
Agents on Rails: Deterministic AI for Chip Sign-Off

Abhishikta Pandit, Aditi Nigam, Nishanthi Vangipurapu, Anusha Vasan, Olivier Sagner
Meta

Abstract

The most time-consuming work in chip design and verification is judgment-heavy language: interpreting sign-off logs, triaging regression failures, adjudicating waivers, reasoning about coverage holes. Large language models suit this work, but non-determinism and hallucination are disqualifying where results must be reproducible, auditable, and trusted to gate silicon.

We present agentic-AI applications built on a principle we call deterministic by construction: agents run at every stage of a sign-off flow, but no value a gate reads may originate in a model response. *Operational* decisions (is this run worth starting, is this job already doomed) are the agent's, and getting one wrong costs an afternoon. *Sign-off* decisions (the violation count, the verdict) belong to deterministic code and the EDA tool, and getting one wrong ships silicon. We enforce the boundary with per-phase tool allowlists, schema-constrained output, ground truth computed before the prompt and never read from the reply, and a single guarded write path. A model-proposed edit is visible before it lands, reversible after, and re-checked on the next run; flows degrade gracefully without the model. The deterministic infrastructure was itself largely written with AI: the same rule, with code review as the gate.

We walk the lint auto-fix loop end to end, and two performance studies where the measurement leaves the cause open; because enforcement lives in the framework rather than in each flow, the same boundary extends to other flows. The model reasons; deterministic code decides. Agentic productivity with sign-off-grade trust. Familiar tools, new intelligence.

1 Introduction

Sign-off is where a design stops being edited and starts being judged, and the judgment is machine-consumed. A static-verification run that completes with a dirty report exits with a code distinct from one that never ran, and continuous integration branches on the difference. No human stands between that number and the decision it drives.

The tools producing the number are deterministic. Acting on it is not. Which of four hundred log messages matter, why a regression failed, whether a waiver is legitimate or merely convenient: that work is linguistic, scales with headcount rather than compute, and is among the most tedious.

Large language models are good at exactly this. The difficulty is not capability but evidence. A model's output is a draw from a distribution, so two identical runs can disagree, and hallucination makes the disagreement silent. Low temperature, self-consistency and spot review narrow that variance without removing it. None produces the line a release record needs: not "*the model reported this count*" but "*these columns of the tool's summary table were summed.*"

1.1 Determinism as an architectural property

Our position is that determinism should be a property of the architecture around the model, not of the model, and therefore something one establishes by reading code rather than by sampling output.

We state this as a placement rule and build to it: **no value a gate reads may originate in a model response**. The rule does not restrict where agents run. They validate inputs before a flow starts, watch it throughout and stop it early, interpret results when it finishes, and propose fixes afterwards. What it constrains is what they may decide: operational questions are theirs, and sign-off questions (the count, the verdict) never are. In that sense the agents run on rails: they go wherever the flow goes, but flow code lays the track. The EDA tools engineers already trust stay as they are, and the agents add judgment around them.

1.2 What this paper describes

Sections 2–4 give the rule, the mechanisms that hold it, and where agents sit. Section 5 follows one lint run end to end, including a violation the agent declines to fix, and Section 6 two performance studies where the measurement is just as deterministic but does not identify a cause. Section 7 states the limitations. The deterministic infrastructure was itself largely written with AI, which is the same rule at a different timescale (Section 2.5). Where the system is built we say so, and where it is a direction we say that too.

Contributions. Prior work bounds what a model can do; we bound what a gate may read. Concretely, we contribute (i) a placement rule for agents in sign-off flows, with a split between operational and sign-off decisions that lets agents work at every stage without deciding any sign-off value; (ii) the argument, and the mechanisms behind it, that the rule can be checked by reading code rather than estimated by sampling runs; (iii) a guarded write path that makes model-proposed edits visible, reversible and re-measured; and (iv) a real lint run through the architecture, including a wrong prediction that the deterministic count caught.

