// WHITE PAPER · PUBLISHED 9 AUGUST 2026

The Scaffolding Comes First

What has to exist before AI coding licenses produce code worth shipping

Jason Newell · MAX Research Collective · Report WPR-CLD-001 · Published 9 August 2026
Licensed under the Creative Commons Attribution 4.0 International License (CC BY 4.0),
https://creativecommons.org/licenses/by/4.0/
Original: https://www.maxresearchcollective.com/articles/the-scaffolding-comes-first.html
Cite as: Newell, J. (2026). The Scaffolding Comes First. MAX Research Collective, WPR-CLD-001.
https://www.maxresearchcollective.com/articles/the-scaffolding-comes-first.html

Part I, Why the pipeline became the constraint

What the volume does

Predicted shard imbalance went from 2.38 times to 1.00 after a careful rebalance, which is a textbook result. Measured wall-clock went the wrong way, from 12.3 minutes to 13.8.

Every individual step had been done correctly. The durations file the rebalance trusted couldn't see fixture setup, module import time, or a subprocess spawned and then joined, and in a real suite those three dominate, which means the whole exercise optimized the distribution of a number that was itself wrong. Profiling what the tests actually did found what no duration record could report. Twenty tests held 56.5 percent of a suite that ran 1,563 seconds. One of them recursively spawned the entire unit suite as a subprocess and then blocked waiting on the orphan, at 300 seconds, or 19 percent of the whole run on its own. Four more each cold-started the same expensive model fit at 170 seconds apiece. No splitting algorithm reaches any of that.

That week is the cheapest failure in this record. It happened inside a pipeline built to surface it. The expensive ones happen somewhere else entirely: inside a pipeline still sized for the rate of work it had two years ago, where nothing is instrumented well

enough to notice that the assumptions underneath it stopped holding some time last quarter.

An AI coding agent doesn't make a pull request cheaper. It makes many more of them. Nothing about the unit changes: a branch, a diff, tests, a description, a review, a merge. Only the rate does. This record runs at roughly 280 code pull requests a month from one developer, about fourteen on a working day, against a human baseline of one to three. Call it a multiple of five to ten, depending on where you put the baseline. That volume is unusual today. It's also a fair preview of a normal team eighteen months after adopting the same tools, which is why the failure modes generalize even where the multiples don't.

Four things follow from the rate alone. Review capacity becomes the bottleneck rather than authorship, since nobody reads fourteen pull requests a day and stays useful. Review either gets partly automated or it degrades into rubber-stamping. There's no third outcome. Anything priced per unit of work reprices itself; no vendor had to raise anything, because the account simply started buying five to ten times as many units. A wasteful habit that costs nothing at two pull requests a day costs real money at fourteen. And one spread widens enormously: the distance between the cheapest place to catch a defect and the most expensive one.

That last point organizes everything below, and it's the one most teams underrate. The pipeline's job stopped being to find bugs. Its job is to find each bug at the cheapest stage that could have found it.

Cost of catch

The same defect costs wildly different amounts depending on where it surfaces. Six layers sit between an idea and production, and each one prices a finding differently.

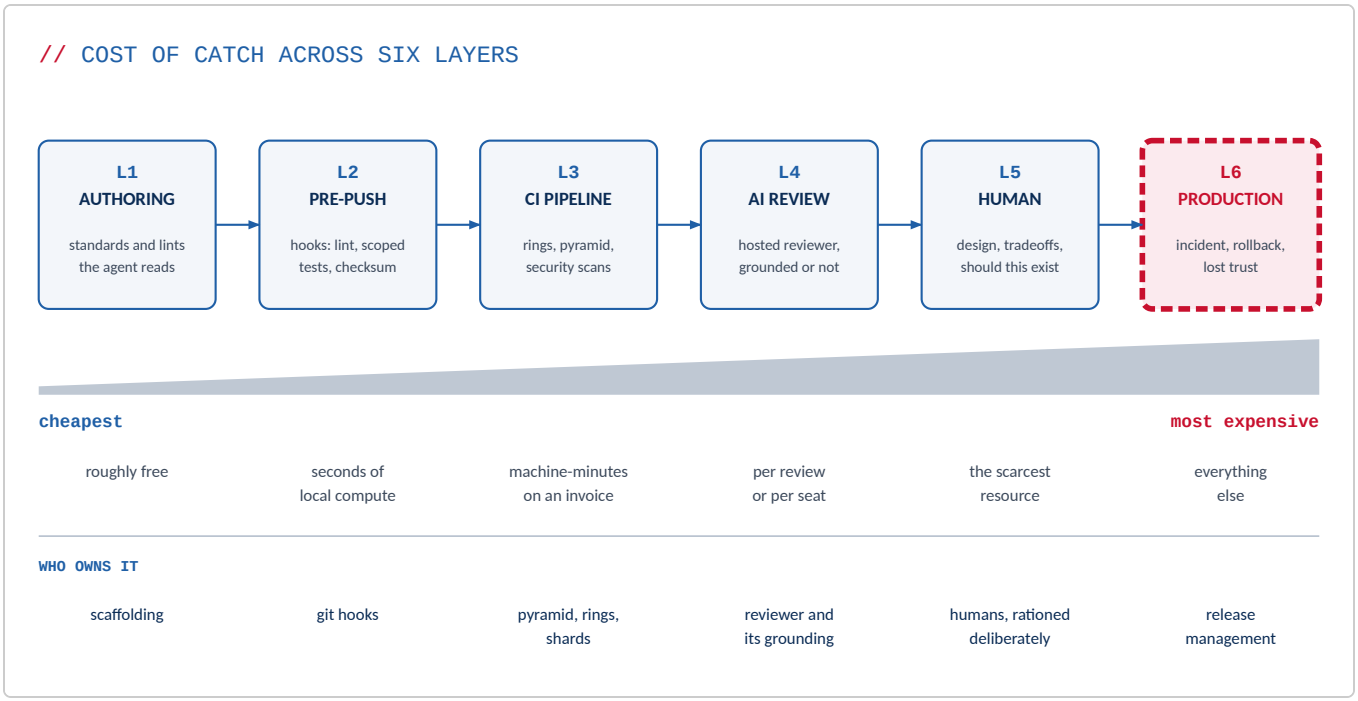


Figure 1. The six layers a defect can surface in, and what a catch costs at each one.

