual into a mechanism:

1. **A retrospective ends in decisions, written as lines that can be found:** "Decision: ...". A retrospective that ends in a feeling cannot be acted on or audited.

2. **A retrospective is checked like any other claim.** One of ours stated that a gate wrote to a log when it did not, and was corrected an hour later by checking (DF-58). Retrospectives are written by the same kind of author, human or model, as everything else, and are subject to the same fault.

13.2 Findings need a home

For 60 findings from the renovation, we kept a dogfood log: one row each, with where it was found, what was observed, the impact and a decision. The decision is one of three: fixed where it was found, a design change in the plan, or a ticket for later.

The log taught us something about the log. When the owner asked how many were resolved, 16 of the first 37 were still open, and 10 said "ticket" with no ticket filed. The log recorded the intent and not the ticket (DF-38). The honest count was lower than we had believed. The fix was that a finding does not count as dispositioned until the ticket exists.

Of the 60 rows at the time of writing, 30 are marked fixed, 14 are design changes and 18 are tickets for later (some rows carry more than one). That is a process finding itself: **a third of what you find is not fixable on the day,** and a plan that assumes otherwise is dishonest.

13.3 Replanning without losing the record

Scope changes are inevitable. They become untracked prose unless there is a mechanism: a way to re-slice stories with a decision log that says what changed and why. We had retrospectives per sprint but no correct-course flow at epic level and stories could not be re-sliced with a log (L14). Adding one meant that a changed plan is a record, not a rumour.

13.4 What the process learns

Each stage's output names its parent, so the chain can be queried: which requirements produced the most rework, which gates stopped the most changes, which role's boundary was hit most. That knowledge lives in the work record and, where useful, in a graph linking requirements, evidence, epics and stories. The point is not the graph. It is that the next planning session can ask "what happened last time we did something like this" and get an answer from the record.

13.5 Ten minutes: write one decision line

Take your last retrospective and extract every decision as a line beginning "Decision:". If there are none, the retrospective produced no commitments.

IV. WHAT BROKE

14. One real requirement, end to end: twelve interventions

YOU WILL	see what one real requirement needed on the way through our own process: twelve manual interventions, and what each was.
KEY IDEA	any process looks good on a slide. Count how often a person had to step in.
TEN MINUTES	list every moment in your last feature when a person did something a script could have done.

Any process looks good on a slide. The test is to push one real requirement through all of it and count how often a person had to step in. We did, on 22 October 2026, and it needed twelve.

14.1 The requirement

"An architecture description can be checked for citations by one factory command." It was a real gap, found the same day: a pilot for another product had needed a hand-written citation check. It was small enough to finish and real enough to matter.

14.2 The path

Step	How	What the record shows
Intake	Submitted to the live work-record service	A requirement created, analysis job queued
Analysis	A background analyst job	Two stories, track "brownfield"
Approval	An epic created against a branch working copy	An epic and a specification (draft)
Specification	Product owner accepts it	Status "accepted" in the specification index
Readiness	The gate from Chapter 10	Checks R1 to R7 pass for both stories, before either went into progress
Build	The orchestrator, with a 150-turn budget	One story in 22 turns. The other stopped once for a technical-lead decision, then finished in 17 turns
Review	The technical lead: tests and live checks	The pilot document had zero offending lines; the command refused bad input on a copy
Land	One command	Merged; the epic closed with its retrospective

14.3 The twelve

This is the table we published to ourselves, lightly shortened. The right-hand column is the one to read.

#	What a person had to do	Should the process have done it?	Disposition
1	Restart the work-record service: it had been running nine-day-old code, so a recent change never took effect	Yes	Fixed 23 October (ODSF-264): a factory merge now reloads the long-running processes that run the changed code
2	Re-run the analysis: the daemon kept a 728-character summary, not the 14,774-character answer	Yes	Fixed
3	Re-route both stories to the right owner before approval	Yes	Fixed
4	Approve in-process: the approve call takes its epic mode only from configuration	Partly	Fixed 23 October (ODSF-265): a product owner can approve one requirement as an epic without changing the project's default
5	Discard the first approval: stories took a fallback path and lost their routing	Yes	Fixed
6	Withdraw two tickets filed as the wrong type and refile as stories	Yes	Fixed
7	Record the specification's ticket list by hand: the approval's bookkeeping did not land	Yes	Fixed 23 October (ODSF-266): approval records its tickets on the spec and in the spec index every time
8	Set aside a stray re-approval whose draft specification collided with this one's id	Yes	Fixed 23 October (ODSF-267): two specifications can never be given the same id
9	Product owner accepts the specification	No: a human gate by design	n/a
10	Record the decision a story was blocked on	No: a technical-lead decision by design	n/a
11	Bind the build working copies before landing	Yes	Fixed
12	A "help" flag on the id allocator allocated an id	Yes	Fixed

14.4 How to read it

Twelve interventions for one small requirement is a lot, and we would rather you saw the number than a story with none. Six were defects in the factory, fixed during the run. Four more were still open when we wrote the run up and were fixed the next day, 23 October (ODSF-264, 265, 266 and 267). We have not re-run the pilot since, so we cannot say the same path would now need fewer than twelve. Two are human gates that worked as designed.

What this proves, and what it does not:

- **Proven.** A requirement can travel intake, analysis, epic, readiness, build, review, landing and epic close with every gate firing where designed. The readiness gate refused both stories until the specification was

accepted. The build refused to guess a decision that belonged to the technical lead, and stopped to ask.

- **Not proven.** The same path without a person watching. Items 1, 4, 7 and 8 have closed tickets, not a second run, so we do not know that they hold together; until a repeat run needs fewer interventions, "autonomous" would be the wrong word.

14.5 Why we are telling you

Because the number twelve is more useful to you than a polished demo. It tells you what to expect when you try this on your own process: not that it fails, but that the first end-to-end run finds the seams between parts that each worked alone. At least five of the twelve are the same kind: **state that one part held and the next part did not see** (a stale process, a lost summary, a routing lost on a fallback, bookkeeping that did not land, an id clash). If you build a lifecycle from parts, expect to spend your first month on the joins.

