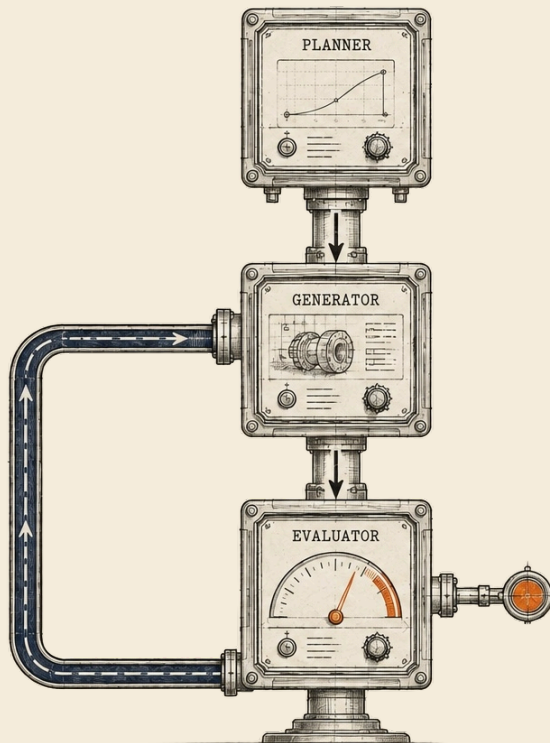


Evaluators Let You Automate Your Coding Agents



SPECSTORY PRESS

What we're trying to do

We want to *consistently* get more out of our coding agents. Two goals drive everything in this document.

Make quality repeatable. We do not want one good result from one clever prompt. We want a system where quality is a property of the outer harness, meaning the system of prompts, tools, and checks that surrounds the agent. Then every run benefits from every lesson already learned, no matter which engineer starts the run or which repo it touches.

Let agents run longer, on harder tasks, with less supervision. An agent is worth more the longer it can work unattended. The limit is not what the model can do. The limit is how much we trust the output. Datadog states the shift plainly. Agents can now produce software faster than any team can verify it. The slow step has moved from writing code to trusting what was written.

Using evaluators can support both of these needs. They let us move from that feeling of “the agent stopped, I wonder what it did” into confidence that “the work is done”. Evaluators are also needed if we want to let the agent keep going past its natural stopping point. Anthropic’s experiments, detailed below, show how strong this dependency is. Without an evaluator, a six-hour autonomous build produced an app whose main feature simply did not work, and nobody knew until a human clicked through it. With an evaluator, the same kind of build shipped working software, because failures were caught and **fed back during the run** instead of after it.

One fact underlies all of this, and it is the core reason evaluators exist as a separate component. **Agents cannot reliably grade their own work.** Anthropic saw this directly. When agents were asked to evaluate output they had produced, they praised it with confidence, even when a human could see the quality was mediocre. The problem is worst on subjective tasks such as

design, but it appears even on tasks that can be checked. Separating the generator from the evaluator does not automatically remove the bias. The evaluator might still use a language model, and language models tend to be positive. But the separation makes it possible to optimize for strong evaluation. It is far easier to tune a separate evaluator to be skeptical than to make a generator criticize its own work. And once outside feedback exists in concrete form, the generator has something to improve against.

The framework: Planner, Generator, Evaluator

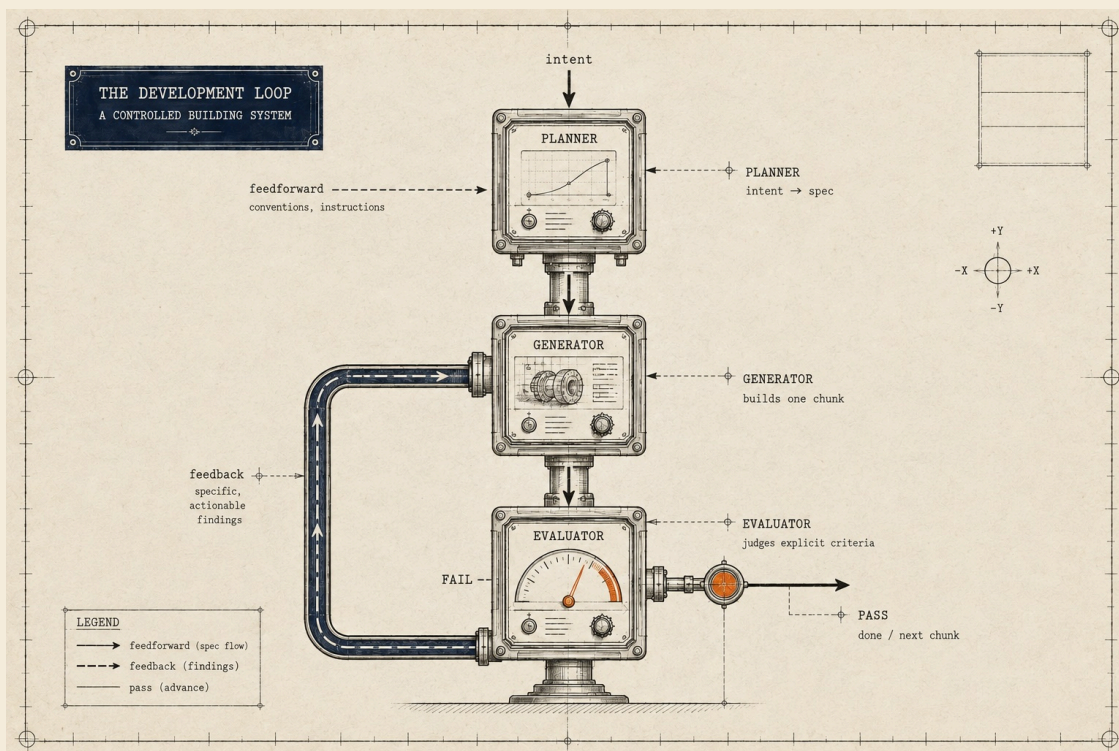
Evaluators are one component in any agentic coding system. Though they're sometimes given different names and are run in different ways, every example in this document uses the same three roles in a loop:

- **Planner.** The planner turns intent into a spec. It stays at the level of product goals and high-level design on purpose. If it fixes technical details too early and gets one wrong, the error spreads into everything built on top of it. So it constrains what must be delivered and lets the generator choose how to build it.
- **Generator.** The generator does the work, one manageable chunk at a time. It checks itself as it goes, but it never gets to declare its own work done.
- **Evaluator.** The evaluator is a separate agent or system. It judges the work against explicit criteria. It either passes the work or sends it back with findings specific enough to act on.

Real world experimentation gives us two refinements that are worth considering from the start.

The sprint contract (Anthropic). Before any code is written, the generator and the evaluator agree on what “done” means for the chunk. The generator proposes what it will build and how success will be verified. The evaluator challenges the proposal. They go back and forth until they agree, and they communicate through files. This connects a high-level spec to a testable implementation without fixing details too early. It also gives the evaluator a concrete contract to grade against instead of a feeling. One of Anthropic's sprints carried 27 testable criteria for a level editor alone.

Reset expectations for your evaluator over time (Anthropic). Models and commercial harnesses (like Claude Code running Fable 5) are improving



all the time. Assess your evaluators periodically to see if they’re doing necessary work and doing it at the right times. With past models, evaluation might have been needed at each step of an agent’s plan. Now, it might make more sense to run evaluations only at the completion of a plan or when a generator claims to have achieved a goal. Each new model release breaks some of the assumptions baked into your loop. So on each release, recheck where the evaluator sits, and what prompts and tests it runs.

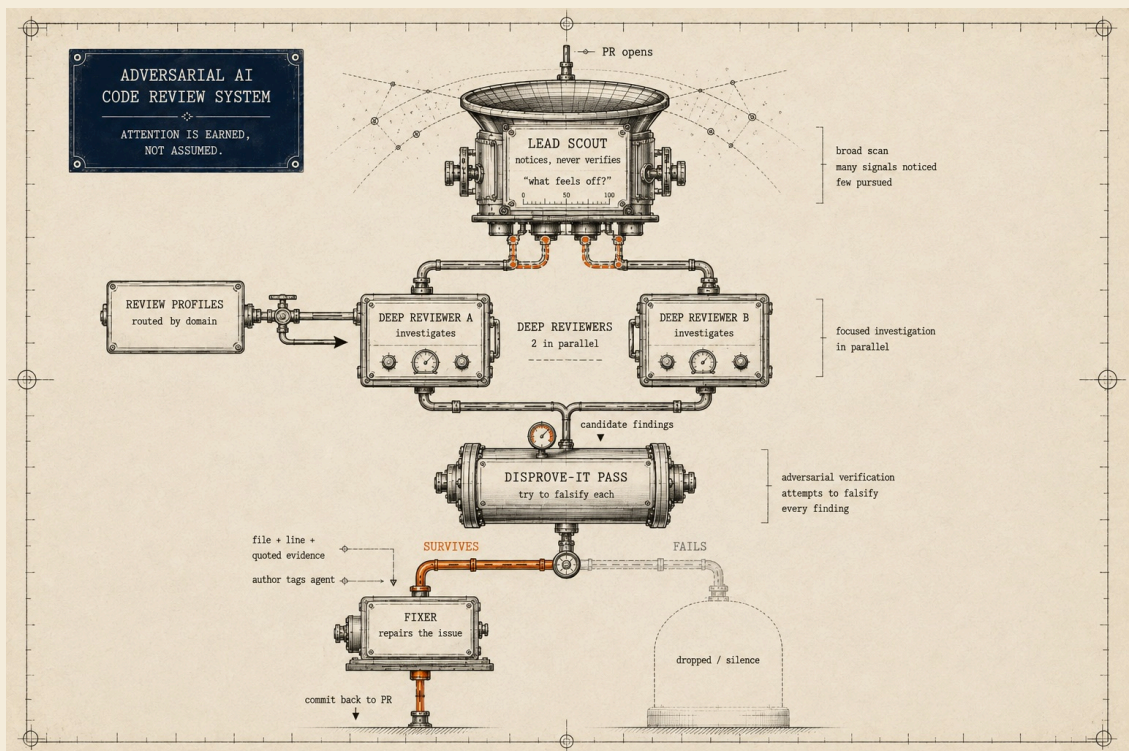
The rest of this document describes the three types of evaluators that appear across the sources.