Terminology. “Tool” and “code” are overloaded here, so we fix usage throughout. The *EDA tool* is the commercial checker. *Flow code* is our infrastructure around it: it runs the tool, parses its output and computes the verdict. The *agent layer* is the prompts, model calls and reply handling, and *the model* is the language model itself. A *model tool* is a function the model may call. Only the agent layer and the model are non-deterministic. A *gate* is any point where a value drives an automated sign-off decision: an exit code, a verdict, a downstream trigger that depends on the design being clean. Starting, stopping or rerunning a flow are operational controls, not gates. A *flow* is one tool-and-judgment pipeline, such as lint or clock-domain crossing, run as a graph of steps.

2 Deterministic by construction

A sign-off agent does not fail when the model is wrong, but when a value a gate consumes originated in a token stream. A wrong suggestion is discarded by whoever reads it; a wrong number written into a report is indistinguishable, downstream, from a computed one. The rule we build to is therefore about placement, not model quality: **no value a gate reads may originate in a model response**. The rule governs provenance, not influence: an edit changes the artifact the EDA tool measures, and a waiver counts only once an engineer accepts it, after which it is the engineer’s, like one written by hand; but no value a gate reads is ever copied from a model response.

2.1 Free in placement, bounded in authority

The rule restricts not where agents run but their authority. A sign-off flow makes two kinds of decision, and conflating them is the mistake the rule exists to prevent.

Operational decisions govern the run. Are the inputs coherent? Is this run worth starting? Is this job making progress, or is it already doomed and worth stopping? Should the flow be rerun? These are questions about cost, time, and machine resources. Getting one wrong wastes an afternoon.

Sign-off decisions govern the design. How many violations are there? Is the report clean? Does this design pass? These are the values a gate reads, and getting one wrong ships silicon.

Agents operate freely on the first kind and never make the second. They can propose an edit, which changes the design the EDA tool measures, or a waiver, which changes what it counts and takes effect only when an engineer accepts it (Section 4.5); the count and the verdict stay the EDA tool’s. That is a larger role than it may first appear: the model reads logs, interprets EDA tool output against methodology, stops doomed jobs and proposes specific edits, work no amount of parsing produces. What it never does is sign anything off. The model may read bounded windows of reports and design

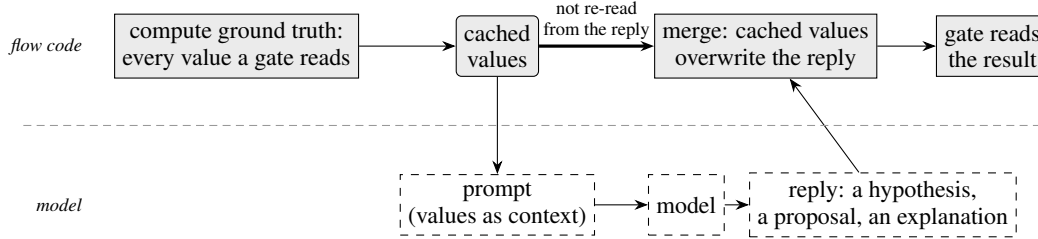


Figure 1. Compute, cache, merge: the gate’s copy comes from the cache, never from the reply.

files, interpret them, and propose: a status from a closed set, a root-cause hypothesis, an edit, a waiver. Flow code alone counts, classifies, parses, applies edits and renders artifacts. The split is by consumer, not difficulty: the test is whether a gate reads it.

2.2 Compute, cache, merge

Enforcement rests on *when* a value is computed rather than on what the reply contains (Figure 1). Ground-truth values are computed before the prompt exists and cached. They are supplied to the model as context, so it reasons over real numbers rather than guessing them. When the reply returns, the gate’s copy of each value is taken from the cache, never recovered from the reply.

A wrong number in the reply therefore cannot survive the merge. It is not detected and rejected; it is simply never read. The model contributes only the fields no gate consumes: the hypothesis, the explanation, the proposal.

2.3 Why this is checkable rather than estimable

Because the property is about where a value is computed, verifying it is a finite reading task. For each field a reply handler returns: is it re-injected from the cache, re-derived against the artifact, or declared advisory? Any other field is a defect, visible to a reviewer the day the handler is written.

