AEVAL: From Anecdotal to Deterministic Testing for Agentic Skill Workflows

Tejas Singh Anand [*1,2] **Yuet Ying Christina Wang** [1] **Wanting Jiang** [1] **Steve Masson** [1] **Tian Zheng** [1] **Bingjie Zhou** [1]

Abstract

Modern agentic systems increasingly rely on *skills*: installable packages of natural language and code that teach an LLM agent how to perform a domain task. As skill repositories grow, developers need automated, trustworthy quality signals on every change to a skill, yet most evaluation today is *anecdotal*: a developer asks an agent to "try the skill," watches a demo turn, and forms a subjective impression of whether it works. This produces neither reproducibility across runs nor comparability across versions, and it scales poorly to multi-skill marketplaces where a single regression can silently break dozens of downstream workflows. We present **AEVAL** (*Agentic Evaluation*), a CI-integrated framework that replaces this anecdotal practice with a deterministic, reproducible test pipeline for agentic skills. The framework treats every skill change as a triggered test event, runs each skill against a developer-declared evaluation contract (`eval.config`) inside an automated executor, and emits a structured, evidence-grounded quality signal that downstream CI can route on. A key technical ingredient is a structural separation between the executor and the grader, which prevents a subtle but pervasive failure mode of agentic evaluation: an agent that silently self-corrects during execution and then grades its own patched outputs as passing. Our contributions are: (i) a deterministic, change-triggered evaluation protocol for skills with per-skill evaluation contracts and per-run artifact schemas; (ii) a formalization of *self-correction bias* as a distinct failure mode of naive agentic evaluators; (iii) a structural executor/grader separation with a first-attempt grading rule and explicit self-correction tracking; and (iv) a tiered, grounded-evidence suggestion scheme (LV1 causal fixes, LV2 quality improvements) posted as inline merge-request comments. We validate the framework against a runtime-agnostic interface tested with multiple popular agent SDKs. Across a set of real skills in a production agentic stack, the protocol converts spurious 100% pass rates into reproducible first-attempt FAIL signals with an auditable record of every fix the executor applied, enabling reliable routing and stopping decisions in downstream agentic workflows.

1. Introduction

Agentic coding systems increasingly externalize domain capability through *skills*: small, versioned packages containing a natural-language specification (typically a `SKILL.md`), supporting scripts, and configuration (Anthropic, 2025). An agent runtime discovers a skill from its library and follows its instructions to perform a task. In a production stack, a skill failure is effectively a product failure: a regression in a single skill file silently breaks every downstream workflow that depends on it.

Despite this, evaluation of skills today is overwhelmingly *anecdotal*. The dominant practice is for a developer to open an interactive session, paste a representative prompt, observe a demo run, and form a subjective impression of whether the skill "still works." The signal is informal, non-reproducible, and incomparable across versions: two consecutive runs may exercise different code paths, produce different outputs, and elicit different developer judgments. As a skill market-

[*]Tejas Singh Anand contributed to this work during an internship at NVIDIA. [1]NVIDIA [2]University of Waterloo. Correspondence to: Tian Zheng <tiazheng@nvidia.com>, Tejas Singh Anand <tejassingha@nvidia.com>, Yuet Ying Christina Wang <christwang@nvidia.com>.

place grows to dozens or hundreds of artifacts, this practice does not scale, and silent regressions accumulate between releases. We argue that what is needed is the same shift that software engineering underwent decades ago: from anecdotal manual testing to a deterministic, change-triggered test pipeline whose outputs are reproducible artifacts rather than human impressions.

This paper introduces **AEVAL**, a framework that performs that shift for agentic skills. AEVAL treats every change to a skill as a CI-triggered test event. The skill ships with an explicit *evaluation contract* (`eval.config`) declaring its test prompt, expected outcome, and required credentials; the framework installs the modified skill into a clean session, executes it against the contract via an automated agent runtime, and emits a structured, machine-readable quality signal (per-assertion pass/fail with cited evidence, a first-attempt pass rate, an analysis pass over benchmark statistics, and tiered fix suggestions). The signal is reproducible across runs, comparable across MRs, and consumable by downstream CI without human interpretation. In practice this turns the `eval.config` into a *persistent, declarative test harness*: the developer describes the test workflow once at skill-creation time, and every subsequent push automatically replays the full install–execute–grade–suggest-fixes loop with no human in the loop. There is no need to re-open an interactive coding agent, re-load project context, or re-supply credentials per run; the harness is a checked-in artifact that any push or manual CI trigger replays end-to-end. When grading fails, AEVAL also acts as the developer's automated reviewer: it posts inline fix-suggestion commits on exactly the source lines responsible for the failure, ready for the original author to apply with a single click and re-trigger the pipeline.