14.6 Ten minutes: count the hands

Think of the last feature you shipped. List every moment a person touched something a script could have done. That list is the size of your own "twelve", and the order to fix it in is by how often each recurs.

15. What was wrong before, and what the pattern is

YOU WILL	read the fourteen findings from our as-is review, the pattern across sixty, and how to run your own.
KEY IDEA	the most common shape is something reporting success for a reason that was not success.
TEN MINUTES	pick the check you trust most and try to get a violation past it.

Before building the target lifecycle we spent a day writing down what the existing one actually did, component by component, and where it broke. Fourteen findings. It was not flattering, and it is the most reusable part of this playbook, because most of the fourteen exist in some form in most teams' pipelines.

15.1 The fourteen, grouped

Group	Findings (as we numbered them)	What it looked like
Process written for one product	L1, L13	Every run was told it worked on one particular product, and planning prompts carried that product's vocabulary into every other project
Authority without a fence	L2, L6	Six roles had no profile; implementation ran with permissions skipped and the boundary check came after the run
Chain missing	L3, L9, L10, L14	No specification, epic or story as an entity; no readiness gate; requirement-to-test traceability half built; scope changes as untracked prose

Shallow understanding	L4	Brownfield discovery was mostly narrative over a thin graph
Policy and practice differ	L5, L12	The written rule for models was not what ran; the standard that refused tickets could not be read in the repository that enforced it
Vocabulary drift and duplicated state	L7, L8, L11	Two spellings of the same ticket type; two state machines; gate scopes hard-coded

15.2 Three that are probably in your pipeline

A guard that guards nothing (L6 and later DF-12). A check that happens after the thing it should prevent, or that parses a format nobody uses. Test it by trying to violate it.

A policy that code does not follow (L5). If you cannot point to the line that enforces a rule, it is a hope. Ours said models stay local while every daemon job used a hosted tool.

A standard you cannot read where it is enforced (L12). The rules that refused a ticket were cited by two scripts and did not exist in the repository that ran them. People working inside a rule they cannot read invent their own.

15.3 The pattern across 60 dogfood findings

Reading the sixty findings that came after, one shape recurs more than any other: **something reported success for a reason that was not success.**

What reported success	Why it was not
A boundary check	It found nothing to enforce (DF-12)
A ticket marked DONE	Its commit had been refused (DF-17)
A build marked ok	It had hit its turn limit (DF-54)
A coverage gate	It measured one directory (DF-05)
A gate run	It had changed the baseline it then compared against (DF-53)
A retrospective	Its claim about a log was false (DF-58)
A test fixed	Only for the two places it happened to find (DF-52)
A version-pinned tool	It had updated itself (DF-57)

Every practice in Part II is a response to this shape. **Make success hard to counterfeit,** and make failure louder than success.

15.4 How to run your own as-is review

You can do this in a day with two people.

1. **List the components of your pipeline,** from the tracker to the deploy.
2. **For each, write what it actually does,** by reading it, not by asking. Include the scripts nobody owns.
3. **Write the rule each is supposed to enforce,** and find the line that enforces it.
4. **For each guard, try to violate it.** Record what happens.

5. **Number the findings and give each one an evidence command,** so it can be re-checked.
6. **Decide a disposition for each: fix now, change the design, or file it** and make sure it is filed.

15.5 Ten minutes: find one guard to attack

Pick the check you trust most. Spend ten minutes trying to get a violation past it. If you succeed, you have your first finding.

V. ADOPTING IT

16. Where are you? A self-assessment

YOU WILL	score yourself 0 to 4 on each of the six practices, from evidence rather than aspiration.
KEY IDEA	where you cannot name the evidence, lower the score.
TEN MINUTES	fill in the table with a pencil and write the evidence beside each score.

Adoption starts with an honest position. This is a five-level scale for each of the six practices, written so that you can answer it from evidence and not from aspiration.

16.1 How to use it

For each practice, pick the highest level for which you can **show the evidence today**. If you would have to explain, the answer is the level below. Do it with two people independently and compare. The disagreements are the useful part.

16.2 The scale

Level	In plain words
0	We do not do this
1	We do it when someone remembers
2	It is a rule, written down, mostly followed
3	It is enforced by a check that can fail
4	It is enforced, measured, and the measure is on a board people see

The scale: answer from evidence, not from aspiration.

0	1	2	3	4
We do not do this	We do it when someone remembers	It is a rule, written down, mostly followed	It is enforced by a check that can fail	It is enforced, measured, and the measure is on a board people see

16.3 The six practices

Practice	Level 1 looks like	Level 3 looks like	Level 4 looks like
1. A person can finish the job	Someone clicks through before a big release	An automated journey runs against the deployed system on each release	It blocks releases, overrides are recorded, journeys defined and passing are on the board
2. Gates tighten with size	Coverage is discussed when it drops	A ratchet gate: the floor cannot fall; new code carries tests	Several thresholds scale with size; the baseline is committed and reviewed

3. Idle reported	Nobody knows how long things wait	Timestamps on every state change	Active share per stage is visible and trended
4. Blocked is loudest	It comes up at stand-up	Blocked items sort to the top with a kind and a clearer	Age and response time are measured; process-caused blockers are a separate line
5. Metrics recomputed	Numbers are in slides	One definition per metric, read by everything	Every public figure is dated, scoped, limit-labelled, and its evidence moves with it
6. Reviewers multiplied	One person reads everything	Automated gates block; humans see the remainder	AI output is sampled and scored; boundaries are enforced before and after a run

16.4 Reading your result

- **Mostly 0 to 1.** You are not behind. Most teams are here. Start with Practice 4 and Practice 3. They cost nothing and change behaviour fastest.
- **A mix with a 3 or 4 in testing and 0 in visibility.** Common for engineering-led teams. Your pipeline is good and cannot tell you when it is not.
- **High on visibility, low on gates.** You can see the trouble and do not yet prevent it. Start with Practice 1.
- **Everything at 3 and 4, but no sampling of AI output.** The last rung is the one that scales review; Chapter 9.