Sampling cannot establish this. A hundred identical runs are consistent with a field being re-derived, and equally consistent with the model having been consistent a hundred times. The rule converts a statistical question, which one can only bound, into a structural one, which one can close.

2.4 Bounded change: when the model writes

In the fix agents, model text does reach a design file: an *anchor*, copied verbatim from the file, and replacement text. The EDA tool then re-measures the changed design. The model is an engineer proposing a change, not an oracle reporting a result. It proposes; the deterministic check disposes.

That position is safe because the change is bounded four ways. It is *visible*: every edit is recorded with its file, anchor, replacement and reason before it is made. It is *reversible*: every file is snapshotted before it is touched, and a session replays to the original bytes. It is *re-checked*: the EDA tool re-measures deterministically, so the delta cannot be noise, and a human accepts or reverts against the EDA tool’s numbers. And it is *conservative*: an agent edits within an existing file, at anchors that must each match exactly once, and cannot restructure a design. A waiver, which changes what is counted rather than what is built, takes effect only when an engineer accepts it (Section 4.5).

2.5 The same rule in the development loop

Essentially all of the platform (the execution engine, the report parsers, the validators, the agent framework itself) was written with AI assistance, including the deterministic code that enforces the boundary. Code is not a value a gate reads at run time; it is an artifact reviewed and merged before any run happens. The two uses of AI differ only in *where the deterministic gate sits*. In the development loop the gate is human review, type checking and continuous integration, and variance lands in a code review, where a human decides what merges. In the execution loop the gate is the placement rule, and variance would land in a sign-off artifact, where nothing intervenes.

AI-written code is therefore held to exactly the standard human-written code is: it cleared the same review, type-checking and CI gates, and the agent framework also passed an external architectural review whose findings were closed before it shipped. Parsers, validators and precondition checks are

the least interesting code in any system and the most necessary in this one; generating them cheaply let us put a plain function behind every model tool.

3 Enforcing the boundary

A placement rule is only as good as the points where it is enforced. This section walks one model interaction through five stages (before the call, at the reply, at the values, at the write, at the artifact), each closing a different way a model value could reach a decision.

3.1 Before the call: what the model can see and do

An agent with an open-ended prompt and unrestricted capabilities will eventually read or call something it should not. A phase, the unit of model work, therefore declares its inputs and the model tools it may invoke, and the framework returns only those. The pre-check phase in our lint flow declares exactly two: one that validates inputs and one that reads a file. Phases run in a declared order. Context is capped: file and report reads return a bounded window *and say when they truncated*, conversations have a turn limit, and work is chunked.

Rules need more than an allowlist. If a rule is described in the prompt *and* implemented in flow code, the two drift, and nothing errors when they do. An EDA tool's coding rules are therefore kept as a schema-validated catalog and delivered to the model as structured data, not as prompt prose. With one place to change and read a rule, instructions and implementation cannot drift apart.

3.2 At the reply: shape before content

The tempting way to get a value out of a model is to pattern-match its prose, and the dangerous default is to treat a miss as benign: no matches, therefore no problems, therefore clean. Replies are instead constrained to a declared structure. A phase names the schema its result must conform to, and validation runs before the phase's own code sees the data. A malformed reply is rejected outright, and a checkpoint corrupted on disk is dropped rather than half-trusted. Fields that represent a choice are enumerated; a value outside the closed set is discarded and replaced by one flow code computes itself, so the model's choice is used only when it is valid.

Well-formed replies need checking too. In pre-check, the model returns a status and a list of issues, and deterministic extensions add issues of their own. If the model reports a pass while any merged issue carries error severity, flow code sets the status to fail. The model's judgment is an input to the status. It is never the status. But a schema constrains the *shape* of a reply, not the *truth* of it: a perfectly valid object can contain a wrong number. That is what the next stage is for.

3.3 At the values: the model requests, it does not compute

Counting looks like the safest thing to delegate and is among the least safe, because sign-off numbers are conventions, not facts, and the convention lives only in the code.