At the left end sits authoring time. The agent reads the house standards before writing a line, and a catch there costs roughly nothing. Next is the local pre-push hook, spending seconds of a developer's own machine. Then continuous integration, the automated build and test run that fires on every push. It spends machine-minutes, and machine-minutes land on an invoice. Then the AI reviewer, spending a per-review charge or a seat. Then the human reviewer, spending the scarcest resource in the building. Then production, which spends everything else.

Three rules fall out of that picture. A finding at the AI review layer that a lint could have prevented is a leak, and at 280 pull requests a month it's an expensive one. Layers don't substitute for each other, so a reviewer finding a great many things is no evidence that the system works; it may instead be evidence that the layers to its left are underbuilt and that the scarcest resource in the building is being spent on the cheapest class of finding available.

Direction of travel is the real metric. A healthy pipeline shows findings migrating leftward over months. A reviewer still catching the same class of thing in month six is attached to a flywheel that isn't turning. Teams measure how much their pipeline catches and feel good about a large number. The useful measurement is where it caught it.

Twelve weeks of an unprepared rollout

Buying seats is the easy part, and it's the part that gets approved. The scaffolding that makes those seats produce work worth shipping has no line item, no vendor, and usually no owner.

Hand a team AI coding licenses without the layers below and the sequence is predictable enough to write down in advance. Throughput jumps in week one and everyone is pleased. By about week three, review is the bottleneck and rubber-stamping starts, quietly, because the alternative is telling a colleague that their fourteenth pull request of the day waits until tomorrow. Around week five the pipeline minutes run out mid-month, and a gate gets skipped once, with permission, as an exception. Week seven brings the first escaped defect nobody can explain, and trust drops faster than it rose. Week nine brings an invoice at a per-unit price nobody approved. By week twelve the guardrails either get built retroactively, under incident pressure, at the worst possible moment anyone could choose for designing a control system, or the tooling gets blamed and pulled.

None of that is a tooling failure. Every step is a delivery-system failure that the tooling revealed by going faster, on a schedule set by whoever signed the purchase order.

Part II, Scaffolding

What scaffolding is

Scaffolding is the written, machine-readable context an AI tool reads before it does anything. In practice it's a small set of files living in the repository beside the code.

It matters because an ungrounded reviewer or agent is a different and worse product than a grounded one running under the same brand name, sold at the same price, and evaluated in most published comparisons as though the two were interchangeable.

Five artifacts carry it. A standards file holds the house rules in checkable form, phrased so a machine can apply them: assert behavior rather than source text; keep

synchronous work off the event loop. A defect registry is append-only, recording every bug class that shipped and hurt, with the symptom, the root cause, and the check that would have caught it. Per-directory rules scope narrow guidance to a path, so backend rules govern backend code. Structural lints are executable pattern rules matching on code structure rather than on text, and they fire on every diff. Agent operating rules state what the agent may do unattended, what needs a human, and what's forbidden outright.

The grounding effect is measurable. On a controlled first pass here, an ungrounded reviewer produced two false positives out of three findings. Every false positive costs a triage cycle and a reply, at full price. Two thirds of the output arriving as noise is a different product.

Scaffolding isn't free. Treating it as free is the commonest budgeting error in this area. The set in this build runs to a standards file of roughly 700 lines, a defect registry, per-directory rules, five structural lints, and a reviewer configuration of about 200 lines. All of it is authored and maintained by the agent; it gets paid for inside the agent's own line item rather than anywhere a finance team would think to look. A tool evaluated out of the box against a deeply scaffolded incumbent isn't being compared fairly. Most published comparisons are exactly that.

The flywheel

Standards derived from real defects get followed. Imported best-practice lists get ignored by humans and diluted by machines, because nobody in the room remembers what any particular line was protecting against.