	REVIEW EVALUATOR	PRODUCT EVALUATOR	COMPUTATIONAL EVALUATOR
Judges	The change (a diff or PR)	The built product, by using it	Invariants and contracts
Mechanism	A language model that looks for faults. A scout flags risky spots, then reviewers verify each one and try to disprove it	A language model that uses the running app and grades it against explicit criteria	Deterministic checks such as simulation, property tests, proofs, and telemetry
Execution type	Inferential	Inferential	Computational
Output	Findings on a PR	A graded verdict and a bug list on working software	A pass or fail result that can be reproduced
Source of depth	DoorDash	Anthropic	Datadog

SECTION 3

The Review Evaluator (DoorDash)

What it is. A system built on language models that judges changes. It reads a pull request (PR) the way a senior engineer would. It forms hunches about what is risky, investigates only those hunches deeply, and posts only the findings that survive an attempt to disprove them. DoorDash runs it on more than 10,000 PRs per week across 56 repos. The repos hold several kinds of code: Go backends, mobile apps for iOS and Android, web frontends, infrastructure code, and data pipelines. The system starts automatically when a PR opens and usually posts about 7 minutes later, which is often before any human has looked at the change. A review costs about \$3 on average.



The architecture, and why it took three tries

The current system is the third version. Each earlier version failed in a way the team would otherwise have had to discover again.

Version 1 used specialist agents. There was one agent for each concern: security, tests, performance, and code quality. Each had a narrow checklist. This was good at mechanical bugs: missing nil checks, unhandled errors, and test gaps. But it was blind to architectural problems. A refactor that quietly changed a contract slipped past it. So did a deletion that broke something three repos away. Specialist agents miss problems that span the whole change, because no single agent looks at the whole change.

Version 2 used parallel generalists. Two general-purpose reviewers each read the whole change. They were better at problems that cross boundaries. But each one had too much to do in one session. It had to read the full diff and apply every relevant rule. It also had to trace callers and check sibling files, and then verify every concern it raised. Attention spread too thin, and genuine findings got lost. The deeper problem was that the system had not decided what deserved deep investigation.

Version 3 added a lead scout. The scout is a front agent whose job is not to verify anything. It reads the diff and notes what feels wrong, e.g., “this deletion looks suspicious,” or “this enum case isn’t handled in the sibling file,” or “this error path silently swallows failures.” The scout produces a list of leads to investigate. Two deep reviewers then take those leads. They verify the leads that hold and drop the ones that do not. The scout does two useful things at once. It produces a list of suspect spots, and it filters out the parts of the change that need no scrutiny. By the time the deep reviewers run, they focus on the few places where something might be wrong, and they use the rest of the change as context.

The design lesson is to separate noticing from verifying. That is how strong human reviewers work. They form a hunch first and then investigate it. DoorDash says this change made the largest difference. Their summary of the product lesson is this. Code review agents do not fail only because models make mistakes. They fail because they investigate the wrong things and use up the author’s trust too fast.

Precision over recall

The single biggest decision was refusing to try to catch everything. The common approach posts every concern, ranks them, and lets the human filter. That approach predictably produces PR spam, then muted notifications, then ignored comments. Instead, before any comment posts, the system runs an explicit step where it tries to disprove its own finding. If the claim does not hold up, the system drops it. The result is fewer comments than a naive approach would post. Each comment is anchored to specific lines and includes quoted evidence. Teams keep enough trust in the reviewer that they turn it on voluntarily. DoorDash’s own words belong in every evaluator design doc.

“We’d rather miss some issues than become noise. A reviewer that’s muted catches nothing.”

The reviewer gets its own context

The insight that changed their results is that AGENTS.md and CLAUDE.md files are written for engineers who write code, not for engineers who review it. Those files mix four kinds of content into one document: setup, architecture, patterns, and style. That document is useful for writing and noisy for judging. So the review evaluator gets its own layer of context. For each domain there is a review profile. The profiles are built from four sources:

- AGENTS.md files, keeping only the invariants a reviewer needs
- historical PR comments, with extra weight on what senior engineers flag again and again
- Slack decisions and “don’t do X” threads that never made it into docs
- incident history

Every candidate rule must pass three tests. If CI would already catch it, drop it. If the language model knows it from general training, drop it. If you cannot point to a concrete file and line as evidence, drop it. What survives is truly specific to the organization. Each rule carries a severity and a description, plus evidence like “PR #48291 changed AccountId semantics, breaking downstream analytics readers; reverted in PR #48904; pattern recurred in #50177 and #50632.”

Then the system routes. A PR that touches the payments gateway loads payment rules and nothing else. A feed change loads a completely different set. The evaluator that reviews fraud code is in effect a different reviewer than the one that reviews pricing. DoorDash credits routing for acceptance holding steady across 56 repos that differ widely. The agent does not apply one universal standard. It applies the standard that fits that change.

What it catches that humans miss

The goal is not to replace human review. The goal is to catch the class of issues that humans find hard to catch. These issues do not look suspicious at a glance, but they turn into incidents.

- **Deletions.** Humans skim deleted code. Additions look dangerous and deletions look like cleanup. But removing a struct field can silently change runtime behavior while everything still compiles. Removing a config flag or an interface method can do the same. The evaluator treats every deletion as a question. Who depended on this? What was true before that is not true now?
- **Drift across boundaries.** A PR updates one side of a boundary, and the matching pieces on the other side go stale. The updated side might be any of these: one brand’s adapter, one of two producers, or one handler of an enum. CI never sees the problem because each side compiles on its own.
- **Silent behavior changes.** Some API changes do not break the signature. Error handling can quietly swallow more cases than before. A cache miss can start being treated as an error. The evaluator reads the surrounding code, not just the changed lines, and asks what changed that the diff does not show.

