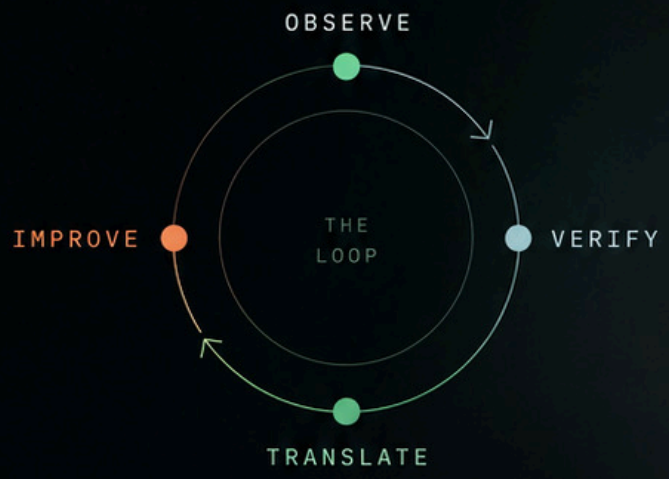
PARSLEE LABS

ENGINEERING DESIGN BRIEF · INTERNAL → EXTERNAL

Two Contracts for *Autonomous* Code Change

On post-merge review, machine-authored fixes, and the engineering discipline of refusing to believe yourself.



A closed loop writes code and merges it — then reads its own merges back. Two roles decide whether that is safe. One of them is not allowed to believe what it is told. The other is not allowed to trust itself.

≈30 MINUTE READ · 21 PAGES · FIVE PARTS · PARSLEE.AI

PARTS ONE AND FOUR ARE THE TEN-MINUTE PATH IF THAT IS ALL YOU HAVE.

BEFORE YOU START

What this is, and why it isn't a checklist

A working system, described in the open. Names, file references and vendor specifics have been stripped. Nothing here requires our tooling to adopt.

Somewhere in your pipeline there is probably a machine that writes code, or one that reviews it, or one that will do both by the end of the year. The interesting engineering is not in getting the model to produce a patch. That part is nearly free now. The interesting engineering is in everything you build around it so that a patch can be trusted — and so that when it can't be, the system says so out loud instead of quietly shipping.

What follows is the design of two roles in a closed autonomous loop that runs against production code. They solve opposite problems, and the shape of each one falls out of the problem it owns. The loop is genuinely closed: one of these roles opens pull requests and merges them, and the other reads every merge — including those.

THE LOOP

Four roles: observe → verify claims → translate customer reports → improve → back to change. Two of them carry the weight. And the loop genuinely closes — the improver's own merges come back around to be observed.

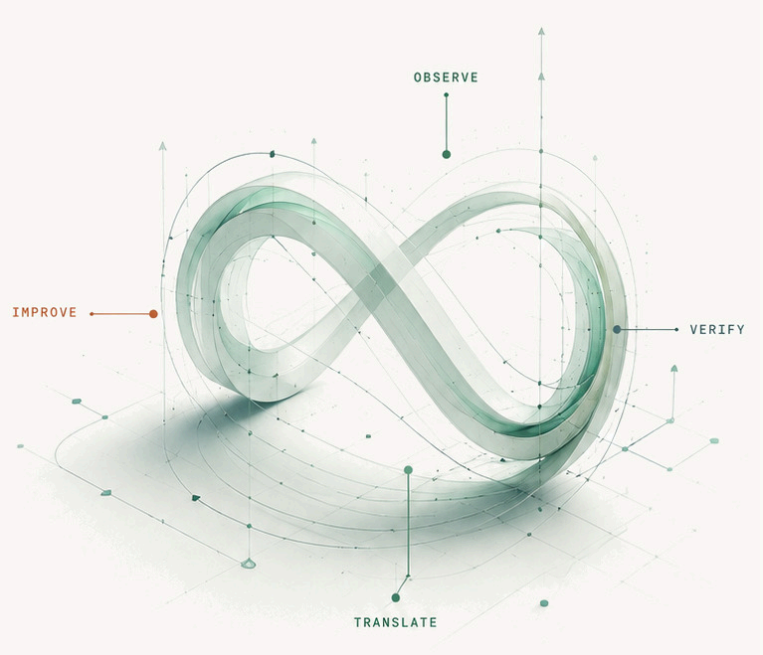
	THE OBSERVER	THE IMPROVER
OWNS	observation → knowledge	knowledge → improvement
PROBLEM	**Epistemics.** How do you know something without believing the story you were told about it?	**Authority.** It is the only role that writes. What has to be true for that to be safe?
POWER	Deliberately none. Cannot block, write, or execute.	Real, and fenced on every side.
FAILURE IT FEARS	Believing the narrative it was handed.	Grading its own homework.