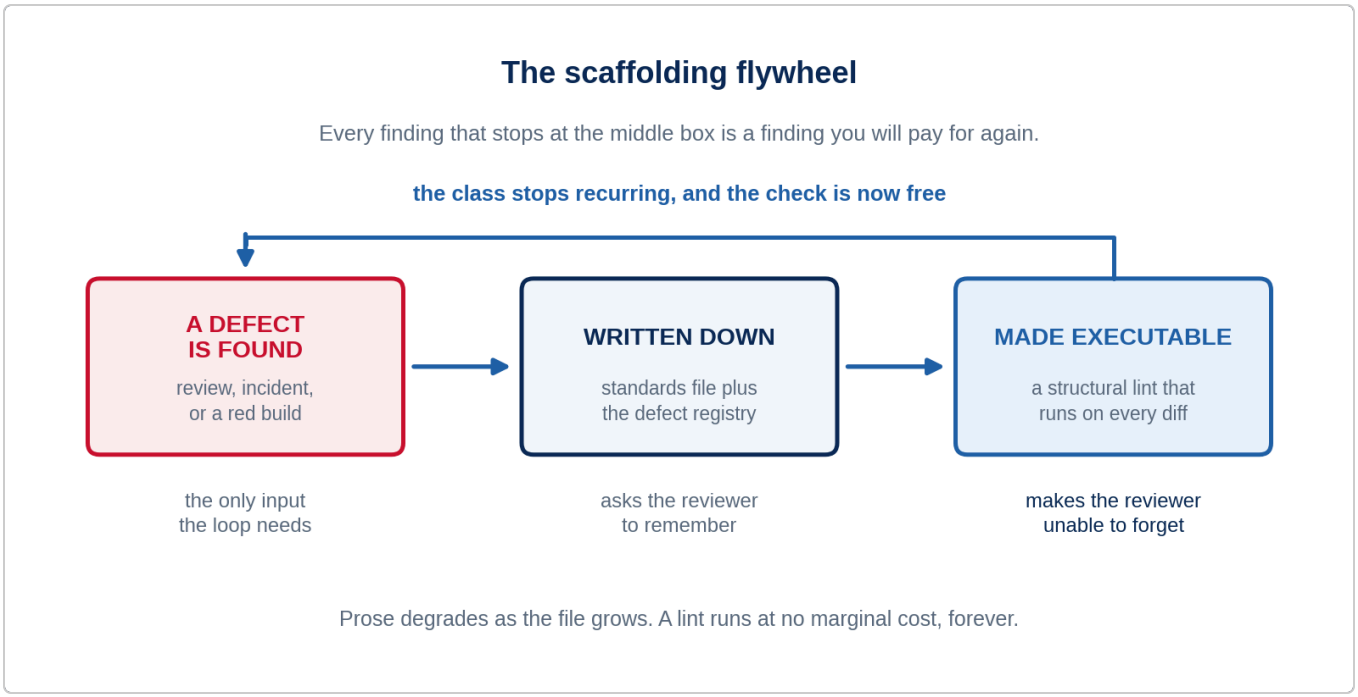


Figure 2. The three states a finding passes through, and the loop that closes when it becomes a lint.

A finding moves through three states. Everything valuable sits in the gap between the second and the third.

A defect gets found. Somebody writes it into the standards file and the registry, which asks the reviewer to remember it. Then somebody promotes it into an executable structural lint, and now the reviewer can't forget it. Prose in a standards file works imperfectly. It degrades as the file grows, because a rule nobody can recall the reason for is a rule that quietly stops being applied. A lint runs on every diff at no marginal cost, forever. The class stops recurring.

A reviewer that accepts executable rules is a ratchet: a check that tolerates the violations already in the tree and blocks new ones. A reviewer that accepts only prose is a reviewer. When comparing two tools, that single difference outweighs any findings-count on a comparison page, because one of them compounds into a permanent guard while the other keeps producing the same advice at the same price for as long as you keep paying for it. It is almost never on the comparison page.

The failure mode is a standards file that grows without promotion. Two thousand lines of prose that nothing executes is a document. Growth is healthy only when

recurring items graduate into lints, which is also the only mechanism by which a review layer gets cheaper over time instead of steadily more expensive.

The guardrail that parsed and did nothing

A guideline entry in the reviewer configuration turned out to be directory-scoped. An entry sitting under one folder governed only that folder, so rules written for an entirely different part of the tree never applied to anything at all.

The configuration parsed cleanly. The schema check passed. The rule reported nothing, because it did nothing. Validating that a configuration parses is not validating that it works, and the second claim was never actually made here.

Two lessons came out of it, and the second is the general one. Configuration has no symptom when it's wrong. A misconfigured guardrail and an absent guardrail look identical from outside; both of them are silence. The only real test is behavioral: change something the rule should catch, then confirm it gets caught. That's an observation. Everything short of it is a validation, and validations answer a smaller question than they appear to answer.

This surfaced only because two independent reviewers flagged it within half a minute of each other, which is the kind of corroboration that costs nothing to act on and cannot be manufactured after the fact by anyone reading the same configuration file more carefully. On one lane, the gap would still be there. It would also still be invisible.

Part III, Continuous integration

What exists and what runs

Tests come in layers. Many tiny fast ones, fewer medium ones, and a handful of slow realistic ones, where each layer catches a class of defect the others can't and costs more than the layer beneath it. That's the quality pyramid. It describes what exists in the repository.

What runs on a given change is a separate question. Conflating the two is where most test-strategy arguments actually live, and the argument tends to go badly because both sides are right about the thing they're actually talking about while disagreeing about scheduling under the impression that they're disagreeing about quality.