From finding to fix

A finding still creates work, so the same system closes the loop. Anyone can reply to a review comment and tag the agent, e.g., “handle the nil check here” or “resolve the merge conflicts.” A fixer agent then runs in a remote VM with a full checkout. Its prompt is the reviewer’s entire context. That includes the diff and the finding, along with the surrounding code and the suggested direction.

It pushes a normal commit back to the PR. The commit goes through the same CI and human review as any other change. Multiple fixes on one PR run at the same time in isolated worktrees. Nobody has to switch branches or rebuild the context of the issue from scratch. The point is not to remove ownership. The point is to remove the manual handoff between “the review found something” and “someone has to stop what they’re doing to patch it.”

Lessons from production

- **A turn counter is not a progress detector.** An agent that is stuck in a loop does not advance its turn count. The team had one timeout for the whole run, and a single stuck agent still ran for 20 minutes and wasted all of those tokens. The fix was a soft timeout per agent, backed by a hard cutoff that kills the run. When the soft timeout fires, the system interrupts the agent and tells it to stop investigating and to return only the findings it has already verified, without anything speculative. That turns a wasted run into a bounded result that is still useful.
- **The cheapest model is not the cheapest review.** Weaker models produced invalid structured output and had to retry. That cost more than stronger models that succeeded on the first try. The unit that counts is cost per successful review. Because the workflow has stages, cost can be tuned. Cheap models handle simple steps. Strong models handle the steps that verify findings. Low-risk PRs skip some passes. A test suite for the reviewer itself guards against quiet drops in quality.
- **A correct finding can still be a bad comment.** Broad summaries cost trust. So does weak language, e.g., “consider checking...”. So do duplicate comments across repeated reviews. These habits cost trust even when the concern is valid. Comments that work are anchored to a changed file and line. They explain the concrete behavior at risk, and they tell the author where to start. If a comment has no action point, keep it out of the review.
- **Reporting needs its own checks.** The system checks that it never posts a clean review when the analysis found issues. It reconciles stale findings when the PR changes during a review. It collapses old comments when it

reviews again, so the author sees the current state instead of a growing pile of outdated bot feedback.

The evidence

- Of the 2,256 high and critical findings that reached a decision, 60.2% led to code changes before merge. Their previous third-party tool had 46% acceptance. Findings that came from automatic webhook runs, with no hand picking, were accepted 59.0% of the time.
- DashBench replays 105 historical PRs. On that replay, the staged system found 504 genuine findings at 53.6% weighted recall, meaning it found that share of the known issues, weighted by severity. The strongest baseline that reviews in one pass found 164 findings at 30.7%. That is about three times the genuine findings at comparable precision, meaning the share of findings that were correct, at 87.0% versus 84.1%. The stages add coverage, and the added cost and latency are visible and worth it.
- On critical severity issues, the best staged pairing reached 80.0% recall. The baseline that reviews in one pass reached 37.5%. That is more than double on the bugs that count most.

The evaluator needs its own evaluator

DashBench exists because the obvious metric misleads. Acceptance rate only tells you about two of the four possible outcomes. An accepted comment counts as a true positive, and a rejected comment counts as a false positive. Acceptance can never see the false negatives, meaning the bugs the reviewer missed. It can never see the true negatives, meaning the times silence was correct. It also assumes the human is the source of truth, and humans are not. People accept and reject comments for reasons of their own: timing, urgency, ownership, or how invasive a fix would be.

DashBench's answer is to combine several imperfect sources of truth. It replays frozen historical PRs. It judges findings against three imperfect sources: the original authors' annotations, production feedback, and a language model judge. Humans resolve the disagreements by hand, and the resolutions are fed back to correct the judge's future grading.