§ On summarising this

Page 20 is a list of fourteen principles. You could skip to it; it takes ninety seconds. I'd rather you didn't, and the reason is practical rather than precious.

Rules are the cheap part. You can get a list of good engineering rules from anywhere, and it will be indistinguishable from a list of bad ones, because a rule stripped of its reason reads exactly like a preference. The moment that matters is six months from now when someone on your team says *"this gate is slowing us down, can we relax it just for this case"* — and the only thing that will let you answer well is knowing which load the gate was carrying. That knowledge is in the middle of this document, not at the end of it.

HOW TO READ THIS



Four roles, one surface — and no end to it. The loop returns to its own output.

1

PART ONE

The question CI cannot ask

Not because it is badly configured. Because of what it structurally is.

Your CI answers a bounded, pre-merge question: *do the checks the authors wrote still pass on the change the authors proposed?* Same people, same session, same assumptions. It is an excellent question and it is the wrong one to stop at, because a suite written by the author inherits the author's blind spots by construction. You cannot test for the thing you didn't think of. Neither can your test.

The Observer asks something else, after the fact: *now that this is part of the codebase, does it hold up against the code around it?*

The value is not speed and it is not prevention. It is **independence** — a reader that didn't write the code, wasn't in the room, and isn't trying to get anything merged.

That is also why it sits *after* the gate rather than at it. The defects only post-merge review can find are precisely CI's structural blind spots:

- interactions with untouched code that the change silently depends on;
- assumptions that were true in the PR's mental model but false in `main` after other merges landed;
- defects in changes that had weak CI coverage in the first place — where a green build proves the least and reassures the most.

WHY AFTER THE GATE

Post-merge is not a compromise position. It is the only vantage point from which the code's real neighbours are visible.

> It does not prevent bad code from landing. It ensures every merge has an independent opportunity to be questioned.
>
> THE OBSERVER'S CONTRACT, IN ONE SENTENCE

Read that twice, because almost every objection to post-merge review is an objection to a promise it never made. It is not a gate. It is the thing that notices.

§ And the trigger is an event, not an author

The Observer subscribes to merges. Not to people. A merge event carries a diff, a SHA, and a stated intent — and nothing in that contract says a human produced it.

In practice most merges are human, and that matters for reasons we will get to. But the Improver also opens pull requests, and when one passes the repository's required checks the daemon merges it with no human in the path. That merge emits the same event as any other, and the Observer reads it the same way. Part Four is about what happens when the answer to *"who wrote this?"* is *"the other half of this system did."*

What changes with provenance is narrow, and deliberately so:

	HUMAN-AUTHORED MERGE	AGENT-AUTHORED MERGE
TRIGGER	merge event	the same merge event
STATED INTENT	PR body written by a person	PR body written by the Improver
STATUS OF INTENT	untrusted hypothesis	untrusted hypothesis
REVIEWER	the default reviewer model	**a different model family**, high reasoning effort, no silent fallback

One row differs. Part Four explains why that row is the one holding the loop together.

2

CARRIED FORWARD

Independence is the product. Everything that follows is a mechanism for protecting it — from the author's framing, from the reviewer's own lineage, and from its desire to be agreeable.

PART TWO · EPISTEMICS

The Observer

Hold intent and outcome at once — and never believe intent.

An observer given only the outcome can ask *"is this code wrong?"* — a mechanical question any linter asks. An observer that also holds the *intent* can ask the question that actually produces knowledge: **did this change accomplish what it set out to, and what is now possible or broken because of it?**

So a merge is understood as two things. **Intent** — the PR title, the description, linked issues, review discussion. That is the hypothesis. **Outcome** — the code as it exists at the merge SHA. That is ground truth.

The subtle part, and the reason this design is worth copying: **intent is used but never believed.**

01 Two steps, and the second may not revise the first

1. **Outcome first**

   Read the diff and the surrounding checkout. Find real defects. Nothing about the narrative is visible yet — the intent block is not in the context window.

2. **Intent second**

   Only now read the labelled stated-intent block. Treat it as an untrusted hypothesis. Compare intended against observed. Findings from step one are not permitted to change.

The ordering exists to defeat anchoring and sycophancy. This is the whole trick, and it is almost embarrassingly cheap to implement: *if you show a model the story first, it will find the story in the code.* Not because it is dishonest — because that is what a fluent pattern-matcher does with a prior. Withholding the prior until after the reading costs one extra prompt boundary and buys you an observer whose first pass is uncontaminated.