Ask a mature flow how many errors a run produced and the log supports several answers. Our static-verification EDA tool summarizes a run as a table: ten error columns by type (syntax, synthesis, connection, structure, simulation, naming, coding guideline and user-defined among them), plus fatals and two columns it classes as informational. The verdict sums eleven columns, the ten error columns and fatals, and skips the informational ones; any non-zero result fails the run. A configuration option adds ten warning columns to the sum for projects that enable it. With or without fatals, the informational columns or the warnings, each total is defensible.

Only one of them is the verdict's, and which one is written down nowhere but in flow code. A model asked to count will produce a number that is plausible, defensible and wrong for its consumer. It does not reason badly; the information needed to pick the convention is not in the report. And the error is silent, because a count off by one column looks entirely normal.

Every model tool is therefore a thin wrapper over a plain function that flow code and the observers also call directly. The wrapper has no logic of its own, so the model cannot compute a value, only request one, and what comes back came from the same function the flow itself trusts. In our lint agent this is one-to-one and exhaustive: five capabilities (validating inputs, polling the job, checking the logs, reading a report, reading a file), each a named function whose wrapper's entire body is a call to

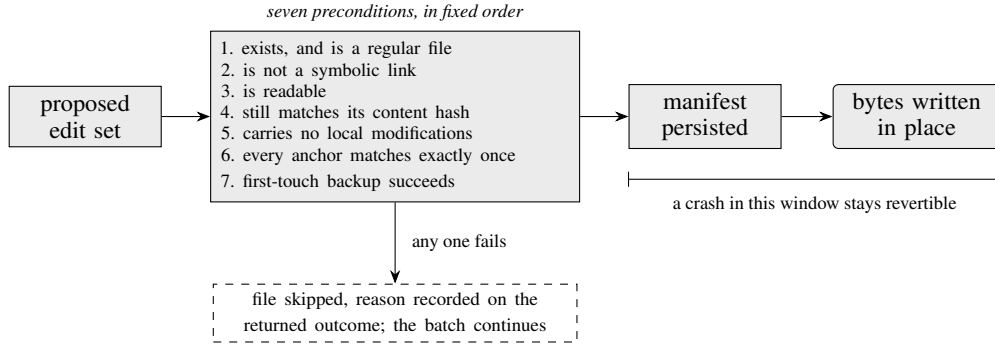


Figure 2. The single write path: seven preconditions, then the manifest, then the write.

it. And ground-truth values are computed and cached before the prompt exists, so the gate reads its copy from the cache, never from the reply (Figure 1).

3.4 At the write: one guarded path

A model proposing text that lands in a design file is the sharpest exposure: a fabricated location, a partial edit, a file changed underneath the agent, a symbolic-link escape, or a crash mid-batch.

Every model-proposed edit reaches disk through one engine, which walks each target file through the seven preconditions of Figure 2 before modifying a byte. Each closes a specific failure: refusing symbolic links keeps edits inside the working area, a content hash recorded at proposal time refuses a file changed since, refusing uncommitted local changes protects an engineer’s work in progress, and anchoring on an in-memory copy makes each file’s rewrite all-or-nothing. A file failing any precondition is skipped with the reason recorded and the batch continues, so a partial apply describes itself. The manifest is written before the first byte changes and kept outside the run directory, so after a crash mid-apply the manifest can still be replayed. If the write itself fails after the checks pass, the engine restores its snapshot: a failed write leaves the original, not a fragment.

Revert replays the manifest. By default a second round of edits to the same file is refused, since the first round’s edits are local modifications; when the engineer asks for another round, the first-touch snapshot is shared across rounds, so several rounds still revert to the *pristine* original. Revert is also drift-aware: a file the engineer has edited since the apply no longer matches the recorded hash and is skipped rather than restored, so undoing the agent cannot silently discard later work. Applying edits has three settings: off (the default), propose-only, and apply.