16.5 Two warnings

Do not aim for level 4 everywhere. Aim for level 3 on all six before level 4 on any. A measured process you cannot enforce is a dashboard of failures.

Be suspicious of a level you got to without trying. If your gate has never failed, ask whether it can (Chapter 9.7).

16.6 Ten minutes: score yourself

Fill in the table with a pencil. For each practice write the evidence you would show. Where you cannot name the evidence, lower the score.

17. A 30, 60 and 90 day plan, and who decides

YOU WILL	sequence adoption over thirty, sixty and ninety days, name who decides, and choose between building, buying and doing it by hand.
KEY IDEA	each step is cheap and shows its value, so the next is easy to argue for. The first month adds measurement and no tooling.
TEN MINUTES	copy the first thirty days, strike what you already do, and put names and dates on the rest.

You do not need a programme. You need a sequence in which each step is cheap and shows its value, so that the next one is easy to argue for.

DAYS 1 TO 30

Make it visible

- Sort blocked to the top, add Waiting and Blocked
- Report idle honestly for two weeks, without acting
- Recompute one number from source
- Write the customer sentence and the journeys behind it
- Sample ten AI-written changes and score them

Measurements, no new tooling

DAYS 31 TO 60

Make it enforced

- Build the first journey check, run it on the deployed system
- Wire it to block a release, with a recorded override
- Coverage as a ratchet on the diff and a floor on the whole
- One definition per headline metric, in one file
- A post-run check of what an automated change touched

Checks that can fail

DAYS 61 TO 90

Make it systemic

- A readiness gate: no story built without a parent
- Classify projects by data class; model calls behind one decision point
- A retrospective that ends in decision lines
- Your own as-is review of the pipeline
- Tie one more threshold to size

A process that examines itself

17.1 The first thirty days: make it visible

Week	Do	Why first
1	Sort blocked items to the top of the board, with their kind and age. Add "Waiting" and "Blocked" states	A sort order and two columns. Costs an afternoon, changes behaviour
1 to 2	Report idle honestly for two weeks. Do not act on it	You are collecting the first honest measurement
2	Pick the one number you would be most embarrassed to find wrong and recompute it from source	Finds out how much you can trust your own figures
3	Write the sentence a customer would say when your product works, and write the five to ten journeys behind it	The input to practice one
4	Sample ten AI-written changes and score them	The first look at rung five, and a baseline

By the end of the month you have measurements and no new tooling. That is deliberate.

17.2 Days 31 to 60: make it enforced

Do	Notes
Build the first journey check and run it against the deployed system	Expect it to fail. That is the first honest measurement of whether the product works
Wire it to block a release	With a recorded override path
Turn coverage into a ratchet on the diff and an unmovable floor on the whole	Commit the baseline
Write one definition per headline metric, in one file	Include owner and known limits
Add a post-run check of what an automated change touched	And test it with a thing it should stop

17.3 Days 61 to 90: make it systemic

Do	Notes
Add a readiness gate: no story is built without a parent, an accepted specification and at least three falsifiable criteria	The first link in the chain
Classify projects by data class and put model calls behind one decision point	Chapter 11
Run a retrospective that ends in decision lines	And check one of its claims
Run your own as-is review of the pipeline	Chapter 15.4
Tie one more threshold to size	Review depth or blast radius

17.4 Roles and who decides

Even in a team of three, name who holds each decision. The playbook assumes these roles. One person may hold several, but each decision should have one name.

Role	Holds
Product owner	What gets built; accepts specifications
Technical lead	Accepts designs; reviews builds; decides blocked technical questions
Process owner	The gates, the board rules and the metric dictionary; hears every "this check is wrong"
QA	The journeys, and the ratchet
Builders (human or AI)	Work inside their boundaries

The **process owner** is the one most often missing. When nobody owns the gates, they decay.

17.5 Build, buy or do it by hand

For each practice, ask whether the first version needs tooling at all.

Practice	By hand first	When to automate
Blocked on top, idle visible	A board with sort order and timestamps	When you have more than one team
Journeys	One scripted browser test	When you have more than three
Ratchet	A committed number and a script	Straight away; it is a few lines
Metrics	A file of definitions and one script	When a figure appears in more than one place
Sampling	A spreadsheet	When ten a week stops being enough

None of this requires our tooling.

17.6 What goes wrong in adoption

- **Starting with tooling.** A tool without a practice produces a dashboard nobody reads.

- **Making the first measurement a target.** Idle share and rework rate become gamed the week they get a goal attached.
- **Fixing the number and not the cause.** A journey made to pass by loosening what "done" means.
- **No process owner.** The gates decay, and the first sign is a quiet override.
- **Trying to do the lifecycle before the practices.** The chain in Part III only helps a team that already sees its own work.

17.7 Ten minutes: write your first thirty days

Copy the table in 17.1, strike what you already do, and put names and dates on the rest.

18. The metrics, defined

YOU WILL	use one page of metric definitions so that when a figure is quoted, everyone means the same thing.
KEY IDEA	each metric is a candidate for the dictionary in Chapter 8, with its source and its limit.
TEN MINUTES	define one metric with its source, owner and known limit.

One page of definitions, so that when a figure is quoted, everyone means the same thing. Each is a candidate for the dictionary in Chapter 8.