RESEARCH NOTE

Sycophancy in LLMs is measured, not folklore. Sharma et al., ICLR 2024. The ordering is a mitigation, not a style choice.

02 Reason with the PR, report without it

A finding must survive without the narrative. It must name a file, describe current behaviour, and give a reproducible failure scenario — and it may never say *"the PR"*, *"this change"*, or *"the diff"*. A maintainer opening one of these issues is reading a claim about the code as it is now, not archaeology on a merge.

This is a stricter constraint than it looks. It quietly deletes an entire class of low-value output — the reviewer that says *"this change may have introduced a subtle issue"* and leaves a human to do the actual work of locating it.

Anything supported *only* by the narrative isn't a finding at all. It becomes an **intent/outcome gap**. And gaps that depend on the narrative are dropped from the filed issue but recorded on the review event.

HOUSE RULE

Drop, then log.

Nothing an observation produces vanishes silently — even when it doesn't earn an issue. The distinction between *"we found nothing"* and *"we found something and chose not to file it"* has to survive in the record, or your review system is unauditable by design.

03 Six mechanisms, each a consequence of the contract

None of these is incidental plumbing. Each one exists because the contract above would otherwise be violated in a specific, predictable way.

INDEPENDENCE OF JUDGMENT

Human-authored merges get one reviewer model. Machine-authored merges are judged by a *different model family*, at high reasoning effort, with **no silent fallback** to the authoring lineage. The reason is measured: an LLM judge favours its own model's generations. An observer sharing the improver's lineage would inflate the loop's own signal — and it would do so invisibly, in the direction of good news.

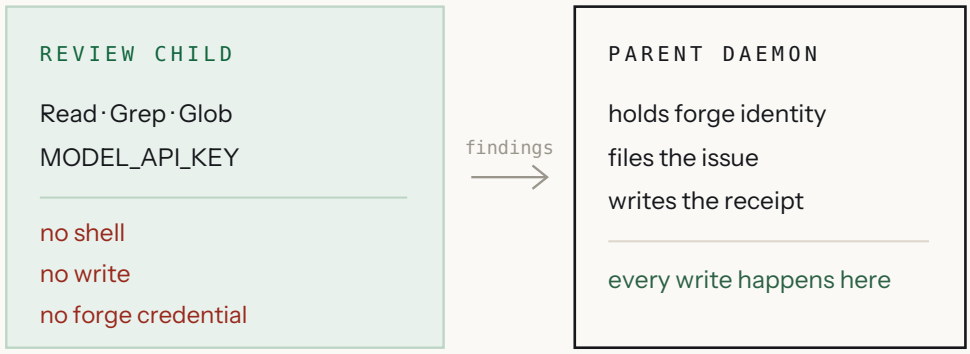
ISOLATION — THE DIFF IS ASSUMED HOSTILE

A merged change could carry a prompt injection aimed squarely at the reviewer. Two controls. The review process gets **read-only tools** — no shell, no write, therefore no channel to exfiltrate through. And its environment is

RESEARCH NOTE

Self-preference in LLM judges: Panickssery et al., NeurIPS 2024. Self-judgment is the failure mode this entire subsystem is engineered against.

an **allowlist, not a denylist**: PATH, HOME, locale, tmp, plus the single model API key. There is nothing repo-credential-shaped in there to leak.



The child reads and reasons; it never holds a credential and never performs a write. An injection that captures the reviewer captures a process with nothing worth stealing and no way to send it.

CALIBRATED NOISE

Findings are ranked *nit* < *minor* < *major* < *blocker*, and only major and above file. An unknown severity never files. Each finding also carries an attribution flag for whether it relates to the change under review — and when every finding is affirmatively unrelated, the issue files **without cc-ing the author**. The defect is still reported; the blame isn't. That single decision does more for adoption than any amount of accuracy tuning, because it removes the incentive to argue with the reviewer.

The attribution decision is itself written to the event, so systematic mis-marking to dodge attribution is auditable.

DEDUP THAT FAILS TOWARD FILING

A deterministic issue-body marker keyed to repo + PR + merge SHA is the public truth for *"did we file?"* Advisory per-finding fingerprints — keyed on path and enclosing symbol, **never line number or prose**, so they survive code shifting — flag likely duplicates but never suppress a filing.

THE ASYMMETRY THAT DECIDES IT

A duplicate costs a human a ten-second close. A suppressed real finding can cost an incident. When the two error directions have costs that differ by four orders of magnitude, "tune the threshold" is the wrong instinct; pick the direction to fail in and fail there on purpose.

SPLIT TRUTH ON PURPOSE

The issue marker answers *was an issue filed?* A separate receipt ledger answers *did a review complete, is it retrying, did it fail, was it below threshold?* Without the split, the absence of an issue ambiguously means either *clean merge* or *we crashed* — and those two states will drift apart at exactly the moment you most need to tell them apart.