3.5 At the artifact: making runs comparable

If two runs over identical inputs produce differently ordered output, every diff is noise. Records are written atomically, so a reader never sees a partial file. Key ordering is the one place determinism is not yet by construction: deterministic ordering is available, but each caller has to switch it on. It affects how two runs’ artifacts compare, not any value a gate reads, and it is the first place we would look if runs stopped comparing. Ordering is necessary but not sufficient: artifacts carry run identifiers and timestamps as provenance, so comparison needs a *declared list* of the fields that vary. An exclusion list that grows whenever a comparison fails is worse than no comparison at all.

3.6 What it costs

Every capability an agent needs must be declared, which makes adding one a deliberate act rather than an incidental one; that friction is the point. Every value that matters is computed eagerly, before anyone knows the model will need it. Legitimate edits are sometimes refused: a file an engineer touched between proposal and apply is skipped even if the edit was fine. And the list of fields allowed to vary between runs is policy somebody has to maintain honestly. We prefer each of these to the alternative.

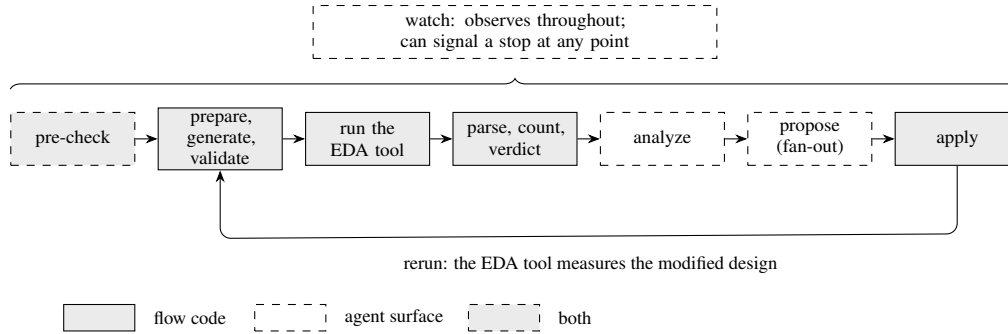


Figure 3. A flow's lifecycle and its agent surfaces; the watcher spans the whole run.

4 Where agents sit

Agents operate throughout a flow's lifecycle (Figure 3), with a surface at each point and one before the lifecycle begins. What follows is where the model does work, and what stays with flow code.

4.1 Before there is a flow: authoring and migration

Before any run, an engineer has to express their design in the platform's input format, or carry a working configuration across from the previous flow infrastructure. Three assistive skills help, invoked on demand rather than in a run: one answers platform questions and composes command lines, one generates a flow's input file, and one converts an existing configuration.

The input generator holds the most authority, because a wrong input file produces a run that is perfectly reproducible and measures the wrong thing. It works from the declared input schema, and **it asks rather than infers**: it searches the design area for candidate values, but where a field is ambiguous or load-bearing, such as the design's top-level name, it asks the engineer instead of choosing. The engineer reads and commits the result like any other file, and pre-check validates it on the first run. An agent helped write it; nothing downstream trusts it on that account. Teams can also write their own skills (Section 4.7).

4.2 Before the run: input validation

Pre-check runs before a flow, where it is enabled, and establishes that the run is coherent: filelists resolve, referenced files exist, configuration is well formed. Catching a malformed input here, rather than hours into a job holding a license and reserved cores, is the economic case. Most of this checking is deliberately deterministic, but the surface accepts agentic pre-checks where judgment helps, such as noticing a combination of options that is self-defeating in a way no single rule captures. It can be as agentic as the problem warrants because it never judges the design: it decides whether the run is worth starting, and a deterministic error-severity finding overrides its pass (Section 3.2).

4.3 During the run: watching

Once a flow starts it is committed to finishing, unless something is watching. The watch surface runs from pre-check until the flow ends, alongside the flow, not as a step in it, looking for evidence that the run will not produce anything useful: a pattern in the emerging log, a resource trajectory that will not finish in its window, an error that will not resolve itself. A stop it signals also stops everything depending on the flow, returning licenses, machines and attention hours early. The decision is operational: not a view about the design, a view about whether to keep paying for it.