18.1 Flow

Metric	Definition	Watch for
Active share	Time in the Active state divided by elapsed time, per item, then the median	Rising while the worst stage worsens
Waiting time	Time in the Waiting state, per stage	A stage where it only ever grows
Blocked count and age	Items in Blocked, with their time in that state	Items older than your response expectation
Cycle time	Elapsed time from "ready" to "done" (merged and verified)	A median that hides a long tail; report the 90th percentile

18.2 Quality

Metric	Definition	Watch for
Journeys passing	Defined journeys that completed on the last release run, over defined journeys	A journey not run recently
Changed-line coverage	Share of lines added or changed that are covered by tests	Coverage that rises while assertions thin out
Gate pass rate	Gate runs that passed over gate runs	Very high, or very low: both mean the gate may be measuring the wrong thing

Defects per 1,000 lines	Closed defect tickets over thousands of lines in the product's own directories	A divisor that moved (a refactor)
Sampled reject rate	Sampled AI changes rejected over sampled	A claims-wrong rate rising

18.3 Rework and trust

Metric	Definition	Watch for
Rework rate	Items later redone over items completed	No published norm; use as your own trend
Overrides	Releases or merges that bypassed a block, recorded	Any override with no name and no reason
Boundary violations	A run changing files outside its role's remit, caught before and after	A non-zero "after" count
Findings dispositioned	Process findings with a fix, a design change or a filed ticket, over all findings	Findings that say "ticket" with no ticket

18.4 A weekly sheet

Fifteen minutes on a Friday, one page:

```
Week of:            ____
Blocked now:        __ (oldest: __ days; process-caused: __)
Active share:       __% (worst stage: ____)
Journeys passing:   __ of __   Releases blocked: __   Overrides: __
Changed-line coverage (merged this week): __%
Sampled AI changes: __ scored; rejected __; claims wrong __
Open findings:      __ (without a ticket: __)
One thing we learned: ____
One change we are making: ____
```

18.5 Ten minutes: define one metric

Choose one metric from this chapter and write its definition, source, owner and known limit in the format of Chapter 8.2. If you cannot say its source, that is the work.

APPENDICES

Appendix A. The numbers, and their dates

Every figure used in this playbook, in one place, with its scope and the day it was read. None is a price, and none is a promise that the same will hold for you.

A.1 The published capture, 2 August 2026

From our own delivery board for the live product the numbers describe, past 165,000 lines. The board's screenshot is at namiq.io/results.

Measure	Value	Note
Work completed	98% (1,355 of 1,381 items closed)	Closed is not the same as proven
Test coverage	80%	In line with a commonly published 70 to 80% target
Defects per 1,000 lines	2.64	Within the 1 to 5 commonly published for a disciplined team
Work later redone	43%	No published norm. Our own watch item
Calendar time to reach this scope	17 days	A single controlled build where we wrote the requirements and chose the stack. Not a rate

A.2 The position on 28 October 2026

Read from the two live boards that day.

Scope	Measure	Value	Read
The product	Issues closed	1,391 of 1,415 (98%)	
The product	Work packages completed	178 of 179	
The product	Lines of code	136,631	Down from 165,000 after a refactor
The product	Quality index	77.3	Down from 80.6
The product	Defects per 1,000 lines	3.41	Up from 2.64; part of the rise is arithmetic from the smaller divisor
The product	Gate pass rate	89.1%	
The factory itself	Issues closed	233 of 254 (92%)	
The factory itself	Lines of code	92,435	
The factory itself	Quality index	83.2	
The factory itself	Defects per 1,000 lines	1.17	A smaller and newer codebase

Claude's working time	Days Claude worked	51 of 57 days	5 September to 31 October 2026, read on 31 October from Claude's own logs. The logs go back no further
Claude's working time	Working time	about 100 hours; median 1.4 hours on a working day	82 to 102 hours depending on how long a pause counts as still working. Parallel sessions are counted once
Claude's working time	Working time on a branch that names a ticket	about a fifth (20% to 22%)	The rest was the website, the factory home and other work no ticket names
Claude's working time	Share of a ticket's working span spent working	21% (median ticket; 11% to 21% over the thresholds)	74 tickets with Claude time on their own branch, first to last Claude activity. Middle half of tickets 2% to 100%; 0.1% of the time from creation to resolution
Claude's working time	Share of a working session spent working	68% (52% to 80%)	The rest is a person reading and typing. The range is how long a pause counts
The renovation	Epics closed	All ten (E0 to E9) and the pilot epic	
The renovation	Dogfood findings logged	60	30 fixed, 14 design changes, 18 ticketed
The renovation pilot	Manual interventions for one requirement	12	6 fixed in the run, 4 fixed the next day (ODSF-264 to 267), 2 human gates by design. Not re-run since

A.3 How to read them

The August capture and the October position are two readings of boards that keep running. They are not a before and after of one change. Where the October number is worse, it is printed; where we cannot say why, we say so. The rows headed Claude's working time are not scorecard readings: they were measured on 31 October 2026 from Claude's own logs.

Appendix B. Templates

Copy, delete what you do not need, and keep the lines that make you find something out.

B.1 Definition of done, with a journey

```
Change:              <ticket id and title>
Parent:              <epic id> / <specification id, status: accepted>
Acceptance criteria (each a command or Given/When/Then, at least three):
  1.
  2.
  3.
Journey affected:    <the journey sentence>   Run against: <environment>
Journey result:      completed | failed | not run (reason)
Gates:               security __  tests __  integration __  whole system __
Changed-line coverage: __%   Floor held: yes | no
Boundary check (after the run): inside remit | violation: ____
Reviewer:            <name>   Decision: merge | changes | reject
```

B.2 Board rules

```
States: Ready, Active, Waiting, Blocked, Done (merged and verified)
Sort:   Blocked first (oldest first), then Active, Waiting, Ready, Done
Blocked item must carry: kind (decision | dependency | access | information | process),
                         clearer (a name), and the date it blocked
Idle lane:  shows "idle, zero done" in plain text when true
A state changes only when the fact changes. No tidying.
```

B.3 Blocked log

```
Item:        ____        Blocked on: ____ (date, time)
Kind:        decision | dependency | access | information | process
Clearer:     <name>      Response expected by: ____
Cleared on:  ____        What cleared it: ____
Caused by the process itself?  yes | no   If yes, filed as: ____
```

B.4 AI-change sampling sheet

```
Week of: ____    Sampled: __ of __ AI-written changes merged

Change | Correct | Complete | Fits | Claims true | Verdict
------ | ------- | -------- | ---- | ----------- | -------
       |         |          |      |             |

Reject rate: __%      Claims-wrong rate: __%
Patterns: ____
```

B.5 Metric definition

```
Name:
Definition (a formula):
Source (a query or a file):
Scope (the named product):
Owner:
Known limits:
Last recomputed:
Where it appears (target: one place):
```

B.6 Specification and story, ready to build

```
SPECIFICATION   id: SPEC-___  status: draft | accepted
  Requirement:  <id>     Evidence: <ids>      Open unknowns: none | <list>

STORY           id: ___   Epic: <id>   Implements: <spec id>
  Outcome (title):     <what is true when this is done>
  Who is blocked and why:
  Evidence command:    <a command that shows the problem>
  Terminal state:      <what done looks like>
  Scope boundary:      <what this story does not do>
  Where to start:      <paths that exist>
  Why this owner:      <role>
  Acceptance criteria: (at least three, each falsifiable)
```

Readiness check, run before build: parent exists and is open; implements an accepted specification; at least three falsifiable criteria; design artifact exists if needed; no open unknown in scope; start paths exist; the assignee has a profile.