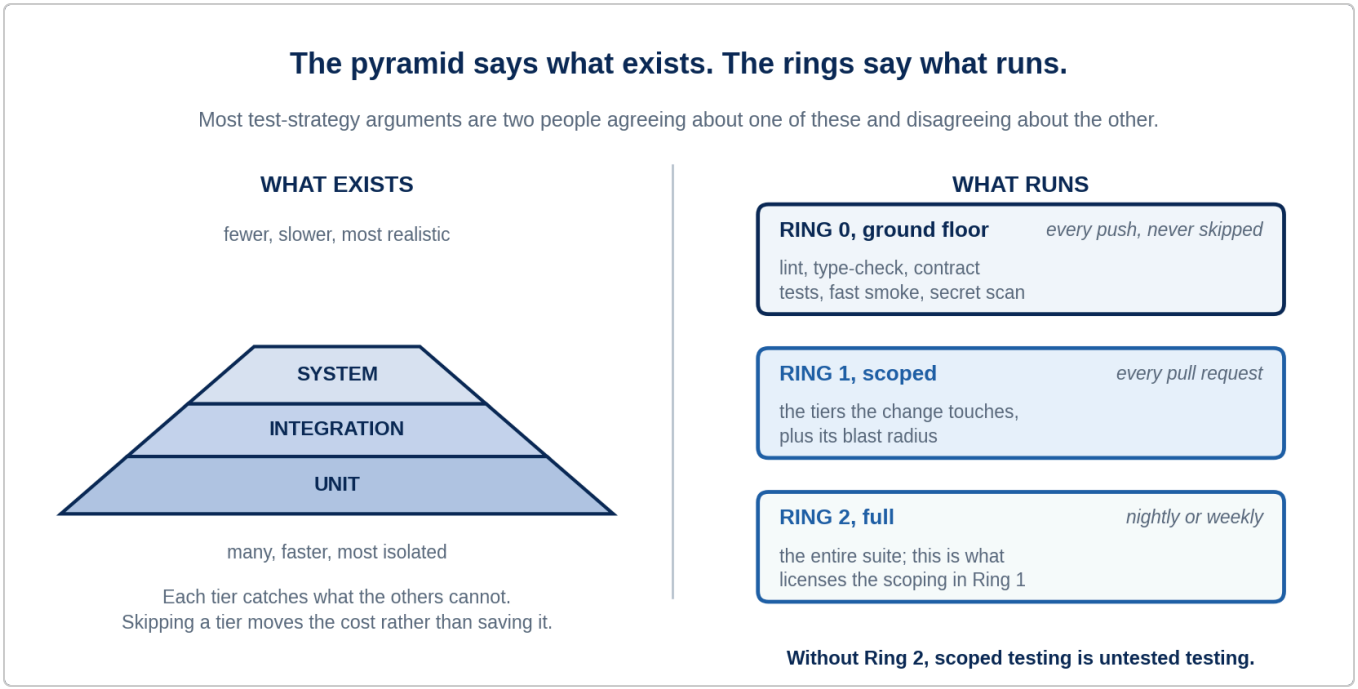


Figure 3. The tiers that exist on the left, the rings that decide what runs on the right.

Three rings answer the second question, and they're independent of the pyramid. Ring 0 fires on every push and is never skipped: lint, type-check, contract tests, and a fast smoke set. Ring 1 fires on every pull request on top of Ring 0, covering the parts of the pyramid the change touches plus its blast radius, meaning whatever else that component can plausibly break. Ring 2 runs the entire suite on a fixed cadence, nightly or weekly.

Ring 2 licenses Ring 1. Without a periodic full run, scoped testing is untested testing. Nothing exists by which anyone would discover that the scoping rule has been wrong for three weeks. With Ring 2 in place, a scoping miss has a bounded lifetime.

Two things deserve to be absolute rather than negotiable. Ring 0 should be short enough that nobody ever wants to bypass it, and hard enough that nobody can.

Auto-fixable failures deserve their own class as well. Some red builds are purely mechanical. A checksum drifted, or a generated file went stale, and the right response is to regenerate and merge rather than to diagnose and re-run. Telling mechanical drift apart from behavioral regression saves a full cycle every time. The docs-only fast path here exists so the heavy pyramid never re-runs for a checksum.

Green by vacancy

A feature-by-feature map of this codebase turned up a component with source files and zero system tests. It had been green the whole time, green by vacancy, and a tier that never runs anything looks identical on a dashboard to a tier that passes.

Well tested and well covered are different claims. Coverage measures the wrong one. A coverage percentage says which lines executed. Whether the assertion afterward would fail if the behavior broke is a question it never asks.

The sharpest version here is a class of test that asserted on source text, checking that a file contained a particular string, rather than on behavior. Those tests pass forever. They survive any refactor that preserves the string. They catch nothing. They count fully toward coverage. The rule that came out of it is short: a test that can't fail when the behavior breaks isn't a test, whatever the coverage tool says about it.

Two other habits belong here. Growing a suite without ever profiling it is where the real money hides, which the opening already paid for. Treating end-to-end tests as the safety net inverts the pyramid, since they're the slowest, most expensive, and flakiest tier in it. They should be a thin confirmation that the wiring is real.

Shards buy time, not money

A shard is a slice of the test suite running on its own parallel machine. One commercial detail surprises people often enough to state early. Sharding cuts wall-clock. It doesn't cut cost. Most pipeline billing sums per-machine minutes, so four machines for ten minutes bills the same as one machine for forty, which means

parallelism buys speed with money that was already being spent and saves nobody anything.

There are two ways to cut a suite. The tooling defaults to the less useful one.

Table 1. Two ways to cut a test suite, and what a red shard tells you under each.

	HORIZONTAL, BY DURATION	VERTICAL, BY FEATURE
What a red shard says	Some quarter of the tests failed.	The capture feature is broken.
Blast radius	Re-derived by a person on every red build.	An explicit artifact, computed once.
Change scoping	Not possible, since slices carry no meaning.	Natural, since a slice is a feature.
What it optimizes	A number that is easy to measure.	The next action after a failure.
Why teams pick it	It is what the tooling does by default.	It has to be built deliberately.

Table 1 is the complete data set for what it shows, so it takes no appendix.

Horizontal slicing splits by duration, which is what a default configuration does, because duration is easy to measure. A red shard then tells you that some quarter of the tests failed. Somebody opens the log, works out which feature the failure belongs to, works out what else that feature touches, and decides what to re-run. That happens on every red build, and it's unpaid, invisible work every single time. Vertical slicing splits by feature instead, so a red shard says the capture feature is broken and its blast radius is these three others. The failure is identical either way. What changes is who does the mapping: a person on every red build, or a computed artifact once.

The sequencing rule came out of the rebalance that made things slower, and it's the transferable part. Kill the outliers first. Re-slicing a suite whose cost sits in twenty

tests just moves those twenty tests between slices. More generally: model the wall clock rather than a proxy for it. Any optimization driven by a proxy will optimize the proxy.

Some suites are genuinely expensive and genuinely rarely relevant. A full model retrain, a hardware-in-the-loop run, and a long soak all qualify. Those belong in specialty shards firing on a trigger, and both guardrails around them were learned here by nearly getting them wrong, which is the usual way a guardrail gets written. The skip has to be loud: a specialty shard that silently doesn't run is deleted coverage wearing a costume. There also has to be a maximum staleness. "Runs when relevant" degrades into "never ran" inside about two months unless something forces it.