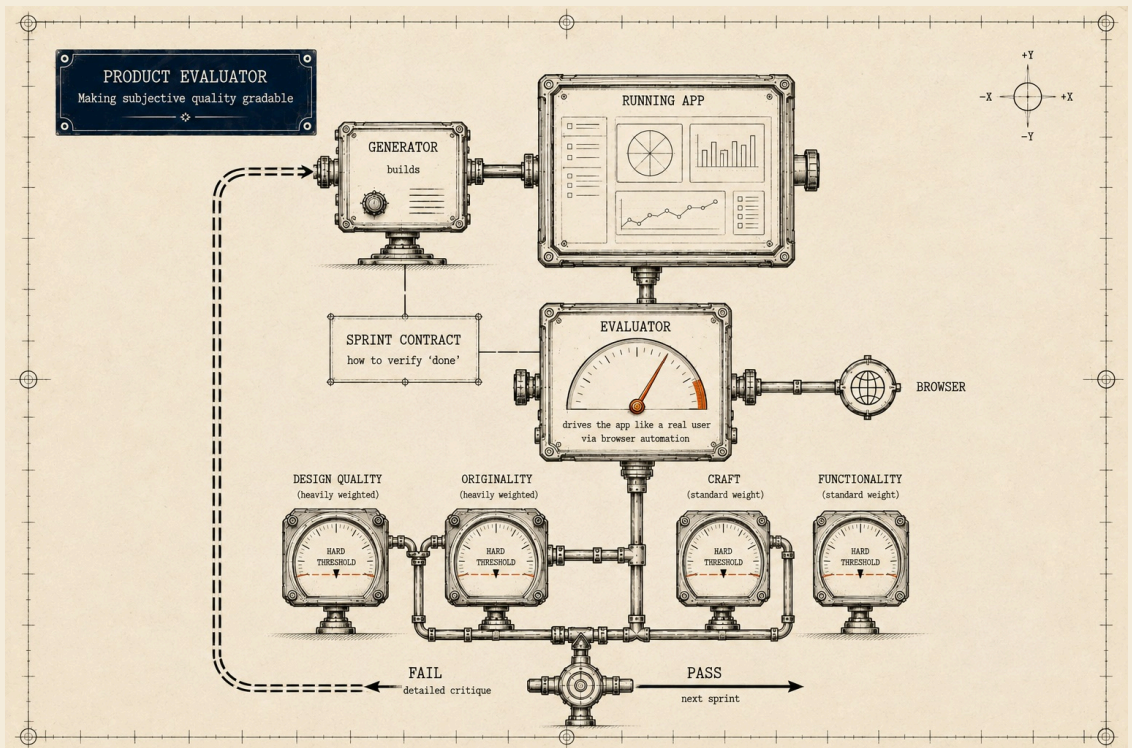
Their key practices are these:

- Include clean PRs, so the benchmark measures restraint. A reviewer that is loud on clean code is its own failure mode.
- Include PRs that were reverted or hotfixed, so actual regressions are testable.
- Run each configuration several times, because a single run understates how much coverage varies. Their phrase for this is “variance is a feature, not a bug.”
- Weight by severity, because two configurations that look close on an aggregate score can miss different kinds of issues.

Day to day, a small curated set of past misses and incidents is the development loop. Every change runs against that set before it reaches an engineer. That includes changes to prompts and profiles, and changes to retrieval and models. Production acceptance confirms the results later.

The Product Evaluator (Anthropic)

What it is. A language model that judges the built product by using it. It operates the running application through a browser the way a user would. It grades the app against explicit weighted criteria. It fails work back to the generator with findings specific enough to fix without further investigation. It answers a question the Review Evaluator cannot answer. The Review Evaluator asks whether a change is safe. The Product Evaluator asks whether the product is good and whether it works.



Making subjective quality gradable

The work started on frontend design, where the failure of self evaluation is easiest to see. Left alone, models produce layouts that function but look safe and unremarkable. Then the models praise those layouts. Two insights shaped the fix.

First, aesthetics cannot be reduced to a score, but they can be improved against criteria that state principles. “Is this design beautiful?” has no consistent answer. “Does this follow our principles for good design?” can be graded. Anthropic wrote four criteria and gave them to both the generator and the evaluator:

- **Design quality.** Does the design feel like a coherent whole with a distinct mood and identity, rather than a collection of parts?
- **Originality.** Is there evidence of deliberate custom decisions, rather than template layouts and library defaults? Does the design avoid patterns that look AI generated? Unmodified stock components fail this criterion outright. So do telltale signs of AI output, e.g., purple gradients over white cards.
- **Craft.** This covers typography hierarchy, spacing consistency, color harmony, and contrast. It sets a minimum for execution. It does not measure creativity.
- **Functionality.** This is usability apart from aesthetics. Can users find the primary actions and complete tasks without guessing?

The weighting is the strategic decision. Design quality and originality get heavy weights, because the model already passes craft and functionality by default. The weights push the model toward taking aesthetic risks and away from what the team calls “AI slop.”

Second, separate generation from grading, and give the grader tools. The evaluator got the Playwright MCP, a tool that lets it control a browser, and it used that tool. It navigated the live page on its own. It took screenshots and studied the implementation. Then it scored each criterion and wrote a detailed critique that fed the next generation. Runs went 5 to 15 iterations and up to four hours of elapsed time. The generator also had to make a strategic decision after each evaluation. If scores were improving, refine the current direction.

If not, switch to an entirely different aesthetic. That instruction produced the best result of the whole effort. On the tenth iteration of one run, the model rebuilt a museum site as a navigable 3D gallery. Nothing that inventive had ever come from generation in a single pass.

Three observations from these runs are worth keeping:

- **The wording of the criteria steers output in ways you will not predict.** A phrase like “the best designs are museum quality” made many designs converge on one style. The rubric is itself a guide for generation, not just a grading sheet. Write it knowing the generator will optimize for its exact wording.
- **Improvement is not steady.** Later iterations were better as a whole, but a middle iteration was often better than the final one. Keep the intermediate versions and let a human pick.
- **Even the first iteration beat the baseline.** The criteria alone steered the model away from generic defaults before any evaluator feedback arrived.

The evaluator as the QA function

Anthropic then applied the evaluator to complete application builds, where it became the quality assurance function. It clicked through the running app. It tested the app at three levels: UI features, API endpoints, and database state. It graded each sprint against the negotiated contract, plus criteria adapted for this work: product depth, functionality, visual design, and code quality. Each criterion had a hard threshold. If any one score fell below its threshold, the sprint failed and went back to the generator with detailed feedback.