4.4 After the run: interpretation

When a run completes dirty, interpretation begins. Forty violations are frequently not forty problems but four root causes, and recognizing that means reading the violations against one another and against design intent: the linguistic work that motivated the whole exercise.

Flow code still parses, counts and classifies; the model reads across the findings, proposes a root cause and explains it, and neither substitutes for the other.

4.5 Acting on the run: proposals

The fix surface proposes an edit to a design file, or a waiver for a finding that should not have been flagged; neither takes effect because the model said so. An edit travels the guarded write path and is re-measured on the next run. A waiver enters through the same input as a human's, so by the time the EDA tool sees it, nothing says who wrote it. It is also the one proposal that can lower a count without changing the design, so it takes effect only once an engineer accepts it; the agent's waivers arrive in their own file, with their justifications, so review can tell them apart.

Across all of these surfaces, an agent can help an engineer express a run, start it, stop it, skip it, explain it, and propose a change to the design under test. It cannot say whether the design passes, and a run it stops or skips can never report a pass.

4.6 When the model is absent

A sign-off flow runs against a schedule that does not care whether a model is available, so outages, exhausted quota, or switching the agent layer off for a tapeout week must be survivable without editing a flow. The deterministic path *is* the flow; the agent layer is an addition to it. A plugin in the flow's own directory opts it in, and one environment variable turns the agent layer off before any model client exists. With the model unreachable, the plain functions behind model tools still work, steps degrade and proceed, and a circuit breaker stops the retries. Observer callbacks have their exceptions caught, so a broken agent cannot reach the flow's control path.

The worst case is therefore a run that behaves exactly as it did before the agent layer existed: same EDA tool invocation, same parse, same verdict, same exit code. Runs that complete with the agent layer off and on should agree on every value a gate reads, and any disagreement is a defect with a location.

4.7 Adding a surface

A discipline maintained by its authors is not a discipline; it is a habit. What decides whether this outlasts its team is what happens when someone else extends the platform, which is open at four points: a new flow, a new agent surface, an assistive skill, or a new server exposing model tools. None of the enforcement in Section 3 is implemented per flow; the framework's schema check, its tool allowlist and the one write engine apply it to anything new, so the author of a new surface gets the mechanical part of the rule without needing to know it.

This inverts the usual trade. Ordinarily every extension point is a place a guarantee can be violated, so a system wanting strong guarantees stays closed; here the guarantees are structural, so opening the platform costs little. The limit is judgment: a contributor can still write a reply handler that takes a value from the model where a deterministic function was available, and today code review, not an automated check, catches that (Section 7). And a new flow inherits the boundary only for work that goes through the framework; its own step code is reviewed like any other.

5 A lint run, end to end

We walk one real lint run through the architecture; lint is where the loop is complete, and the same mechanisms serve the other flows. Names, paths, rule identifiers and process references are replaced; no counts are altered. Table 1 lists the run's values, by who produced them.

Table 1. *One lint run: the EDA tool and flow code beside the model. † marks values a gate reads.*

	EDA tool and flow code	Model
Findings	0 fatals, 23 errors, 390 warnings [†]	which types of error matter
Error types	7 synthesis, 11 guideline, 1 naming, 4 structural	waive the 11 guideline errors, if approved
Verdict	fail, before any agent reads it [†]	predicts a pass once the other 12 are fixed
Rerun	5 errors, 93 warnings [†]	59 anchored edits; vendor IP errors waived
Left over	attempted, not cleared: 3 errors, 28 warnings	held back on purpose: 2 errors, 65 warnings
Time	EDA tool 9 min (two runs); apply under 1 min	analysis 80 s; proposals 7.5 min

5.1 The run

A lint flow runs on a sequencer block, following Figure 3: pre-check passes in seconds, the EDA tool runs, and flow code computes the verdict, on which continuous integration branches before any agent reads it. Those counts and that verdict stay the parser's for the rest of the run.