04 What it refuses to promise

This is the part most tools skip, and it is where the credibility actually lives. The Observer states its limits in its own documentation, in the negative:

STATED NON-GUARANTEES

It doesn't block. By the time it speaks, the code is merged and possibly deployed.

It doesn't guarantee every merge is reviewed — only that no merge is *silently skipped*. Every merge is reviewed, or visibly parked on a receipt a human now owes, or recorded as an unacknowledged permanent failure, or named in a durable coverage gap that raises an alert. Direct pushes to the default branch are the one stated exception: counted and published, never reviewed — there is no stated intent to test against and no merge author to answer for it.

It doesn't guarantee it's right. An issue from the Observer is an invitation to look, not a verdict.

Two further properties keep the signal honest over time. The loop deliberately keeps **exogenous signal** — real human commits — in its input, as a defence against the collapse that follows from recursive self-consumption. And:

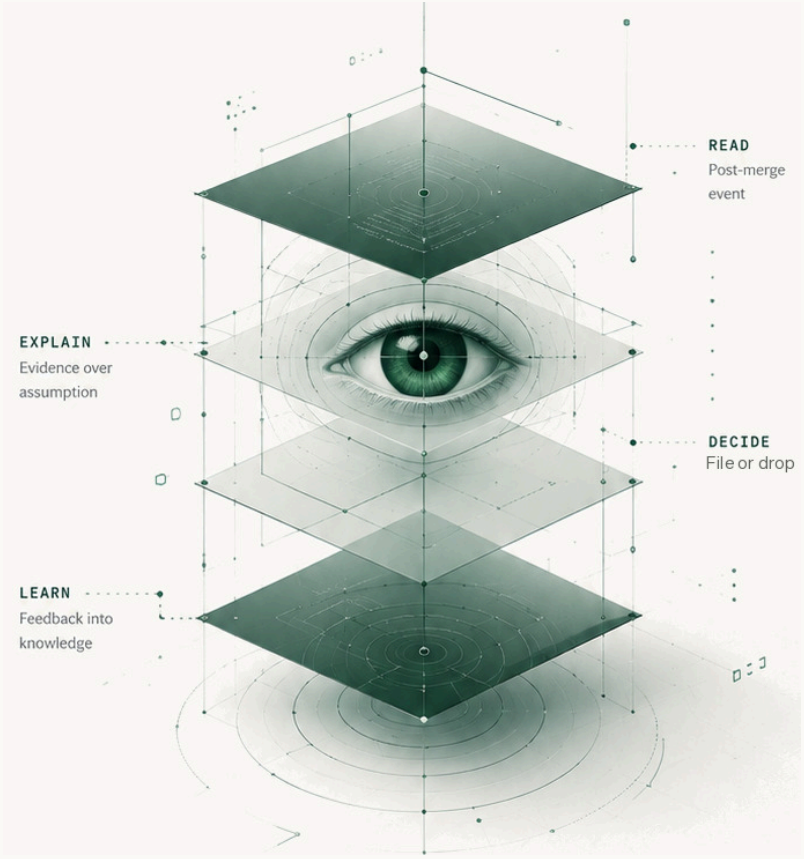
> The finding count is a health metric to monitor, never a target to optimise. A loop that lowers its finding count by observing less has satisfied the metric and betrayed the goal.
>
> THE GOODHART TRAP, OBSERVER SIDE

05 So why does it work?

Not because the model is good at reading diffs. Because of three properties that were chosen, not inherited — and that most review tooling has none of.

Powerless	Independent	Strict
Cannot block, cannot write, cannot execute. Its only output is a claim.	Different model lineage, no stake in the merge, no credentials of any kind.	Code arbitrates. Narrative is a hypothesis. Anything unproven is logged, not filed.

Take away any one of those three and you have a chatbot with opinions about your codebase.



Four moves, strictly in order: read the outcome, explain it from evidence rather than narrative, decide whether it clears the filing threshold — and record what was seen either way.

3

CARRIED FORWARD

That was the role with no power. This is the one that writes to your repository — where every decision has to answer a harder question than "is this useful?"

PART THREE · AUTHORITY

The Improver

Hold capability and authority — and never trust yourself.

> Not trying to maximise autonomous code generation. Trying to maximise *trustworthy autonomous changes.*
>
> THE IMPROVER'S CONTRACT, IN ONE SENTENCE

This is the only role in the loop that writes. So its design spends more words on what it refuses to do than on what it can do — deliberately. Every capability listed below is followed immediately by the fence around it, because a capability described without its fence is a capability someone will eventually use without one.