Building a deterministic pipeline on top of a stochastic agent introduces a specific reliability problem that any naive instantiation will inherit, and that we address as a core technical contribution. When the same agent both executes and grades the skill, the signal is systematically optimistic: LLM agents are *trained* to recover from errors (Saunders et al., 2022; Bai et al., 2022) and will silently retry, amend configuration, or patch the skill in place until something succeeds, then truthfully report the state of the world they produced. The resulting evaluation measures agent debugging ability rather than skill quality, a pattern closely related to observed self-preference effects in LLM-as-judge settings (Panickssery et al., 2024; Zheng et al., 2023). We refer to this failure mode as *self-correction bias*, formalize it in Section 3, and prevent it through a structural separation between the executor and the grader (Section 4.3). Without this separation, the deterministic pipeline returns reproducible *noise*; with it, it returns a reliable signal that downstream uncertainty quantification methods (Kadavath et al., 2022; Lin et al., 2022; Angelopoulos et al., 2023) can consume.

Contributions. We make four contributions.

1. **Deterministic skill testing** (Sections 4 and 6): a change-triggered evaluation protocol for agentic skills with per-skill evaluation contracts, per-run artifact schemas, and an agent-runtime-agnostic executor interface, replacing anecdotal manual evaluation with a reproducible CI pipeline.
2. **Self-correction bias** (Section 3): we formalize a distinct failure mode of agentic evaluators and show it arises structurally rather than from a specific model choice.
3. **Structural separation and first-attempt grading** (Section 4.3): a protocol that constrains the grader to a restricted information set (outputs plus execution transcript) so it cannot participate in corrective action. Combined with a *first-attempt grading rule* and explicit self-correction tracking, this yields a quality signal grounded in the skill as shipped rather than the skill as patched.
4. **Grounded, tiered suggestions delivered as one-click MR commits** (Section 5): every suggested fix must cite a specific failed assertion or tracked self-correction entry, with LV1 (causally implicated in failure) and LV2 (improvements flagged by the grader) tiers preventing speculative "nice to have" churn. Each suggestion is posted as an inline GitLab MR comment with an "Apply suggestion" diff, so when AEVAL detects a defect it also produces an empirically validated patch the original developer can apply in a single click, closing the test-fail-fix loop within the same merge request and turning the framework into an automated reviewer for the skill author.

2. Background and Related Work

Agent evaluation frameworks. Existing platforms sit at one of three layers. *Prompt/response* platforms (LangSmith (LangChain, 2025), Braintrust (Braintrust, 2025), Promptfoo (promptfoo, 2025)) score model outputs against assertions; they do not install or execute deployable skill artifacts. *Capability benchmarks* (OpenAI Evals (OpenAI, 2024), AgentBench (Liu et al., 2024), SWE-bench (Jimenez et al., 2023)) evaluate a *model's* general problem-solving on a fixed dataset; they cannot be pointed at a developer's modified skill in an MR. *Sandboxed agent evaluation* (Harbor Framework (Harbor, 2025)) runs agents against predefined tasks in containers; it tests the agent, not the skill artifact, and does not spawn a structurally independent evaluator.

LLM-as-judge and self-evaluation. LLM-as-judge protocols (Zheng et al., 2023) and self-evaluation pipelines (Saunders et al., 2022; Ren et al., 2023) are known

to exhibit self-preference (Panickssery et al., 2024) and calibration issues (Kadavath et al., 2022; Lin et al., 2022). These works study judgment *of model outputs*. Our setting is different: we study judgment of a *skill artifact's behavior* under an executor that actively modifies the world during evaluation.

Uncertainty quantification for agents. Distribution-free methods such as conformal prediction (Vovk et al., 2005; Angelopoulos & Bates, 2023) and prediction-powered inference (Angelopoulos et al., 2023) provide rigorous coverage under exchangeability. Our protocol is complementary: we do not compete with conformal guarantees but rather prepare a *reliable ground-truth signal* that such methods could consume. Without self-correction-bias-free evaluation, downstream conformal or sequential procedures are fed contaminated labels and their nominal guarantees collapse.