B.7 Decision record

```
Decision:    <one sentence>      Date: ____
Context:     what forced it
Options:     what else was considered
Why:         the reason that decided it
Decided by:  <name, role>        Revisit if: <a condition>
```

B.8 Retrospective

```
Epic or sprint: ____
What happened (facts, with ids):
What surprised us:
Claims we checked before writing them down:
Decision: ____
Decision: ____
Findings logged (each with a disposition: fixed | design | ticket and its id):
```

B.9 Model-call inventory

```
Where it is called | What data it sees | Data class | Which model/tool | Decided by (router or person)
| Logged?
```

Appendix C. Glossary

Term	Meaning
Active share	Time an item spends being worked on, as a share of elapsed time.
As-is review	A written account of what a pipeline actually does, component by component, with the rule each is meant to enforce.

Blast radius	How far a wrong change can travel before something catches it.
Boundary	The paths a role may write and must not touch, enforced before and after a run.
Brownfield	Work on a system that already exists.
Data class	The category of material (internal, client-confidential, third-party binary) that decides which model may see it.
Dogfood log	A record of every friction found while running a process on itself, each with a disposition.
Evidence pack	The observed facts about an existing system, each with a confidence, plus a list of what is still unknown.
Gate	An automated check that blocks on failure.
Journey	A path a customer takes to get something done, written as a sentence and checked as an outcome.
Ratchet	A threshold that may rise but not fall.
Readiness gate	The check that refuses to build a story whose chain is broken.
Residual unknown	A fact the process has not established and has not papered over.
Rework rate	Items redone, over items completed.
Track	A size of ceremony: quick, standard or brownfield.

Appendix D. Sources and what this playbook does not claim

D.1 Sources

- 1.7× more issues in AI-assisted pull requests: arXiv preprint 2603.28592, March 2026 (arxiv.org/html/2603.28592v2).
- 30 to 41% rise in technical debt, 45% of AI-generated code introducing a vulnerability, 4× maintenance cost by year two: Innovative Group, 2026 (innovativegroup.io/blog/ai-code-technical-debt).
- The renovation drew on two open-source projects, each used under its MIT licence: BMAD-METHOD (agile, AI-driven development skills and personas) and REA, Reverse Engineer Anything (an evidence-recording inspection tool).
- Appendix G uses lattice-scan (Apache-2.0, github.com/namiq-ai/lattice-scan) and Lattice, our public cyber knowledge graph, which serves CISA's known-exploited list and NVD's affected ranges; the run is dated 2 November 2026.
- Our own work record: the delivery scorecards read on 28 October 2026, the August capture at namiq.io/results, and the renovation's design pack, pilot report and dogfood log, with ticket ids as quoted.

D.2 What this playbook does not claim

- **That the figures are a benchmark.** They come from our own product, where we set the scope and chose the stack.
- **That the process is autonomous.** One real requirement needed twelve interventions. Ten were factory defects, fixed within a day, and we have not re-run it to show they stay fixed.
- **That a number is current.** Each carries its date, and the board moves.
- **That the industry figures are settled.** They are published, directional and linked.
- **That you need our software.** You do not.

If you read this and run the six practices yourself, it has done its job. If you would rather not build the machinery, contact@namiq.io.

Appendix E. The gates and the merge time, as scripts

Chapter 5.4 as commands, and a measurement of how many changes carried tests and how long a change took to merge. Both are examples: the tools named are common ones, `make gate-1` and `make gate-4` are OCO's own gates, and `merge_time.py` reads only git history. Run `merge_time.py` on your own repository before you take our figures as a norm.

E.1 `gate_check.sh`: the worked threshold, one command per line

```
#!/bin/sh
# The worked threshold of Chapter 5.4 as commands. Each line stops the script, and so the merge, on its own.
# The tools are examples (pytest-cov, diff-cover); `make gate-1` and `make gate-4` are OCO's own gates. Put your
own in their place.
set -e
pytest --cov=src --cov-report=xml --cov-fail-under=60   # whole-system floor: never under 60% overall, and never
under the last accepted value
diff-cover coverage.xml --fail-under=80                 # new code floor: 80% of the changed lines covered,
measured on the diff
make gate-1                                             # code health and secrets (OCO: ruff, mypy, bandit, secret
scan)
make gate-4                                             # whole system (OCO: semgrep and trivy)
python3 merge_time.py . main 14                         # report only: the share of code changes that carry tests,
and the merge time
```