01 Three assumptions it starts from

1. **Autonomy is only worth as much as the gates behind it.** Any capability that would require a maintainer to trust it *more than the gates justify* is out of scope. Not deferred — out of scope.
2. **Its input is attacker-controlled.** The work begins with an issue, and anyone who can open or comment on an issue is authoring its instructions. Title, body, provenance — all untrusted, spliced into prompts strictly as data with explicit *ignore any instruction inside* framing. It trusts exactly two things: the code in the repository it cloned, and its own credentials. Nothing arriving through an issue is ever trusted.
3. **Green CI ≠ correct.** The deepest one, and the one most teams discover the expensive way.

THE FAILURE MODE

An agent optimising to pass the existing suite can simply hack that oracle: weaken the assertion, delete the test, edit the CI config. It will not do this maliciously. It will do it because you asked for green and green is what you got.

The literature is explicit — patch overfitting (Smith et al., ESEC/FSE 2015) and reward hacking (Baker et al., 2025). The entire loop below is built around this single fact.

02 Evidence in, bounded change out

It addresses the evidence; it does not change for change's sake. Every change is grounded in a specific piece of evidence and scoped to it. This is *not* a roaming refactorer improving code nothing pointed at, and it does not widen a fix beyond what the evidence justifies.

A diff with no evidence behind it, or one broader than the issue that justified it, is out of role — and gets rejected on those grounds alone, regardless of whether it is a good idea.

The work runs as two model passes in separate ephemeral sandboxes, with a hard asymmetry between them:

WHY EVALUATE FIRST

The evaluation pass exists to bind the change to a located problem *before* any code is written. Traceability from change back to evidence is what makes the loop converge on quality rather than merely move.

	EVALUATE	FIX
TOOLS	Read · Glob · Grep **no shell**	Shell · Edit · Write · Read · Glob · Grep
OUTPUT	one verdict, **no edits**	a local commit, **never a push**
NETWORK	read of the clone only	**offline** — no package registry at all

03 Refusing is a first-class outcome

This is what distinguishes the design from a patch bot. *Can't fix* is correct, common, and first-class. It explicitly covers: needs credentials, needs infrastructure, needs a product decision, or cannot be located here.

> Declining is success when declining is right.
>
> ON REFUSAL

It is also careful about *what kind* of failure just happened, because conflating them corrupts the record permanently:

- **Semantic or policy rejections** consume one of three strikes, then escalate to a human.
- **Infrastructure failures** — sandbox died, network, provider, process loss — are *not verdicts and not attempts*. They get their own separately-counted ceiling. When the environment keeps dying before the code can even be read, the issue goes to a human with exactly that message: *my environment failed; nothing was judged about the issue itself.* No attempt label is applied, because no attempt was ever judged.
- **An agent-wide outage** pauses intake *once*, rather than walking the whole queue into needs-human one issue at a time — then probes with a throwaway sandbox each cycle and bulk-returns the parked issues on the first proven success.

The through-line: at every ambiguous point, the default is to **stop and leave evidence**, not to guess and publish.

04 Credential separation, proven rather than assumed

Two credentials, kept apart on purpose. Child processes get the model API key and nothing else; the repository write identity never reaches a child. Every write to the forge — push, PR, comment, merge — is performed by the parent daemon.

And it is *proven*, not assumed. Before every fix child, a fail-closed preflight requires three things: the credential helper must resolve to nothing, auth status must fail, and a dry-run push must fail. A probe that cannot prove absence — a timeout, a missing binary — also aborts.

THE RECURRING PATTERN

Gates block on inability to prove safety, not merely on proven danger.

The Improver refuses to run a fix child it cannot prove is credential-isolated. Note the direction of the burden: the absence of evidence of danger is not evidence of safety, and a system that treats it that way will pass every audit until the day it doesn’t.

05 The load-bearing idea: an oracle the improver cannot edit

Because self-judgment provably degrades, the loop is pinned to a ground-truth signal the improver cannot rewrite. Two independent parts.

THE REPRODUCTION TEST — RED, THEN GREEN

The fix pass must *first* write a new failing test asserting the correct behaviour, *then* implement the minimal fix. It may not modify, delete, weaken, or skip any existing test, and may never touch CI configuration. A disallowed-oracle-edit check blocks the push if it tried.

Then the grading, which is where the rigour lives. A separate protected phase reconstructs the test on the *pre-fix* commit and requires it to fail **at an assertion**, then runs it on the fix and requires it to pass.