Gates and suggestions

Two rules can look identical in a configuration file and differ by the entire cost of the operation.

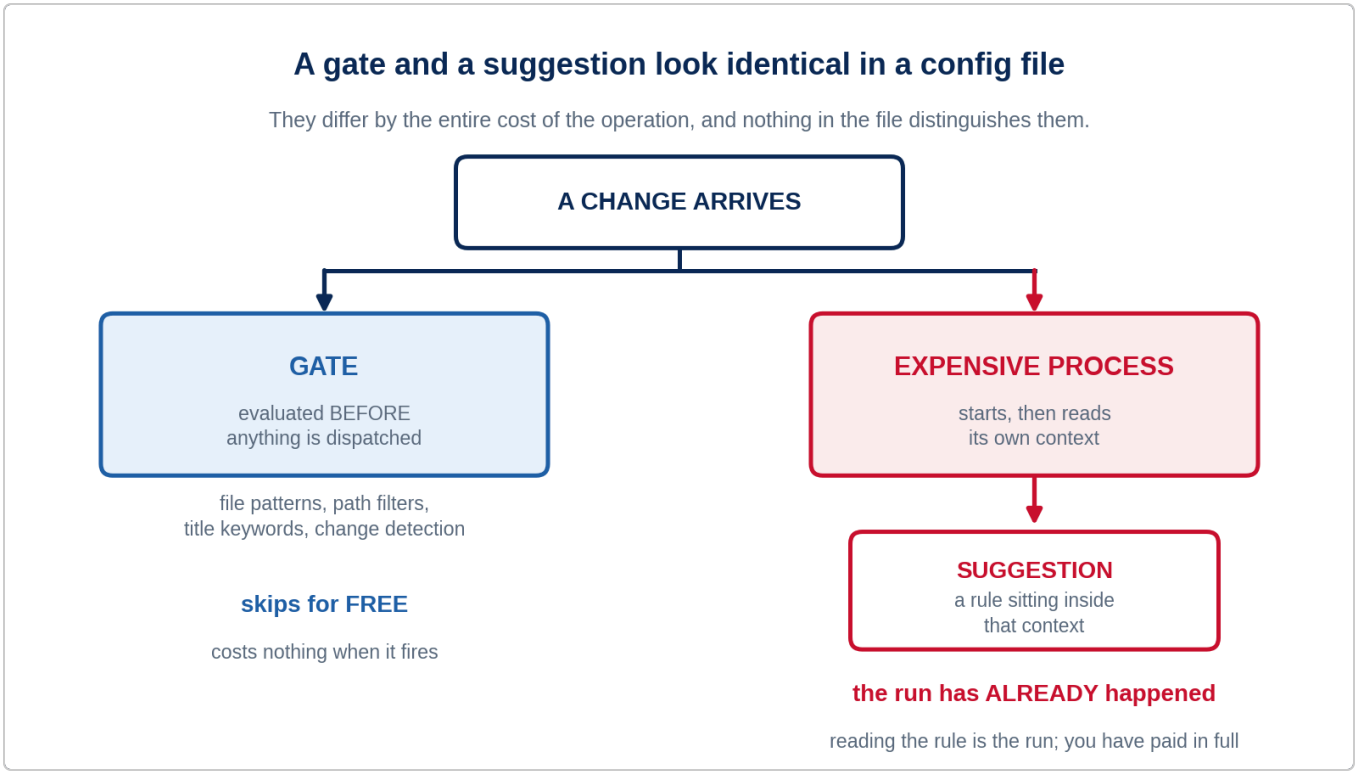


Figure 4. Where each rule sits relative to dispatch, which is what decides whether it can save anything.

A gate is evaluated before expensive work is dispatched, so it costs nothing when it fires. File patterns, path filters, title keywords, and change detection are gates. A suggestion is a rule the expensive process reads while it's already running, which means reading the instruction is the run. A suggestion can improve an outcome. It can never save a cent.

Both were running here without anybody noticing the difference. A rule meant to skip documentation changes was written as guidance inside the reviewer's context, where it couldn't possibly save anything, while the same tool was skipping those same changes for free through a title-keyword match evaluated before dispatch. The buyer question is which rules are gates and which are suggestions. Whether the skip rules fire says nothing about it, and the configuration file won't say either.

The guardrails worth building are mostly cheap, and each one carries a specific way of failing quietly. Change detection and fast paths let documentation-only changes skip the heavy pipeline, and the rule has to name every file such a change touches; the version here excluded docs but not the checksum file that every docs change regenerates, so the condition was never satisfiable. Dependency caching turns a 22-minute install into about 30 seconds, a 40-fold difference, so the hit rate wants a target above 95 percent and an alarm below it. A silent cache miss looks like a slow day.

Concurrency keys cancel the in-flight run when a new push lands on the same branch, which is necessary, and which also means pushing mid-review discards the review, cancels the run, and on a metered reviewer bills the charge anyway. Pre-push hooks run the cheap front half locally before spending a machine minute, and they have to stay fast, since a slow hook gets bypassed and a bypassed hook is worse than none because you believe it ran. Baseline ratchets permit existing violations and block new ones, which is the only realistic way to adopt a standard on a codebase that doesn't meet it yet. They need a burn-down commitment. Without one, the baseline becomes permanent furniture.

One more finding belongs here, because it appears on no pricing page. A hosted tool can be free on its own invoice and still spend money you own. One reviewer here executes as a workflow run inside the repository, consuming the same machine-minute pool as the tests, at roughly 45 seconds a run across hundreds of pull

requests. The check that supposedly confirmed otherwise enumerated one workflow file and was structurally incapable of seeing a differently named one. It returned an empty list, and that empty list got read as nothing exists rather than as this query can't see it.