3. The Self-Correction Bias Problem

Let s be a skill, x an eval prompt from a developer-supplied `eval.config`, and A an autonomous agent. A naive evaluator runs A on (s, x), receives a terminal state y, and emits $\hat{q}(s) \in \{0, 1\}$ indicating whether s worked. We observe that A's execution trajectory

$$\tau = (a_1, o_1, a_2, o_2, \ldots, a_T, o_T)$$

is typically not a read-only probe of s. Because A has tools (edit, bash, write), actions a_t may modify the skill itself or adjacent configuration. Let $\Delta(\tau)$ denote the set of skill-file edits executed during τ. Define

$$q^\star(s) = \mathbb{I}[s \text{ works first-attempt, unmodified}]$$
$$\hat{q}_{\text{naive}}(s) = \mathbb{I}[y \text{ satisfies } x]$$

The failure mode is that $\hat{q}_{\text{naive}}(s) = 1$ whenever A can recover via any $\Delta(\tau) \neq \emptyset$, even when $q^\star(s) = 0$. This is structural: an agent that is competent at debugging will hide defects in any skill s that is merely *fixable* rather than *broken*. We observed this empirically in our own deployment, an agent-first product in which skills are authored by developers and reach end users without an intermediate packaging step. In this setting every developer push to a skill is itself a production event, so a regression masked by a self-correcting evaluator ships directly. On one such push, a first-run execution failed with a missing configuration entry, the agent silently patched the skill in place, re-ran, and produced a 100% pass grade against assertions it had itself authored after observing the corrected output. This is exactly the bias the formal model above predicts.

Two design forces cause this. First, *shared identity*: when the same agent runs and grades, it has no incentive to flag its own patches. Second, *post-hoc assertions*: assertions written after seeing outputs implicitly fit the outputs the agent produced. Eliminating the bias requires addressing both.

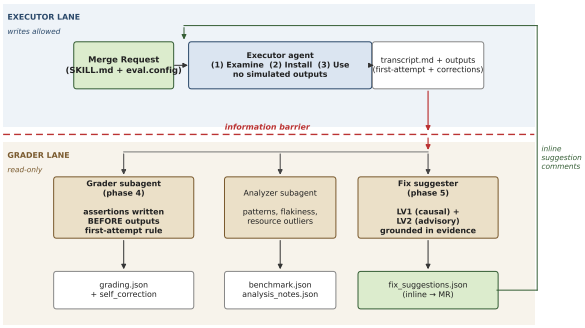

Figure 1. AEVAL pipeline. An MR triggers CI, which detects changed skills and invokes the evaluator. The executor runs the four-phase workflow; the grader is a separate subagent with access only to outputs and transcript; suggestions are posted to the MR.

4. Method

AEVAL is a five-phase pipeline triggered by a merge request on a skill directory (Figure 1). The core pipeline runs: **(1) Examine**, **(2) Install**, **(3) Use**, **(4) Evaluate**, and **(5) Suggest**. The distinctive design choices sit in phases 3–5.

4.1. Skill-Defined Evaluation Contract

Each skill optionally ships an `eval.config` file co-located with its `SKILL.md`. The contract specifies, per test case, a natural-language *prompt*, an *expected outcome*, and a list of *required credentials*. Multiple independent cases are supported via an `evals` array, each executed in an isolated SDK session. A three-tier fallback (per-skill, per-name, generic) ensures every skill is testable even without explicit configuration. Locating the test specification with the skill ensures each artifact declares its own quality criteria, moving the eval contract into the MR's diff and out of a centralized benchmark registry.

4.2. Banning Simulated Outputs

Phase 3 of the executor prompt imposes a hard rule: *no simulated outputs*. If a command fails, a service is unreachable, or a workflow errors, that failure is recorded in a per-case transcript τ as a real test result. The agent may self-correct, but it cannot fabricate logs, synthesize results, or place-hold missing outputs. This is enforced by instruction rather than sandbox, but we found it essential: without it, agents routinely produce synthetic training logs when a GPU cluster was unreachable and grade those as passing.

4.3. Separating Execution from Grading

Phase 4 grades the skill. The core constraint is that the *grader is a separate subagent*, spawned from a fixed protocol (`agents/grader.md`) and restricted to the information set

$$\mathcal{I}_{\text{grader}} = \{x, \tau, \mathcal{O}, \mathcal{A}\},$$

where x is the test prompt, τ is the read-only execution transcript, $\mathcal{O}$ is the set of output files, and $\mathcal{A}$ is a set of pre-execution assertions. Crucially, the grader does *not* participate in the execution and has no tool access that could alter $\mathcal{O}$ or s.