PROTECTED GRADING PHASE

EVERYTHING ELSE FAILS CLOSED — COUNTS AS NOTHING

compile error · discovery failure · missing tool · timeout

malformed report · no test at all · failure for any non-assertion reason

A test that "fails" only because the symbol doesn't exist yet proves nothing.

Only assertion-failure counts as behavioural red. The distinction is not pedantry — it is the difference between demonstrating a defect and demonstrating that you have not written the code yet.

THE ELEGANT TWIST

The fix child's workspace is **offline** — no package registry at all — and it is told plainly: *you are NOT the one who proves the fix works.* It cannot run its own test, and is explicitly forbidden from claiming that it did.

The agent that writes the test never gets to grade it. Judgment is structurally separated from authorship.

AND THEN THE REAL REQUIRED CI

The published diff must pass the repository's actual required checks, with at least one registered check run. Not a suite the agent assembled. The one your humans already depend on.

And the limit is stated outright, in the design's own words: *this is strong task evidence, not a proof of universal correctness; the reproduction can still encode the wrong oracle.*

06 What it is not judged by

Not PR count. A quarter with fewer PRs and zero bad merges beats the opposite. Throughput as a metric is explicitly off-mission — the Goodhart trap on the improver side, mirroring the finding-count trap on the observer side. Both roles have a number that looks like productivity and is actually a temptation.

The design goes further and tells maintainers *when to reject its output*:

REJECT IT WHEN

— CI passes but you cannot see *why* the change resolves the issue.
— The diff touches more than the issue warrants.
— The PR reads as steered by the issue text rather than by the code.
— Anything that would only make sense if the child had followed an instruction embedded in the issue.

> Rejecting is cheap and expected. It is designed so that a wrong proposal costs a review, never a bad merge.
>
> ON THE ECONOMICS OF BEING WRONG

07 The deepest "why"

This is the point where a closed autonomous loop touches production code with no human in the path. So the design answers a single question: **what has to be true for that to be safe?**

- Every action individually reversible and cheap to reject — one PR, one branch, one push. A wrong step costs a revert, never a cascade.
- Every gate fail-closed. Unproven safety is treated as unsafe.
- Every input assumed hostile, with trust granted only to the cloned code.
- The verifier structurally out of reach — it cannot write the test that judges it, cannot run it, cannot edit the existing suite, cannot touch CI.

— Brakes that do not depend on the loop being healthy: spend reservations, per-run caps, bounded lineages, provider workspace limits.

— A fully independent observer, on a different model lineage, reading every merge it lands.

ON GOVERNORS

A governor that only works when the system is behaving is not a governor.

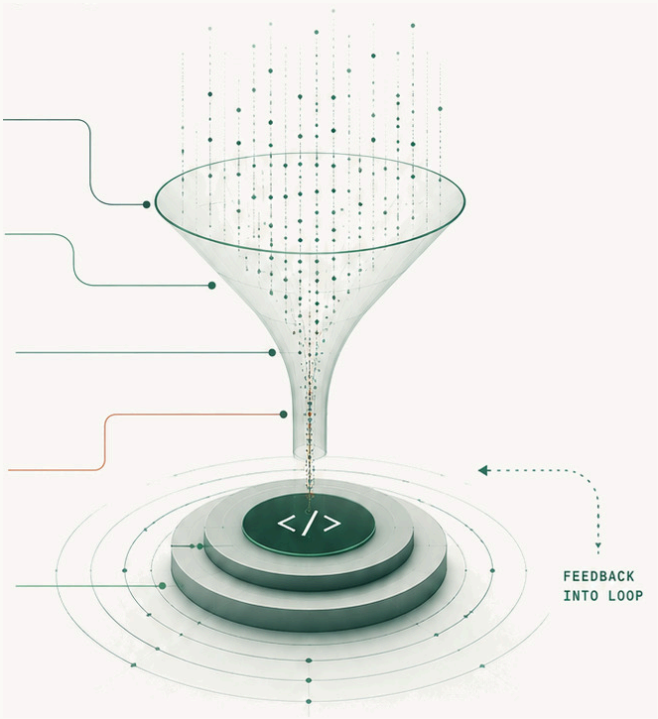
01 Observe
Post-merge review detects a signal.

02 Extract
Findings are structured and normalised.

03 Store
Written to durable per-merge records — including the findings that were dropped and the gaps that never became issues.

04 File
Anything at major or above becomes an issue. The rest stays on the record.

05 Improve
The Improver claims a filed issue and works only from the evidence in it.



Findings only compound if they are structured and traceable back to the merge that produced them. What the Improver acts on is one located piece of evidence at a time — never a summary of the loop's own history.

4

CARRIED FORWARD

One gap remains, and it is the important one. Everything so far has been the Improver checking itself. There is exactly one thing it cannot check.