An observation you couldn't make isn't a negative observation. Report three states, good, bad, and unknown, and never let unknown default quietly into either of the others. A permissions error looks like a no. An empty result looks like none. Neither one is.

Part IV, Security

Where a security defect gets caught

The cost-of-catch curve applies to security findings, and it runs steeper. A functional defect reaching production is an incident with a rollback and an apology. A security defect reaching production may already have been found by somebody else, and the clock on it started before anyone internal noticed. The argument for moving catches leftward doesn't change. The payoff at the left end gets larger.

Three checks belong in Ring 0. They're fast, they're absolute, and they produce no judgment calls anyone needs to arbitrate. Secret scanning asks whether the diff introduces a credential. Dependency scanning asks whether the change pulls in a package carrying a known advisory. Structural lints enforce whatever security rules the house has written down, at the same zero marginal cost as every other lint. All three belong to the class of check that volume defeats rather than strains, because each one substitutes for an act of human attention that used to happen on every diff and that nobody at fourteen pull requests a day is still performing. Nobody is reading a diff looking for a hardcoded key.

Static analysis is slower and noisier. It sits in Ring 1, scoped to what changed, with the full sweep in Ring 2. Run it everywhere on every change and the team watches it take twenty minutes, then turns it off. A real control becomes a disabled one with a green history behind it.

The flywheel carries over intact, and it matters more here than anywhere else in the piece. A security rule the reviewer is asked to remember carries exactly the reliability of anything else a reviewer is asked to remember. Every recurring security finding is a candidate for promotion into an executable check. The defect registry is where that memory lives. A security class arriving in it should bring a test and a lint, rather than a paragraph.

What the agent is allowed to touch

Agent operating rules read like documentation and function as access control. They're the written answer to three questions: what the agent may change unattended, what requires a human on the pull request, and what it may not touch at all. Most teams have answers to those questions. Very few have them written down anywhere the agent can read.

Some paths belong on the human-required list by default. Credential and secret material sits there. So does access policy, authentication and session handling, and anything touching a database migration. The one that gets forgotten is the pipeline's own definition. An agent that can edit the workflow file gating it can remove its own gate, and the change arrives looking like any other green pull request with a reasonable description attached. Whether that's a live risk depends on the permissions model underneath, which is worth checking rather than assuming.

Blast radius needs the same scrutiny. Say change-scoping decides what runs, and the scoping rule came out of a test-coverage graph. A change to an authentication helper can then skip the security suite entirely, because nothing in that suite imports it. A blast radius accurate for correctness can be wrong for security. It fails in the direction of running less.

Behavioral verification applies here with more force than anywhere else in this piece, because a security control that has never been demonstrated catching anything sits in precisely the state that directory-scoped guideline sat in for however many months nobody looked at it. It parsed. It reported. It was silent, and silence is indistinguishable from absent.

The last week of the month

Fail-closed budgets stop a metered service rather than letting it bill, which converts an unbounded invoice into a bounded loss of capability. That's a real trade rather than a free win, and it raises a question deserving a written answer. Which layer truncates first?

In this build the answer is the pipeline, and the consequence took a while to state plainly. If compute exhausts mid-month and the gate stops running, the release process has quietly changed from tested, reviewed, and merged, to reviewed and merged, and that change lands on the last stretch of every month, which is exactly the window when work ships closest to a deadline. Every scan living inside that pipeline truncates with it, security scans included. So the least-checked code of the month is also the code under the most schedule pressure.

None of that shows up on a dashboard unless somebody deliberately put it there. It shows up in an audit as a control that operates three weeks out of four.

Part V, What gets funded

What shippable means at this rate

Product-worthy stopped being a judgment somebody makes at the end. It's a property the pipeline either enforces or doesn't. At fourteen changes a day, a change is shippable when the gates it passed were real gates, when the tests it passed could actually have failed, when the reviewer clearing it was spending judgment rather than catching naming problems, and when the release process can say what's in production and roll back that one change rather than a day's worth of them.

That last clause is where release management usually hasn't caught up. Rollback granularity has to match merge granularity. Merge twelve times a day with a daily rollback unit and you'll be rolling back eleven good changes to remove one bad one. Release notes stop being writable by hand at this cadence too, so either every

change carries structured metadata that assembles into notes automatically, or the notes quietly become fiction. People make rollback decisions from those notes.

When something does escape, the incident review that pays for itself doesn't ask what the bug was. It asks which layer should have caught this and why it didn't, and then it fixes that layer. Severity is a function of how far right the defect got. Caught by a lint at authoring time is nearly free. Caught in human review is a prevention gap. Caught in production is a gap in the test matrix, which is a statement about the matrix rather than about whoever wrote the change.

Minimum and target, by layer

Not every project needs all of this. The minimum is higher than it used to be, because an agent generates volume that overwhelms informal process far faster than a person does.

Table 2. Minimum viable and target state, layer by layer.

LAYER	PROTOTYPE, MINIMUM VIABLE	PRODUCTION, THE TARGET
Scaffolding	One standards file the agent reads. Fifty lines beats nothing.	Standards, an append-only defect registry, per-directory rules, and executable structural lints, all promoted from real defects.
Defect registry	A running list in one file.	Structured entries carrying symptom, root cause, and the check that catches it. Each new class also adds a test and a lint.
Pyramid	Unit tests on logic, plus one smoke test proving it boots.	All three tiers, with an explicit rule that no tier may be empty for a component that owns code.
Rings	Everything runs every time, while the suite is still small.	Ring 0 unskippable, Ring 1 scoped by a derived blast radius, Ring 2 full on a fixed cadence.
Blast radius	In somebody's head.	Derived rather than declared, computed from the import graph, with unmapped code failing the build.
Sharding	None. Do not shard a four-minute suite.	Vertical slices, plus specialty shards carrying a maximum-staleness alarm.
Security	Secret scanning on every push.	Secret and dependency scanning in Ring 0, static analysis scoped in Ring 1 and full in Ring 2, security classes carried in the registry.
Agent rules	A short written list of what the agent may not touch.	Path-scoped operating rules covering credentials, access policy, auth, migrations, and the pipeline definition itself.
AI review	One lane, grounded before anyone judges it.	One paid lane grounded with executable rules, a second independent lane for corroboration, and a hard cap on lane count.