The findings were precise enough to act on without any investigation. These examples come from their logs:

- The contract said the rectangle fill tool fills a region on click and drag. The evaluator failed it. The tool only placed tiles at the start and end of the drag. `fillRectangle` existed but was never triggered on `mouseup`.
- The contract said the user can delete placed entity spawn points. The evaluator failed it. The Delete handler at `LevelEditor.tsx:892` required both `selection` and `selectedEntityId`, but clicking an entity only set the latter. The condition should have been an OR.

- The contract said frames reorder through the API. The evaluator failed it. PUT /frames/reorder was defined after the /{frame_id} routes, so FastAPI parsed “reorder” as an integer and returned 422 errors.

That is the standard a useful product evaluator should meet. It names the file and the line. It states the root cause and suggests a direction.

One agent alone versus the full harness

Anthropic ran the same one-sentence prompt twice. The prompt was “create a 2D retro game maker with a level editor, sprite editor, entity behaviors, and playable test mode.” One run used a solo agent. The other used the full harness with a planner, a generator, and an evaluator.

	SOLO AGENT	FULL HARNESS (PLANNER, GENERATOR, EVALUATOR)
Duration and cost	20 min, \$9	6 hr, \$200
Result	Looked plausible. The layout wasted space and the workflow was rigid. The game itself was broken. Entities did not respond to input, and nothing on the surface showed why	Richer editors, a coherent visual identity, and built-in AI features. Play mode worked

The harness cost twenty times as much. But the difference was not polish. It was the difference between software that worked and software that did not. And the failure in the solo run was invisible until a human dug into the code. Anthropic later ran a simplified harness on a build of a browser DAW, meaning a digital audio workstation. That build took about 4 hours and \$124. The QA rounds still caught gaps that the generator had approved on its own. Clips could not be dragged. Audio recording was only a stub, meaning placeholder code that does nothing, behind a button that toggled. Effect “visualizations” were just numeric sliders. The generator remained prone to stubbing features and reporting them as done. The evaluator remained the thing that caught it.

Tuning the evaluator to be skeptical

With no tuning, Claude is a poor QA agent. In early runs, Anthropic watched it identify legitimate issues, then reason its way to deciding they were minor, and then approve the work anyway. It also tested superficially instead of probing

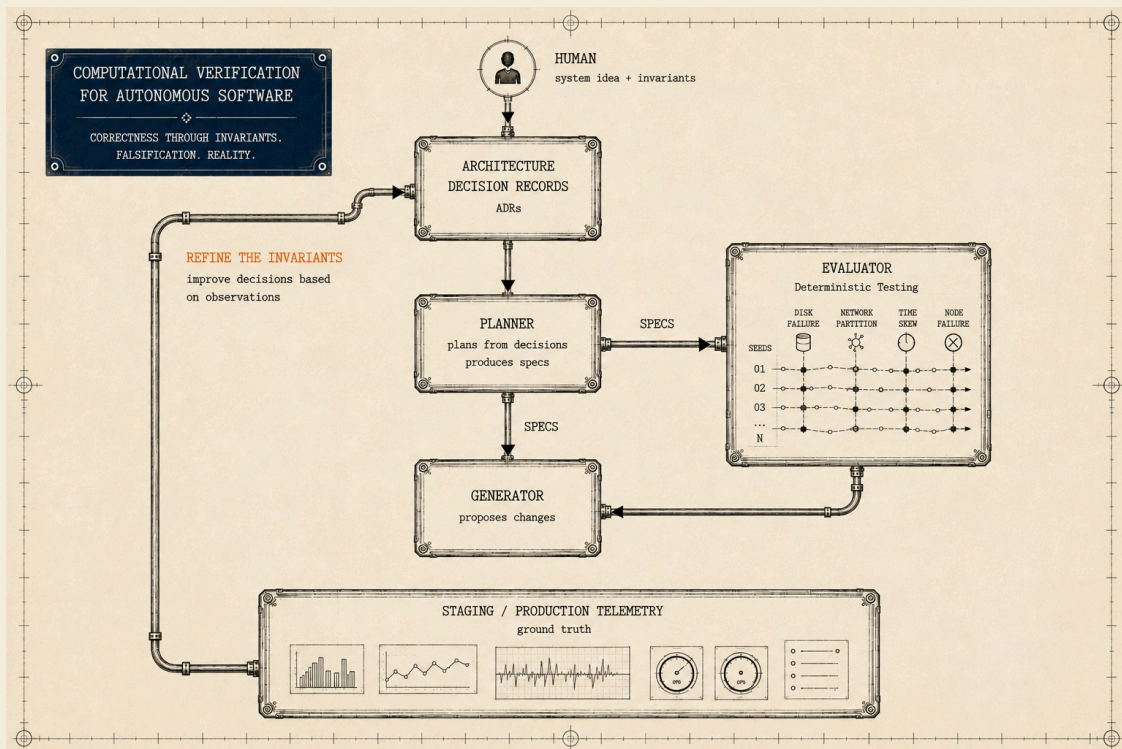
edge cases. The tuning loop was manual and iterative. Read the evaluator's transcripts. Find examples where its judgment diverged from yours. Patch the prompt to remove that difference. Repeat. It took several rounds before the grading was trustworthy. To align the judge with human taste, they put example gradings with detailed score breakdowns into the prompt. That also reduced drift in scores across iterations. Even after tuning, honest limits remained. Small layout issues slipped through. Deeply nested features got too little testing. And Claude cannot hear, so for the DAW, musical taste sat outside the feedback loop entirely. The product evaluator has room to grow, and it also has limits. Know where yours are.