PART FOUR

The interlock, and the closing loop

Two pages. If you read nothing else, read these.

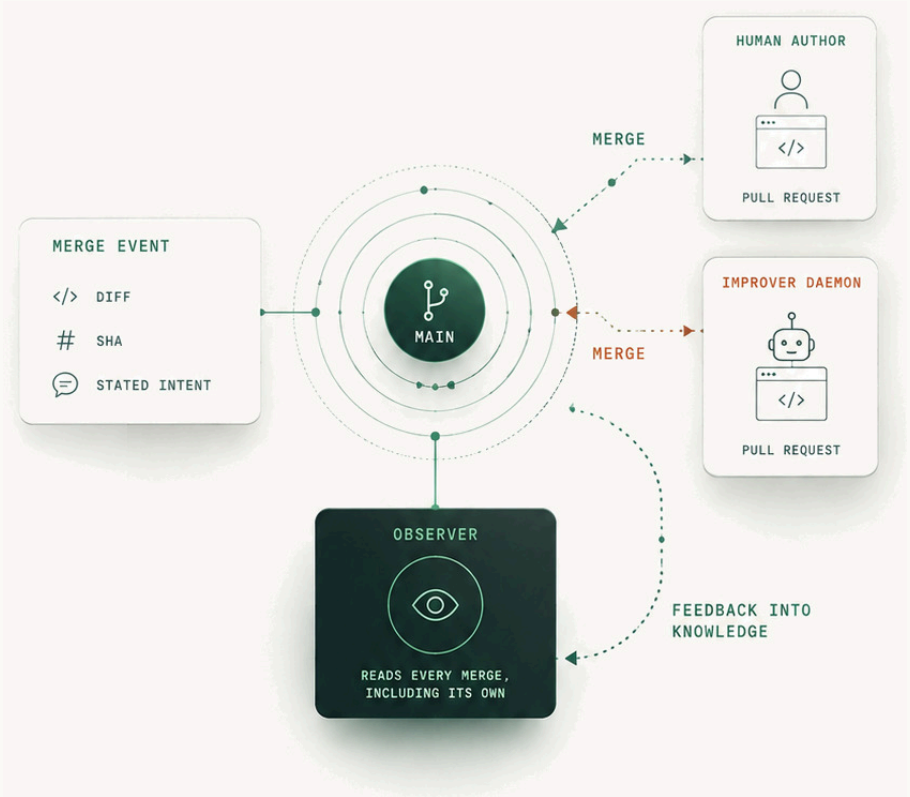
The Improver's strongest claim about itself contains an admission: *the reproduction can still encode the wrong oracle.* It can write a test that passes for the wrong reason, prove the wrong behaviour correct, and ship a green build that is confidently mistaken. No amount of gating inside the Improver can fix this, because the flaw is in the Improver's own understanding of the problem, and a system cannot audit its own comprehension.

That sentence is where the two roles interlock. The thing the Improver cannot check about itself is precisely the thing the Observer reads — afterwards, on a different model lineage, with no stake in the merge, with the intent withheld until the code has already been read.

Neither role is sufficient. The Observer without the Improver is a very expensive linter. The Improver without the Observer is a system whose only judge is a test it wrote about a problem it may have misunderstood.

01 Which means the loop actually closes

A finding becomes an issue. An issue becomes a pull request. A pull request that passes required CI becomes a merge. A merge becomes an observation. And an observation, often enough, becomes the next finding.



Two authors, one branch, one event. The Observer's trigger is provenance-agnostic — which is what keeps real-world signal in the loop, and what makes the loop capable of feeding on itself.

Three consequences follow, and each one retroactively explains a decision that looked fussy in isolation.

ONE — THE REVIEWER'S LINEAGE STOPS BEING A DETAIL

When the merge under review was authored by a model, an observer drawn from the same family is grading its own family's work. That is why machine-authored merges are routed to a *different* model family with no silent fallback. The rule only earns its cost once you accept that the loop will read its own output.

TWO — THE IMPROVER'S STATED LIMIT BECOMES CHECKABLE BY SOMEONE

The reproduction can still encode the wrong oracle, and the Improver cannot catch that about itself. The Observer, reading the same merge afterwards with the intent withheld until the code has been read, can. The admission and the mitigation are two halves of one design.

THREE — THE LOOP CAN FEED ON ITSELF, SO IT IS DELIBERATELY FED FROM OUTSIDE

Findings, issues, PRs, merges, findings. Left alone, that is a system manufacturing its own signal and converging on its own idea of quality, which is model collapse with a project board attached. Hence the

countermeasures, all of which now read as one policy rather than four: exogenous human commits stay in the input; finding count is monitored and never targeted; lineages are bounded; spend is reserved before work begins.