LAYER	PROTOTYPE, MINIMUM VIABLE	PRODUCTION, THE TARGET
Human review	The author re-reads it.	Reserved for judgment: design, tradeoffs, and whether the thing should exist.
Gates	A documentation fast path.	Change detection, path filters, title keywords, and tree-hash skip, every one of them verified behaviorally.
Release	Merge to main, deploy by hand.	Automated promotion on green, staged rollout, one-command rollback, and a written answer for an exhausted budget.
Cost control	Know what you are paying.	Every metered line carries a budget and a documented failure mode, and somebody owns the meter.
Ownership	The developer.	A named owner for the pipeline itself, distinct from the product it ships.

Table 2 is the complete data set for what it shows, so it takes no appendix.

The honest read on the left column is that most of the right column can wait at prototype stage. Three things can't wait past the moment a license is issued, because their absence compounds silently rather than announcing itself. Scaffolding has to exist even at fifty lines, or the agent's output quality is a lottery and its reviewer's output is noise. One unskippable gate has to exist, or volume outruns judgment inside a month. And somebody has to be reading a meter rather than a pricing page, or the first accurate number arrives on an invoice.

Where the money goes

The intuition that the AI is the expensive part turns out to be right, though not for the reason most people expect. The second-largest line surprises nearly everyone.

Table 3. Share of a monthly meter-inclusive stack at roughly 280 code pull requests.

LAYER	SHARE	NOTE
The coding agent	about 65%	Largest line by a distance. Overage ran roughly twice the subscription.
CI compute	about 28%	Scales directly with volume, and it is the only line that does.
AI review	about 5%	The layer this kind of article usually spends its whole word count on.
Everything else	about 2%	Hosting and tooling.

Two things there are worth sitting with. The subscription price and the metered reality are different numbers, often by more than two times, and overage here ran roughly twice the subscription. Three days went into proving exactly that point for two review vendors while the agent's own line was still priced at its subscription, and that error is worth naming plainly: the check got run on every line except the one running the check. Almost every line is flat except compute, which means volume discipline is the available lever and it lands on the compute bill first.

One number makes the rest concrete.

A full pipeline run here bills about 42 machine-minutes, which is 1.4 percent of a typical included allowance of 3,000 minutes a month. That figure reframes several habits at once. One extra review round per pull request, across 280 of them, is roughly 11,760 extra minutes a month plus the agent time to author each fix, which turns a small process inefficiency into a four-figure annual line. Push discipline moves the compute bill by something like 40 percent before any vendor conversation happens, and batching pushes rather than pushing on every small fix is the cheapest optimization anywhere in this article, since it needs nobody's approval and no contract change.

Pushing while a review is running is wasteful three times over. It discards the in-flight review, and that part is visible. It cancels the run, and that part is silent. On a metered reviewer it bills the charge anyway, and that part stays invisible until the invoice arrives. On the first metered pull request here, five of eight charges were

self-inflicted pushes that a one-hour batch would have collapsed into one. Agent cost is a function of how the agent is allowed to work, rather than only of what the vendor charges.

Before signing anything, the pricing shape matters more than the headline number. A flat per-seat plan never fails on the invoice, because it protects itself with a throughput limit instead, so the question is what that rate limit is and whether it degrades with volume; a flat plan at high volume is a latency budget rather than a spend one. A per-unit plan fails immediately at agent volume, and the question there is what the billable event is and how many of them the agent generates per unit of work. Those are two different questions, and only the second one is about you. Included credits with overage fail at the allowance boundary, so ask whether there's a free option at the wall or only a spend cap, because a cap bounds the damage without letting anyone choose to wait instead of pay.

One rule generalizes past tooling entirely. Meter one real unit of work before extrapolating. The estimate derived here from pricing pages came in five times low, and no amount of careful reading would have corrected it.

The readiness gate

Run this before the licenses go out, rather than after the first incident.

Table 4. The readiness gate, run before licenses are issued.

AREA	THE CHECK
Scaffolding	A standards file exists and the agent is configured to read it. A defect registry exists, even if it starts empty. A named person owns keeping both current, and it is a person rather than a team. Every configuration has been verified behaviorally rather than parsed.
Quality	Ring 0 exists, runs fast, and cannot be bypassed. A full run happens on a fixed cadence and somebody notices when it does not. No component owns code behind an empty test tier. Tests assert behavior rather than the presence of text.
Security	Secret scanning and dependency scanning sit in Ring 0. Static analysis is scoped rather than global. Recurring security findings are promoted into executable checks. Somebody has watched each control catch something on purpose.
Agent authority	What the agent may do unattended is written down, before the first incident rather than after. Credentials, access policy, authentication, migrations, and the pipeline definition itself require a human.
Cost	Every metered line carries a budget and a documented behavior at the limit. Somebody reads the meters weekly, meaning the meter rather than the pricing page. One real unit of work has been metered end to end before anyone extrapolated. The team knows which layer truncates first.
Process	Push and batching discipline is written down, because it is now a cost control. Human review is explicitly reserved for judgment. There is a defined answer to what happens when review says merge and the pipeline says no. Somebody owns the pipeline as a product.

Table 4 is the complete data set for what it shows, so it takes no appendix.

One item on it deserves an argument rather than a checkbox. Cap the number of AI review lanes, and cap how many of them are paid. Two lanes, at most one paid, is the rule here.