The Computational Evaluator (Datadog)

What it is. This one is not a language model at all. It is a deterministic verification system. Deterministic means the same input always produces the same result. It judges the work with five kinds of checks: simulation, property tests, model checking, bounded proofs, and telemetry. Telemetry means measurements collected from the running system. The checks test invariants, meaning rules about the system that must always hold, and a machine can check them. The other two evaluators produce a judgment. This one produces proof, bounded but reproducible, in seconds, on every iteration. Datadog's claim, demonstrated on two full systems, is that when the domain allows it, this evaluator is strong enough to replace code review as the primary source of correctness.

Contracts before code

The method starts before any code exists. State the invariants explicitly. For Helix, their Kafka-compatible streaming engine, that meant defining up front what counts as durable versus acknowledged, what counts as committed versus visible, and what happens on a crash at each boundary. Those definitions drew on a decade of operating Kafka. The agent is not allowed to invent what the system means. The design artifacts are the contracts, and every artifact is paired with a check that can catch a mistake in it. Antithesis and AWS call this approach “semi-formal methods”. It means specifications precise enough to be checked and cheap enough to run continuously. The honest accounting of the cost is this. It takes about 2 to 3 times the effort of writing the code itself. The cost is repaid at once, because explicit invariants turn every agent iteration into an objective pass or fail instead of a judgment call.



The verification pyramid

Each layer trades speed for rigor. The rule is to use the lightest check that can catch the mistake in question.

LAYER	TOOL	TIME	CONFIDENCE
Symbolic	TLA+ specs	2 min read	Understanding
Primary	Deterministic simulation testing (DST)	about 5 s per run	High
Exhaustive	Model checking (Stateright)	30 to 60 s	Proof
Bounded	Bounded verification (Kani)	about 60 s	Proof, within bounds
Empirical	Telemetry and benchmarks	seconds to minutes	Ground truth

The TLA+ specs give the team a precise, shared description of the system. TLA+ stands for “Temporal Logic of Actions”. It is a formal language for describing a system’s states and actions, plus its invariants, in mathematical form, so that a model checker can verify them. Stateright and Kani, lower in

the pyramid, are Rust tools that check and prove those properties against the actual code. The specs are generated from architecture decision records. They catch ambiguity early, and they define the state variables and invariants that would otherwise take hours to extract from code.

DST does most of the work. FoundationDB and TigerBeetle made the technique well known. The idea is to abstract physical time and make execution deterministic. Faults are then injected synthetically at three levels: the network, the disk, and the node. Each run takes about 5 seconds and exercises production code through randomized fault scenarios. Deterministic seeds make every failure reproducible. The agent replays the exact sequence and traces the invariant violation to the line that caused it. On top of DST, the team added property-based testing, e.g., the roundtrip property `decompress(compress(bytes)) == bytes`. They also used metamorphic tests and differential tests, which check a program by comparing related runs and alternative implementations against each other. They widened race windows on purpose, in the style of FoundationDB's BUGGIFY, so rare timing bugs would appear more often. The target was 500 seeds per component before a component counted as passing. Then they scaled to 10 million seeds across the system. Then came system-level integration with invariants that match Kafka's semantics: every acknowledged message can be consumed, offsets only move forward, and leadership changes lose no writes.

One catch shows the value of the whole approach. DST found a bug in the write-ahead log where truncation in memory happened before the sync to disk. An injected disk fault meant the segment was never retried, and data was lost. The fix was copy-on-write. Their commentary sums up the whole argument. The bug was obvious once the simulation pointed at it, and easy to miss in review. Bugs like this appear only under specific fault timing. Unit tests do not find them. Code review might sometimes find them. DST caught them deterministically in seconds.

Using the loop to improve performance safely

Once the tests guarantee correctness, performance work becomes a controlled series of small steps. The agent proposes an optimization, and the full DST

suite runs. If the suite passes, measure the change and keep it. If the suite fails, revert the change. The evaluator caught the agent as it broke things. A set of CPU optimizations in redis-rust, guided by a profiler, violated the linearizability check for a single node, meaning the check that operations behave as if they happened one at a time. The system reverted them automatically. The loop also produced gains. Production telemetry showed redis-rust using eight times as much memory as Redis. The cause was a hardcoded 4MB buffer preallocation that had been tuned for a micro benchmark. Fed that telemetry, the agent proposed and shipped three optimizations that cut memory by 87%, within minutes. Helix went from its first implementation to about 93% of peak disk throughput in about a day of this loop, while still passing the full test regimen. In staging, Helix served a live stream of 10,000 messages per second from APM profiling, at a median produce latency of 22.2 milliseconds. The baseline Kafka cluster served the same stream at 116 milliseconds.