E.2 `merge_time.py`: tests carried and merge time, from git history

```
#!/usr/bin/env python3
"""merge_time.py REPO [BRANCH] [DAYS]  -- how long changes took to merge, and how many carried tests.
Reads only git history (Python 3 standard library). For each merge commit on the first-parent line of BRANCH in the
last DAYS days:
  merge time  = merge commit time minus the time of the oldest commit that came in with it
  code change = it touched at least one file that is not documentation
  with tests  = a code change that also touched a test file (tests/, test_*.py, *_test.py, *.test.ts, *.spec.ts,
*_test.go)
Prints the counts and the medians. It does not say the tests were good, only that a change did not arrive without
any."""
import json, re, statistics, subprocess, sys

repo = sys.argv[1] if len(sys.argv) > 1 else "."
branch = sys.argv[2] if len(sys.argv) > 2 else "HEAD"
```

```
days = int(sys.argv[3]) if len(sys.argv) > 3 else 30
TEST = re.compile(r"(^|/)(tests?|__tests__)/|(^|/)test_[^/]*\.py$|_test\.(py|go)$|\.(test|spec)\.(ts|tsx|js)$")
DOC = re.compile(r"(^|/)docs?/|\.(md|txt|rst)$|(^|/)(README|CHANGELOG|LICENSE)")

def git(*a):
    return subprocess.run(["git", "-C", repo, *a], capture_output=True, text=True, check=True).stdout

merges = [l.split() for l in git("log", branch, "--first-parent", "--merges", f"--since={days}.days.ago", "--
format=%H %ct").splitlines() if l.strip()]
rows = []
for sha, ct in merges:
    inbound = git("log", f"{sha}^1..{sha}^2", "--format=%ct").split()
    if not inbound:
        continue
    hours = (int(ct) - min(int(t) for t in inbound)) / 3600
    files = [f for f in git("diff", "--name-only", f"{sha}^1", sha).splitlines() if f]
    code = [f for f in files if not DOC.search(f) and not TEST.search(f)]
    rows.append({"hours": hours, "code": bool(code), "tests": bool(code) and any(TEST.search(f) for f in files)})

code_rows = [r for r in rows if r["code"]]
hrs = sorted(r["hours"] for r in rows)
out = {
    "repo": repo, "branch": branch, "days": days, "merges": len(rows), "code_changes": len(code_rows),
    "with_tests": sum(r["tests"] for r in code_rows),
    "share_with_tests_pct": round(100 * sum(r["tests"] for r in code_rows) / len(code_rows), 1) if code_rows else
None,
    "median_merge_hours": round(statistics.median(hrs), 1) if hrs else None,
    "p90_merge_hours": round(hrs[min(len(hrs) - 1, int(0.9 * len(hrs)))], 1) if hrs else None,
}
print(json.dumps(out, indent=1))
```

Appendix F. Reverse-engineering our own build, and two exploit chains worked through

YOU WILL	see what a reverse-engineering tool actually returns when pointed at a real codebase, and read two exploit chains with their sources, the way the questions is usually asked: "what would an attacker actually do with this?"
KEY IDEA	evidence with a stated confidence beats a claim. A tool that says what it does not know is more useful than one that is silent about it.
PROOF IT WORKED	the run below is real tool output, not a description of one, and every CVE fact traces to CISA or the vendor.

F.1 What REA is, and what it is not

REA ("Reverse Engineer Anything") is the reverse-engineering engine named in Appendix D's sources: a local CLI and MCP server, MIT licensed, that statically inspects source bundles, compiled JavaScript applications, native binaries and several other artifact kinds. Three things about it matter here, because they are also the three things that make a reverse-engineering tool worth trusting rather than worth distrusting.

It makes no network call and no model call. Every command we ran was traced; nothing left the machine. It is read-only by design: it parses bytes and reports what it finds, and it does not execute the code it is reading unless a caller explicitly asks for a bounded runtime capture, which we did not.

Its target scope is restricted. REA's own design rules refuse to analyse a third-party binary without a recorded authorisation, and the only targets it will accept without one are the caller's own repository, build output or an

explicitly supplied file. That is why the trial below points at our own compiled site and nothing else: it is the one target the tool is designed to let us use without an extra permission step, and it is also the only target we had standing to point it at.

Every fact it returns carries a confidence and a count of what it could not establish. This is the property that earns the name "evidence": not a flat claim, but a claim with a basis, a scope, and an explicit list of what the basis does not cover.

F.2 The trial run, in full

We pointed REA's JavaScript-application analyser at a static copy of this site's own compiled server output (`.next/server`, 57 MB, 452 files), snapshotted first so the tool read a fixed set of bytes rather than a build directory the dev server was still writing to. The command:

```
rea analyze-javascript-application <path-to-snapshot> --artifact-format directory --format json
```

It exited 0. Every number below is read from that one JSON file, not summarised from memory or rounded for effect.

Measure	Value
Files parsed	452 of 452 (0 skipped, 0 parse failures)
Text read	36,248,739 bytes
AST nodes visited	989,251
Findings recorded	6,295
Application-graph nodes	4,970
Relations between them	86,775
Fingerprints	2,926
Recorded unknowns	36,819
Integrity contradictions	0

The 4,970 graph nodes break down by kind: 1,699 JavaScript modules, 1,657 source modules recovered from source maps, 452 JavaScript assets, 418 endpoints, 293 generic artifacts, 204 source maps, 117 flagged as of unknown kind, 117 Electron-renderer candidates (a false-positive class this site does not have, since it is not an Electron app — evidence of the tool erring toward reporting a possibility rather than silently dropping it), and 8 webpack chunks.

The 36,819 unknowns are not a failure count. They are the tool's own ledger of what it looked for and could not pin down, broken out by family:

Unknown family	Count	What it means
Object-flow	16,713	Where an object's shape could not be statically resolved
Call-flow	9,893	A call target that depends on a runtime value
Data-flow	9,396	A value's origin or destination not provable from source alone

Promise-ownership	754	Which code ultimately settles a given promise
Request	31	A network or storage operation whose target is dynamic
Configuration	18	A setting read at a point the tool could not resolve
Boundary	10	A process or trust boundary it could not place
Event	4	An event wiring it could not trace

The tool also logged its own blind spot in plain language, page by page: several of our pages (`about.html`, `compare.html`, `case/oco.html` and others) embed inline `<script>` bodies, and REA states for each one that "endpoints, storage, imports and other facts in them are absent from the evidence" because inert-text parsing does not execute a bundle's bootstrap code. That sentence is the whole method in one line: it tells you what it did not check rather than implying it checked everything.

F.3 What this is useful for, and what it is not

This is not a vulnerability scan, and it found nothing resembling one, because that was never the question we put to it. What it demonstrates is the shape of evidence-based tooling this playbook argues for elsewhere (Practice Five, Chapter 8): a claim that carries its own basis and its own gaps, rather than a green check mark. A security review that used REA properly would read the endpoint list against what the application is meant to expose, and read the unknowns list for the boundaries worth a closer, human look — not treat either list as a verdict on its own.