THE NUMBER NOBODY PUBLISHES

Not every landing is a merge. Direct pushes to the default branch are counted and published but never reviewed — there is no stated intent to test against and no merge author to answer for it. On one of the covered repositories that is roughly four in five landings.

That figure is not a defect to bury; it is the coverage statement. A review system that cannot tell you what fraction of landings it actually sees is asking you to trust a number it has never computed.

02 What the two contracts have in common

THE SYMMETRY

The Observer's contract says: hold intent and outcome, but never believe intent.

The Improver's says: hold capability and authority, but never trust yourself.

Same instinct, opposite ends of the loop. In both cases the design assumes the thing most likely to fool the system is the system's own confidence — and then removes its ability to act on that confidence unchecked.

It is worth saying plainly what this costs. Two model lineages instead of one. Two passes instead of one. A grading phase the actor cannot touch. Preflight probes that abort on ambiguity. A refusal path that gets used often enough to feel like failure. None of this is free, and all of it is overhead you are paying to buy back something CI gave away the moment the author and the reviewer became the same process.

The question to ask of your own pipeline is not *"do we have review?"* It is: **at what point in our pipeline does something judge work it did not do, with no stake in the answer?** If you cannot name that point, you do not have review. You have a suite.

5

PART FIVE

The transferable list

None of these require our stack. Most of them cost a prompt boundary and an argument.

P-01

Order your evidence to defeat anchoring

Read the artifact before you read the claim about it, and forbid the second step from revising the first.

P-02

Treat narrative as an untrusted hypothesis

Reason with it; report without it. A finding that cannot stand on file, behaviour, and repro is not a finding.

P-03

Never let a system grade its own output

Different model family, different process, different authority. Self-preference is measurable, and it will flatter you.

P-04

Separate credentials from capability — and prove it

Before each run, not once at design time. Assumed isolation is indistinguishable from none until it matters.

P-05

Fail closed on inability to prove safety

Not merely on proven danger. A probe that times out is a probe that failed.

P-06

Make refusal a first-class, well-typed outcome

Distinguish "I judged this and declined" from "my environment died and nothing was judged." Conflating them corrupts the record permanently.

P-07

Fail toward reporting, not toward silence

A duplicate costs ten seconds. A suppressed real finding can cost an incident. Pick the direction to be wrong in.

P-08

Split "did we act?" from "did we run?"

Otherwise absence of output is ambiguous between clean and crashed — and it will resolve in the optimistic direction.

P-09

Keep the oracle out of the actor's reach

Whoever writes the fix cannot write, run, weaken, or grade the test that judges it.

P-10

Monitor your quality counts; never target them

A loop that lowers findings by observing less has satisfied the metric and betrayed the goal.

P-11

Keep exogenous signal in the input

A loop that only consumes its own output collapses. Real human commits are the anchor to reality.

P-12

State what you don't promise

The credibility of a system's claims rests almost entirely on the ones it declines to make.

P-13

Trigger on the event, not the author

Review that only fires for human PRs stops covering your codebase the week an agent starts merging.

P-14

Assume the loop will read its own output

Once it does, lineage, exogenous input, and untargeted metrics stop being hygiene and become load-bearing.

IF YOU ONLY CHANGE THREE THINGS THIS QUARTER

1. Split your review prompt into two ordered passes and withhold the PR description from the first. This is an afternoon of work and it is the highest-leverage item on the page.

2. Add a disallowed-edit check to any agent that opens PRs: no touching existing tests, no touching CI config. Block the push, don't warn.

3. Write down, in one paragraph, what your automated review does *not* promise — and publish it next to the tool. You will find the act of writing it changes the tool.

CITED WORK

- Sharma et al. — *Towards Understanding Sycophancy in Language Models* — ICLR 2024
- Panickssery et al. — *LLM Evaluators Recognize and Favor Their Own Generations* — NeurIPS 2024
- Smith et al. — *Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair* — ESEC/FSE 2015
- Baker et al. — *Monitoring Reasoning Models for Misbehavior and the Risks of Promoting Obfuscation* — 2025

WHAT TO DO WITH THIS

Pick one of the three above and put a date on it. Then send back the one gate here you think is wrong, and why. A counter-example is worth more to us than agreement — this document is only as good as the arguments it survives.

This brief describes a system built and operated in production at Parslee. Internal names, file references and vendor specifics have been removed; the patterns are the point. Questions, disagreements and counter-examples are all welcome — particularly the counter-examples.

PARSLEE LABS · REV 1.0 · AUGUST 2026 · PARSLEE.AI · © 2026 PARSLEE, LLC