Assertions before outputs. $\mathcal{A}$ is written *before* the executor observes any outputs, based solely on the skill's `SKILL.md` and the declared expected outcome. This prevents assertion-after-observation fitting.

First-attempt grading rule. The grader evaluates each $a \in \mathcal{A}$ against the first-attempt trajectory. If τ contains any self-corrective edit $\Delta(\tau) \neq \emptyset$ that was causally required for an assertion to succeed, that assertion is marked FAIL regardless of later success. The grader writes a structured `grading.json` containing:

- per-assertion pass/fail with cited evidence from τ or $\mathcal{O}$;
- a `self_correction` section enumerating first-attempt errors, applied changes, and affected assertions;
- a `claims` section of extracted process and quality claims, each verified against $\mathcal{O}$.

This explicit decomposition makes the distinction between *the skill worked* and *the agent fixed the skill and then it worked* auditable and machine-readable. Downstream CI uses only the first-attempt pass rate as its gate signal.

4.4. Independent Analysis Pass

A second subagent (analyzer) receives the aggregated benchmark and writes freeform observations to `analysis_notes.json`: non-discriminating assertions, flaky assertions with high variance, token/latency outliers. It does not see skill source; its role is purely descriptive over the run distribution.

5. Grounded, Tiered Fix Suggestions

When $\hat{q}^\star(s) = 0$, AEVAL emits fix suggestions intended for human review on the originating MR. Two design rules keep this channel signal-dense.

Grounded-in-evidence. Every suggestion must carry a `grounded_in` field citing either (a) the exact text of a failed assertion in `grading.json`, or (b) a specific entry in `self_correction.changes_made`. Suggestions without a grounding reference are rejected at the prompt level; the agent is instructed that if nothing failed and no correction was needed, it must emit an empty suggestion set. This rules out speculative "consider adding..." churn that we observed in earlier iterations, where two runs on the same MR produced disjoint suggestion sets.

Tiered levels.

- **LV1 (serious)**: fixes causally implicated in first-attempt failure. Must trace to a FAILED assertion or a `self_correction` entry.
- **LV2 (improvement)**: fixes the grader identified as quality- or robustness-improving in `eval_feedback.suggestions` but that did not cause a hard failure.

Tiering matters for routing: LV1 suggestions gate the merge, LV2 suggestions are advisory. This aligns with selective-prediction style calibration, reporting *how confident we are that action is required* (Ren et al., 2023).

Delivery as MR suggestion commits. Each LV1 or LV2 entry maps to a source line in the skill directory and is posted via the GitLab Discussions API as an inline suggestion comment. The developer sees a diff and an "Apply suggestion" button. Fixes that fall outside the MR's diff range are collected into a combined fallback comment with a copy-paste prompt. This closes the loop: AEVAL not only detects failures but produces empirically validated patches that a developer can apply in one click.

6. System and Deployment

The framework is distributed as (i) a Python harness that drives an automated agent runtime through a generic streaming-execution interface (initial prompt, structured tool-use events, per-event transcript capture), (ii) a bundled evaluation skill (containing grader, analyzer, and comparator subagent definitions) injected into the executor's working directory set, and (iii) a reusable GitLab CI template that includes a drop-in `include` directive. The CI template detects changed skills via `git diff`, runs the evaluator per skill, posts per-case summaries as MR notes, and invokes the suggestion poster.

Agent-runtime-agnostic design. The harness deliberately abstracts over agent SDKs: the executor interface specifies only what is needed for skill execution (prompt input, working-directory injection, structured tool-use events, transcript stream), so any popular agent SDK can be plugged in. Internally we have validated the framework against multiple widely used agentic coding runtimes – including Claude-,

Codex-, and OpenCode-style agents – and observed that producing per-runtime reports surfaces compatibility issues specific to each runtime (for example, differences in how tools are invoked, how working directories are mounted, or how multi-turn state is persisted) that single-runtime evaluation misses. The same skill and `eval.config` produce comparable per-runtime artifact trees, enabling cross-runtime regression tracking. The framework is independent of any one SDK choice and the public deployment treats the runtime as a configurable backend rather than a fixed dependency.