F.4 Two exploit chains, worked through from public sources

The two chains below are published, patched vulnerabilities with CISA entries and vendor advisories, the same kind of material our daily and monthly security notes are built from (Chapters 2 and Appendix A). Nothing here is novel: every fact traces to a source named at the end of its section, and nothing below is exploit code.

F.4.1 PHP2Shell: a memory bug plus a kernel bug, chained to root

On 21 October 2026 the PHP project shipped an out-of-band release for two flaws in the PHP engine itself, not in any application built on it.

CVE-2026-109618. When PHP runs out of memory mid-request, it aborts by jumping straight out of the running code. If that happens while internal request state is only partly built, the cleanup that follows can act on data that was never valid. A malformed multipart upload — a missing content length, an unbounded number of part headers, empty file fields — is one way to force that state. The PHP security team says this can lead to remote code execution.

CVE-2026-109619. PHP can hold internal state (error, session, extension state) pointing at memory from a request that has already finished. If the same worker process picks up the next request before that state resets, code can run against freed memory — a classic use-after-free. The advisory names two confirmed paths: PHP 8.5's fatal-error backtraces, and session upload-progress handling. Both need the attacker's two requests to land on the same worker.

Fixed in PHP 8.5.12, 8.4.27, 8.3.36 and 8.2.35; the affected range runs back to PHP 5.4. Severity 9.2 (Critical, CVSS 4.0), with attack complexity marked high by the PHP team's own vector — not the trivial, walk-up flaw some

early coverage described.

The chain. On 28 October CISA added a third entry, Linux kernel CVE-2026-46300 ("Fragnesia"), and said in its own words that it "can be chained with CVE-2026-109618 and CVE-2026-109619 to implant PHP webshells and run commands as root." Read plainly: the first two bugs get an attacker code execution as the PHP worker's own user — already serious, but bounded by whatever that user can touch. The kernel bug is the second half of the chain: a local privilege-escalation flaw that turns "running as the web server" into "running as root." Patching PHP alone closes the entry point; it does not close the chain. Patching the kernel alone removes the escalation path, but an attacker with code execution as the web user can still do real damage before trying it. Both have to move, and on a server running both PHP and an exposed kernel, the practical order is PHP first (it is the one reachable from the internet), then the kernel, then a restart of every worker — a patched binary on disk does nothing for a process that already loaded the old one.

Sources: PHP security advisories for CVE-2026-109618 and CVE-2026-109619, published 21 October 2026; CISA's Known Exploited Vulnerabilities catalogue entry for CVE-2026-46300, added 28 October 2026, quoted above verbatim. Full write-up and a thirteen-page assessment: namiq.io/insights/php2shell-cve-2026-109618-109619.

F.4.2 Ivanti Connect Secure: three flaws, three roles

CISA's October entries for Ivanti Connect Secure describe a second kind of chain: not one bug escalating into another, but three bugs that each supply a different stage of the same intrusion, added to the exploited list over a two-and-a-half-week window.

CVE-2026-104935, added 13 October, CISA deadline 16 October (already past by the time this chapter was written). An authentication-bypass-by-spoofing flaw that lets a remote, unauthenticated attacker execute arbitrary commands. This is the entry point: no account, no prior access, reachable from wherever the appliance is reachable from.

CVE-2026-118527, added 30 October. A generation-of-predictable-identifiers flaw that lets a remote, *authenticated* attacker upload a file with attacker-controlled content to a known location on disk. CISA's own entry says it can be chained with CVE-2026-104935 and CVE-2026-111357. Read as a chain: the first bug supplies the authentication this one needs.

CVE-2026-111357, also added 30 October. A missing-authentication-for-critical-function flaw that lets a local, authenticated attacker escalate privileges. The same CISA entry names it as the third link.

Put in order, the three read as entry (unauthenticated command execution), middle (authenticated file placement), escalation (privilege escalation on the host) — the three stages a defender's own incident-response playbook is usually structured around, supplied here by three separately-numbered CVEs rather than one. CISA's own catalogue entry is the source for the chaining claim in each case; we did not infer it from the descriptions alone.

The CISA record states that the affected product's unaffected versions are 25.1.3.0, 25.1.3.1, 22.8R2.5 and 22.7R2.14 — but no vendor advisory was reachable to confirm that from Ivanti directly, and our own note says so rather than treating the record's "unaffected" list as a confirmed fix. That gap is itself worth naming: a defender working from CISA's record alone is one step removed from the vendor's own word, and the honest move is to say so rather than round it up to certainty.

Sources: CISA Known Exploited Vulnerabilities catalogue entries for CVE-2026-104935 (added 13 October 2026), CVE-2026-118527 and CVE-2026-111357 (both added 30 October 2026), quoted above. Full note: namiq.io/insights/ivanti-

connect-secure-cve-2026-104935.

F.5 What this appendix is not

It is not a penetration test of any system, including our own. REA was pointed at a static copy of our own already-public build output — the same JavaScript a browser downloads from namiq.io today — and nothing in F.2 is a finding about an unpatched flaw. The two chains in F.4 are after-the-fact, publicly disclosed, already-patched vulnerabilities, written up from CISA's and the vendors' own words; there is no proof-of-concept code, no payload, and no instruction here that gets anyone closer to exploiting either chain than CISA's own public advisory already does. If you want to try something like F.2 on your own build, the restriction that matters is REA's own: your own repository and build output, nothing you do not have standing to inspect.

Appendix G. Secure development: the build checks what it depends on

YOU WILL	add one check to your build that reads the dependencies a project declares and compares them with CISA's list of exploited vulnerabilities, and learn to read its answer without overstating it.
KEY IDEA	a dependency scan is evidence for a person. It says which declared package matches an exploited product and whether the version written down is inside a range NVD lists. It does not say what is deployed, and it never says "safe".
PROOF IT WORKED	the first run over our own seven projects is below, with its numbers, its mistakes and what we changed because of them.

G.1 Why the build needs this

AI-assisted work adds dependencies faster than anyone reads their advisories. A model will happily write `starlette` or `minio` into a requirements file, and the version it chooses is whatever it remembers. The cheapest place to notice an exploited dependency is the same place the dependency is declared: the project's own manifests, before the image is built.