Agents reverse the economics of verification

Datadog’s structural point is this. Before agents, code review was the most scalable verification method, because every team already does it. Formal methods were the least scalable, because they are expensive and specialized, and teams justified them only for safety-critical systems. Agents reverse that ranking. The language model generates the TLA+ specs and writes the DST harnesses. It also runs the Kani proofs as pipeline stages. Investments that once took months become automation. Helix could not have been built with review as the primary verification. The agent produced too many iterations, too fast. Review does not disappear. It becomes a fast, cheap check. The reviewer reads the harness output instead of the diff. The output covers three things: which invariants passed, which seeds ran, and what the telemetry confirmed. In their words, it is “Less like reading code, more like reading EXPLAIN ANALYZE.”

The human role was narrow but consequential. Humans defined the system idea and the invariants. Humans reviewed and strengthened the DST harness. Humans set measurable targets and approved architecture changes. The shift to an actor-based design was a human call. Everything else was the agent

running against the evaluator. That included drafting designs and implementing them, along with fixing DST failures and optimizing. The builds reached 300,000 lines.

The objection, and its answer

What happens when the evaluator itself is wrong? Incomplete invariants produce automated confidence in incorrect behavior. That is the worst failure mode, because it is confident. Datadog's answer is observability, meaning the practice of watching the live system through its telemetry. Production telemetry feeds back into the verification pipeline. The telemetry includes metrics and logs, plus traces and trajectories. The feedback reveals mismatches between the modeled behavior and the observed behavior, and the invariants get refined. Without observability, the loop is not closed. And one investment rule governs all of it. Invest in the harness in proportion to the cost of failure. A verification pyramid for the payments path, and only lint plus tests for the internal dashboard, is correct, not inconsistent.

Choosing, combining, and running all three

The three are complements, not competitors. Which one fits depends on what can be checked. The Computational Evaluator is the strongest and cheapest per check, but it works only where a machine can check the property. The Product Evaluator covers what only usage reveals. Does the thing work, and is it good? The Review Evaluator covers judgment about the change itself, which neither of the others catches. Does this diff violate our own rules? Does it break a sibling file? Does it delete something other code silently depends on? Datadog’s “verification frontier” is the sorting rule. Wherever a property can be verified automatically, delegate more to the agent and to the computational layer. Wherever it cannot, an evaluator based on a model stays in the loop, and behind it a human. Böckeler’s terms for the two kinds are computational and inferential. Computational checks run as code and give definite answers, so they run on every change. Inferential checks require judgment, so they run where judgment is needed, and they get a budget.

A typical composition for one feature looks like this:

- Contracts and invariants are defined up front. This is computational, and it steers the work before it starts.
- The generator iterates continuously against DST, tests, and linters. This is computational, and it runs on every change.
- The product evaluator grades each sprint against its contract by driving the app. This is inferential, and it runs once per chunk.
- The review evaluator scouts and verifies the resulting PR against the rules routed for that domain. This is inferential, and it runs once per change.
- Telemetry validates the work in staging and production and feeds back into all three. This is computational, and it runs continuously.

Six laws hold across all three. Each one was earned in production.

1. **Never let the generator certify its own work.** This is the one universal law. Self checks are fine. Self approval is the failure mode all three evaluators exist to prevent.
2. **Skepticism must be tuned, and the tuning is human work.** DoorDash added the step that tries to disprove each finding. Anthropic read transcripts and patched the judge. Datadog strengthened invariants. In every case, the most valuable human work was making the evaluator harder to satisfy, not making the generator better.
3. **A finding must be actionable, or it must not appear.** Give the file and line. Name the concrete behavior at risk, and say where to start. A correct finding without an action point costs trust, as DoorDash saw, and stalls iteration, as Anthropic saw.
4. **The evaluator needs its own measurement.** DoorDash built DashBench for the reviewer. Anthropic built sets of example gradings for the product judge. Datadog reconciled telemetry against the modeled behavior. An unmeasured evaluator drifts. An evaluator based on a model drifts toward being too generous. An evaluator based on invariants drifts toward false confidence.
5. **Bound the runs.** Use soft timeouts that convert stuck work into partial, verified results. Put hard cutoffs behind them. Account for cost per successful evaluation, not per token.
6. **Recheck the evaluator on every model release.** Where the evaluator sits reflects assumptions about what the generator cannot do. Those assumptions expire, so move the evaluator when the model improves. Note what happened in every source, though. As raw capability grew, the evaluator stayed valuable at the new, harder limit of what the model could do. Better models do not retire evaluators. They relocate them.

All three sources describe the same end state. The human's expertise moves from checking the output to designing the checks. Unlike review effort, the checks accumulate. Every check added to an evaluator catches a whole class of failures in every future run. That is what makes quality repeatable, and it is what lets the agents run longer.

Sources

1. DoorDash, “How DoorDash built an AI code reviewer engineers actually listen to” (May 2026).
<https://careersatdoordash.com/blog/doordash-built-an-ai-code-reviewer-engineers-actually-listen-to/>
2. DoorDash, “How we learned to trust our AI code reviewer at DoorDash” (DashBench) (Jul 2026).
<https://careersatdoordash.com/blog/how-we-learned-to-trust-our-ai-code-reviewer-at-doordash/>
3. Anthropic, “Harness design for long-running application development” (Mar 2026).
<https://www.anthropic.com/engineering/harness-design-long-running-apps>
4. Datadog, “Closing the verification loop: Observability-driven harnesses for building with agents” (Mar 2026).
<https://www.datadoghq.com/blog/ai/harness-first-agents/>