5.2 Analysis

The model reads the EDA tool's split of the errors by type and does something a rule table cannot. It recommends waiving the eleven coding-guideline violations, which the configuration raises to errors but which do not change what the circuit does, and leaves that decision to the design engineer. It recommends *against* waiving the seven synthesis and four structural (netlist and topology) errors, which carry real functional risk. The one exception is vendor IP, which may not be edited at all (Section 5.3). That line turns on what the violations mean, not how many there are. At this stage it also predicts that fixing the other twelve errors should produce a pass.

5.3 Proposals

The fix stage fans out, one worker per file, each returning a structured proposal of anchored edits with their rationale. Two are worth contrasting.

An edit that requires arithmetic. A bus index is cast to seven bits where the range it indexes needs only six, so the fix changes 7' (WIDTH*idx) to 6' (WIDTH*idx). One character, and not a substitution a pattern could safely make: getting it right means working out the index's maximum value from the surrounding declarations.

A violation the agent refuses to fix. Another module reports five violations under two rules. The agent proposes no edit: the module is a vendor-supplied foundry cell, and vendor IP is not modified. It emits a waiver instead. That rule is methodology, not syntax. It appears in no report, no schema and no violation message. An agent that fixed those five violations would have produced five clean, correct, entirely unacceptable edits. It held back elsewhere too: at the fix stage it left two errors and 65 warnings untouched on purpose, because fixing them could have changed what the circuit does (Section 2.4); the write path does not check function, so that stays with the reviewing engineer. Across the run, it edited less than it declined to edit.

5.4 Apply and rerun

Each file passes the seven preconditions of Figure 2, and apply writes a manifest, a plain-text fix log for the engineer, and a waiver file. The engineer accepted the guideline and vendor-IP waivers, and the flow reran. The agent had predicted a pass; the EDA tool found 5 errors and 93 warnings, down from 23 and 390: the two errors and 65 warnings the agent held back, and three errors and 28 warnings its proposals did not clear. The prediction was wrong, and the EDA tool's count is what says so. Because that measurement is deterministic, the difference between runs is a difference in the design: the engineer starts the next round from numbers, not from the agent's confidence, with a one-command revert at hand.

5.5 What it replaced

Analysis and proposals took **about nine minutes**, and the apply under one more. Together they stand in for triage and first drafts: reading 390 warnings and 23 errors, grouping repeats, checking candidates against methodology, drafting fixes and waivers. This is agent-side time only; no manual baseline was instrumented, and review remains the engineer's work.

6 Two performance studies

This section walks through two studies in performance analysis. In each, the issues met and their resolution come back to the same thing: making sure the path taken is predictable and respects the constraints. Both studies are real, with design details removed, and are described from practice rather than a captured run, so this section gives no counts.

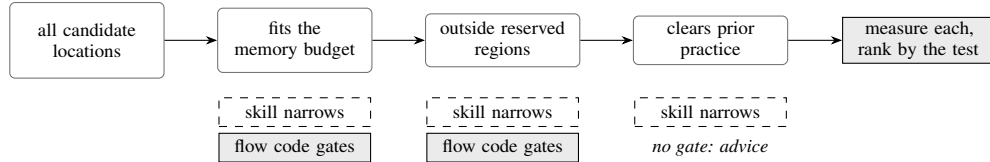


Figure 4. *The sweep narrows: skill at every step, gate on budget and reserved; sizes are not counts.*

6.1 A figure under target

A directed performance test runs and reports its counter reads. Flow code compares each read with a set target value. A mismatch between the two is a failure, and that much is deterministic: the comparison is arithmetic, and nothing about it is the model's.

What the failure does not carry is a cause. A read under target can mean the structure is too small, the stimulus is unrepresentative, or the target assumed something else. The number says the design missed. It does not say why, and no parser can tell the explanations apart.

Supplying that cause is the model's job, and a skill is what makes it systematic. The skill carries the procedure: follow the data path stage by stage, isolate the issues met along the way, quantify the delta each one contributes (arithmetic on the reads that informs the recommendation; no gate reads it), and arrive at a design fix recommendation. A full queue may mean a producer bursting harder than the path was sized for, an arbitration point starving one requestor, or backpressure from several stages downstream. Two of these give the same occupancy and call for opposite fixes: enlarging a buffer whose producer is bursty costs area and buys little. Attributing a delta to each stage is what separates them. Where the evidence fits more than one explanation, the model says so and names the measurement that would settle it. It never revises a measurement.