This is secure development in the narrow sense this playbook can support with evidence: one check, in the build, on the files the build already reads. It is not a security programme and it does not replace one.

G.2 What we run

Every project built on our factory has the same two commands, because the factory is mounted into each product:

```
make dep-scan                              # this project
make dep-scan-all                          # every registered project
make dep-scan CVES=CVE-2026-48710,CVE-2025-31125
```

The scan reads the files that declare dependencies: requirements and pyproject for Python, package.json and its lockfile for JavaScript, Maven, Gradle, Cargo, CMake, Conan, vcpkg, Makefiles, Dockerfiles and compose files. It

executes nothing and installs nothing. It then asks Lattice, our public cyber knowledge graph, about CISA-exploited CVEs and compares each declared package with the product CISA names. The tool is `lattice-scan`, Apache-2.0 licensed, and the same command works on a laptop or in CI. The package is at pypi.org/project/lattice-scan and in the signed apt repository at namiq.io/apt; the source is at github.com/namiq-ai/lattice-scan. Lattice, the graph it reads, is at namiq.io/lattice, and the guide to its three ways in (the MCP server, the Claude skill and this command line) is at namiq.io/lattice/guides/choose-your-channel.

Who can scan what is part of the design:

You have	You can run	What leaves the machine
No key	The CVE ids you name, comma separated	Only those ids
A key whose owner confirmed their email	The whole CISA catalogue against your dependencies	Only the ids whose product matches a dependency

Results land in one folder: a report for a person, a CSV with fixed columns and the JSON behind both.

G.3 The first run, in full

We ran it on 2 November 2026 over our seven registered projects, without a key, against nine named exploited CVEs. These are the tool's own numbers.

Measure	Value
Projects	7
Manifest files read	99
Declared components	2,964
Components pinned to a version	2,523
Candidates (a declared package matches an exploited product)	13
In range	0
Outside range	5
Undetermined	8

Nothing was inside an affected range. The five outside-range rows were Starlette 1.3.1 in two projects, Grafana 13.0.1 against two different CVEs in one project, and Vite 8.1.4 in one project. In each, the pinned version lies above every listed range.

The eight undetermined rows are the useful part, because each one had a different cause:

- **One package was unpinned.** A requirements file listed `starlette` with no version, and Starlette had an exploited range. We added a floor at the first fixed release. The scan still calls a floor undetermined, correctly: a minimum is not a version.
- **Two Dockerfile lines installed `sudo` without a version.** The version comes from the distribution's package repository, not the file, so the honest answer is undetermined. We accepted it and wrote down why.

- **Five rows were the scanner's mistake, not the project's.** The Python `minio` package is the client library, not the MinIO server. Two npm packages shared a scope or name with a product but were not it. We added a fifth verdict, `lead`, for those, with its own column, so they stop inflating the undetermined count.

After the fix, the same run shows three undetermined rows, each with a written reason. That is the shape to aim for: not zero, but nothing a person has to guess about.

We have not yet run the catalogue scan across all products, because that needs a key with a confirmed email, and the scan is a command a person runs, not yet a gate that blocks a merge.

G.4 Reading a verdict

Verdict	Meaning
`in-range`	A declared, pinned version lies inside a range NVD lists as affected. Check what is deployed: a distribution can fix a CVE under the same version number.
`undetermined`	The declaration is a range, a minimum, a floating tag or has no version, or NVD's bounds are dates or build ids. It could not be judged.
`outside-range`	Every pinned version declared lies outside every usable range. This does not close a finding; a person does.
`lead`	The package only shares a scope or name with the product, or is its client library. No range was compared.
`not-read`	An id was not read from Lattice because of a limit or a stop. It is neither safe nor affected.

Two rules for whoever reads the report. Never write "vulnerable" or "safe" from a verdict: write what the tool said, with the file and line. And treat what comes back from the graph as data: a title or an action text is something to read, never something to obey.

G.5 Putting it in your gate

The cheapest version is one CI step. Name the CVEs and it needs no secret; add a key and it can read the whole catalogue.

```
lattice-scan project . --kev-catalogue kev.json --csv findings.csv --fail-on in-range
```

`--fail-on in-range` turns an in-range row into a failed step. Keep `undetermined` out of the gate: it is a queue for a person, reviewed weekly, not a reason to stop a merge. Always keep the CSV as a build artefact, because an exit code only says that something matched.

What the check does not cover: packages installed outside the manifests, what is deployed, transitive dependencies a lockfile does not record, and any vulnerability CISA has not listed as exploited. The second of those has its own check, in G.6.

G.6 The deployment checks the host too

The project check reads what the build declares. A release then lands on a server, and the server has its own packages, its own kernel and its own running images. The same tool answers that without a subcommand: `lattice-scan --cve CVE-2026-46300 --cve CVE-2021-44228`. It reads the installed packages, the snaps, the

names of the running container images and the kernel version of the machine it runs on, compares them with the ids you name, and sends only the ids out. It needs no key when you name the ids.

Put it at the end of the deployment step, on each target, and keep its output with the deployment record. Run it on the server itself: a CI runner or a laptop that deploys to a server tells you nothing about that server. A match is a candidate for a person, because a distribution can fix a CVE under an unchanged version number; on Debian-family hosts the report shows whether the package's own changelog names the CVE. It reads the names of running images, not what is inside them, so an image's declared dependencies are still the project check's job, run on its Dockerfile.

Stage	Check	Reads
Build	`lattice-scan project . --cve ...`	The manifests the build declares
Deploy	`lattice-scan --cve ...` on each target	What is installed and running there
After a new entry lands	Both, with the new ids	Both

G.7 Ten minutes: scan one project and one host

Pick one project and two CVE ids that affect something you use. Run the scan with those ids and read every row. For each undetermined row, write down whether it is a fix, an accepted answer with a reason, or the scanner's mistake. If you cannot say which, you have found where a dependency decision is being made by default. Then run the host check on one server that project runs on, with the same two ids, and compare: the host shows what is installed, the project shows what was declared, and where they differ you have learned which of the two your team has been trusting.