More reviewers aren't worse. A second independent lane produced the single strongest quality signal in this whole record, when two reviewers flagged the same

real defect within twenty-four seconds of each other, neither one able to see the other. That's corroboration nobody had to take on trust.

The cap exists because lanes accumulate by accident without one. One lane here arrived bundled with an unrelated subscription upgrade. Nobody had chosen it, nobody had budgeted it, and it may well have put this build over its own rule. A portfolio limit is only real if something re-checks it when a subscription changes.

Open gaps

A list of solved problems teaches less than a list of open ones, so here are the ones still open.

An escaped defect still can't be priced here. Which layer should have caught it is answerable. What it cost isn't, which leaves the question of whether a layer earns its price to be argued rather than calculated. There's one strong instance of two review lanes corroborating independently. There's no measurement at all of how often two lanes miss the same thing, and that second number is the one that would actually justify a second lane.

The vertical-shard map is derived from the dependency graph and isn't yet driving what runs. Wiring it up is the work in front of this build. The interesting risk is that a derived blast radius is only as good as the dependency analysis underneath it. Specialty shards are a discipline this build doesn't yet trust itself on, since the staleness alarm is a design and not a track record.

The widest gap sits between code-level guardrails and agent-level ones. Whether the code that got written meets the standard is handled reasonably well. Whether the agent should have attempted the change at all is handled much more weakly. That gap grows as autonomy does.

Nearly every wrong conclusion in this record came from a measurement that couldn't see what it was assumed to see: a duration file blind to imports, a query scoped to the wrong file, a counter that wasn't the meter, a parse check that wasn't a behavior check. The recurring lesson is to check what the instrument can actually observe.

What this piece does not establish

Anything quoting this inherits its limits, so they're stated here rather than left to be discovered.

It comes from one organization, one workload class, and an unusual volume. The mechanisms carry over. The multiples don't carry over, and a team running twenty pull requests a month sits comfortably inside most vendors' included allowances, never meets any of the walls described here, and would be badly served by budgeting against these numbers.

The security material in Part IV is argued from the same layer model as the rest, and it isn't backed by the measured record the other parts carry. Those mechanisms are standard. Their placement follows from the cost curve. No figure in that part should be read as measured, because none is offered.

The minimum-and-target table is a judgment rather than a measurement. The left column has been run. The right one is being built toward, and no controlled comparison of the two exists here. Several cost figures are derivations from measured inputs rather than invoice lines, corroborated against billing history where that was possible.

The whole thing is survivor-biased. These are the pitfalls that got hit and noticed. The expensive category is the one made up of pitfalls that got hit and haven't been noticed yet, and by definition none of those are in here.

Glossary

TERM	DEFINITION
Blast radius	What else a change to one component can plausibly break. It's the input to any decision about what a pipeline needs to run. If it's a guess made per incident rather than a computed artifact, change-scoped testing isn't safe.
Continuous integration	The automated build and test run that fires when code is pushed. Here it's the third layer in the cost curve, and the second-largest line on the bill.
Defect registry	An append-only file recording every bug class that shipped and caused harm, with the symptom, the root cause, and the check that would have caught it. It turns a one-off incident into a permanent guard.
Escaped defect	A defect that reached production through a pipeline built to catch it. Severity is measured by how far right it travelled.
Fail-closed	The behavior of a metered service that stops rather than bills when its budget is exhausted. It converts an unbounded invoice into a bounded loss of capability, which is a genuine trade rather than a free win, and it raises the question of which layer goes quiet first.
Flywheel	The loop carrying a finding from discovery, into a written rule, into an executable check. It's the mechanism by which a review layer gets cheaper over time.
Gate	A check evaluated before expensive work is dispatched. It costs nothing when it fires.
Quality pyramid	The tiered test structure: many fast unit tests, fewer integration tests, fewest end-to-end tests. Each tier catches a different class of defect at a different price, so skipping one moves the cost rather than saving it.

Newell, J. (2026). MAX Research Collective, WPR-CLD-001. CC BY 4.0

TERM	DEFINITION
Ratchet	A check that permits the violations already in a codebase and blocks new ones. It's the only realistic way to adopt a standard on a codebase that doesn't meet it yet.
Ring	When a tier runs, as distinct from whether it exists. Ring 0 runs always. Ring 1 runs per change, scoped to the blast radius. Ring 2 runs the whole suite on a periodic cadence, and it is what makes the scoping in Ring 1 safe to rely on.
Scaffolding	The written, machine-readable context an AI tool reads before it acts: coding standards, a defect registry, per-directory rules, structural lints, and agent operating rules. It's the largest single quality lever available, and it appears on no invoice.
Secret scanning	An automated check asking whether a diff introduces a credential. It belongs in the always-on ring because it's fast, absolute, and needs no human to interpret the result.
Shard	A slice of the test suite running on its own parallel machine. It cuts wall-clock time. It doesn't cut cost, since most billing sums per-machine minutes.
Static analysis	Automated inspection of source code for defect and vulnerability patterns without running it. It's slower and noisier than a lint, so it belongs scoped to what changed rather than everywhere on every push.
Structural lint	An executable rule matching on code structure rather than on text. It runs on every diff at no marginal cost. That's what makes it different in kind from the same rule written as prose.
Suggestion	A rule the expensive process reads while it's already running. It can improve an outcome. It can never save money, because reading it is the run, which is the distinction that separates it from a gate and the one most commonly missed when a configuration file is being audited for cost.

CLOUD / DEVOPS WPR-CLD-001

// ABOUT THE AUTHOR

Jason Newell leads organizations and people in technology, and is a researcher, deep technologist and builder — from a CRAY to agentic AI. He founded MAX Research Collective to build AI for regulated industries that runs where its owner has control, with healthcare as its strongest focus. About Jason → Connect on LinkedIn →