6.2 A placement sweep

The flow sweeps candidate locations for a buffer in on-chip memory and measures each with the same test. Each costs a run, so choosing which to measure is the real problem. An installed skill (Section 4.7) carries three kinds of constraint into the search. The first is the memory budget. The second is reserved regions, kept for reasons that have nothing to do with performance. They are removed from the search, not scored badly within it. A constraint that a good enough measurement can outvote is not a constraint. The third is accumulated practice: placements that were legal on earlier designs and caused trouble anyway.

None of this makes the skill a safety mechanism. Flow code checks the budget and the exclusions too, separately, before any candidate is measured and again before anything is applied (Figure 4). The skill narrows the search. The gate decides what is allowed. The knowledge makes the search good. The check makes the result trustworthy.

The candidates that clear all three constraints are then measured rather than argued over. Each is re-run on the same test, and the ranking is the measurement's: the model's predicted best gets no special standing, and if it comes second, it came second. Because the test repeats, a gap between two placements is a real difference and not noise. Otherwise the ranking could not be acted on at all. The analysis repeats for a different reason: the skill pins the procedure, so the same path is walked in the same order with the same deltas attributed at each step.

6.3 What it buys

The procedure is fixed, so similar issues arrive at the same design fix, and data paths stay predictable rather than shaped by whoever happened to look at them. It also saves engineers time, because what the skill records is shared among them without anyone having to communicate it explicitly. As the project evolves and buffers are added or grow, each change is re-evaluated under the same constraints and inherits the allocation reasoning already recorded, instead of being argued from scratch. The skill bounds the problem, so the outcome is reproducible and can be validated. None of it depends on the engineers who learned those constraints still being in the room.

7 Limitations

Argued, not measured. The boundary holds because of how the code is written, and code review keeps it that way; nothing yet checks it automatically. Nor have we shown it empirically, by running identical inputs repeatedly with the agent layer on and confirming that every gate value matches while the model’s prose varies.

What the rerun does not check. The write path establishes that an edit is well formed, correctly located and applied to the right file, not that the edit is right. The rerun catches a wrong edit only coarsely: one that lowers a count while changing behavior the EDA tool does not model will pass. The final judgment is the reviewing engineer’s, by design.

Waivers. An accepted waiver hides a finding rather than fixing it, so the rerun cannot catch a bad one. Agent-authored waivers get the same human review as hand-written ones, and that review is the only safeguard.

Adversarial input. The model reads design files, logs and reports, so text planted in any of them could try to steer it: a comment asking for a waiver, or a misleading log line. The boundary limits what that can achieve. Injected text cannot change a count or a verdict, because those never come from the model; at most it shapes a proposal, and every proposal is either re-measured by the EDA tool or, for a waiver, accepted by an engineer. What it can still do is waste review time, or slip a plausible waiver past a careless reviewer, hence the waiver review above.

Time. Section 5 reports agent-side time only; the agent layer does not yet record cost or latency.

8 Conclusion

Most uses of a language model are judged by a person or a pass rate, both of which tolerate variation between runs. Sign-off inverts that: the output is a decision, the consumer is automated, and “usually correct” and “different each time” are not two problems but one.

Our response is a placement rule rather than a quality target: no value a gate reads may originate in a model response. A rule that improves as models improve is a rule that was never load-bearing, so this one is deliberately indifferent to how good the model is. Agents are free in placement and bounded in authority. Where the model writes, something other than the model checks the result: the EDA tool re-measures an edit, and flow code the model helped write passes human review and CI before it ever runs.

What that buys is a claim someone else can check, by reading a finite set of reply handlers against one rule rather than sampling a thousand runs. A team building something similar does not have to take our word for any of it. They have to read their own code.

The model reasons; deterministic code decides. Familiar tools, new intelligence.