Artifact schema. Each run produces a tree of JSON and markdown artifacts: `transcript.md`, `eval_metadata.json`, `grading.json`, `benchmark.json`, `analysis_notes.json`, `fix_suggestions.json`. Multi-case runs additionally produce a merged `multi_eval_report.json` and human-readable summary. The schema is stable and supports downstream regression tracking: each push can be compared to a last-known-good baseline, with pass-rate deltas and newly-failing assertions flagged.

From single skills to whole workflows. The `eval.config` accepts an `evals` array of independent test cases (e.g., *train*, *inference*, and *evaluate* stages of a model-development workflow), each executed in its own isolated session and graded independently. This makes the framework suitable not only for single-prompt skills but for entire multi-step workflows that a developer would otherwise re-orchestrate by hand. Combined with the persistent-harness property above, this matters for the everyday developer loop: rather than reopen an interactive agent, paste the project context, supply credentials, and describe the workflow each time, the developer encodes the workflow once in `eval.config` and gets an automated QA-and-experimenter substrate that replays the same install–execute–grade–suggest-fixes loop on every push or manual CI trigger. The investment in describing the experiment is paid once at skill-creation time and recovered on every subsequent change, while the structural separation in Section 4.3 keeps each replay's quality signal honest.

7. Empirical Evaluation

We evaluate on a production marketplace of skills spanning model training, data generation, inference, and document generation. We report qualitative findings that illustrate the bias and the protocol's behavior; a full quantitative study is deferred to the extended version.

Case: a deliberately degraded segmentation skill. We evaluated a segmentation skill whose `SKILL.md` documented action names (`segment_train`, `segment_evaluate`, `segment_inference`) that did not match the target container's actual action names (`train`, `evaluate`, `inference`). Under a naive agentic evaluator, the first call returned a `KeyError`; the agent patched the action name in its generated runner script and succeeded. The naive grade was 100% pass.

Under the protocol described in Section 4, the same run produced a *10/10 assertion pass rate* on the corrected outputs but recorded `self_correction.was_needed = true` with three first-attempt errors and two applied changes. Assertions such as "The skill's configured action names work without modification" and "All configuration files are consistent with each other" were marked FAIL with cited evidence. The first-attempt pass rate reported to CI was therefore strictly lower than the terminal pass rate, and the LV1 suggestion emitted against the MR identified exactly the action-name mismatch with an "Apply suggestion" diff.

Grounded-suggestion stability. Prior to the `grounded_in` requirement, repeated runs of the same MR produced non-overlapping suggestion sets. One run generated five "for consistency" suggestions, a second run generated seven largely different ones. After the rule, repeated runs converge on the same causally grounded LV1 set; LV2 remains slightly variable but is advisory. This is precisely the form of stability required for a signal to be consumed by a downstream conformal or sequential procedure.

Ban on simulated outputs. On a skill whose downstream GPU cluster was intermittently unreachable, earlier iterations of the prompt produced a complete set of synthetic log files and graded them as passing. After we added the simulation ban, the agent instead records the connectivity failure and terminates the case with `status = error` and an explicit transcript entry. Downstream CI can then distinguish *skill defects* from *environmental failures*, which are treated differently: the former blocks merges, while the latter triggers a retry.

7.1. Cross-runtime calibration: per-agent grader behavior

Because the executor interface is runtime-agnostic (Section 6), the same skill and `eval.config` can be run through different agentic coding backends and the resulting artifacts compared directly. We exercise this across a matched set of runs of an intelligent finetuning workflow on a production skill that orchestrates multiple sub-skills, with two backends drawn from different families: **Backend A** (a Claude-family agent) and **Backend B** (a GPT-family agent). Each backend is invoked multiple times against the same prompt and configuration so we can extract per-run

quantities rather than a single anecdote. Table 1 reports five concrete metrics per run.

Pass rates are not the discriminating signal. Both backends complete the workflow end-to-end and verdict *PASS*; raw pass rates fall in 70–89% on both sides and the ranges overlap. Pass rate alone does not separate the two graders, in part because each instance writes its own assertion list from the declared expected outcome (denominators 10–18) and so the M/N ratio is not a strict apples-to-apples comparison.

Causal share separates them cleanly. The discriminating quantity is the *causal share*: the fraction of fix suggestions the grader marks as merge-blocking (LV1) rather than advisory (LV2). Backend A's mean causal share is 0.41; Backend B's is 0.82, a partition gap of 0.41 on a $[0, 1]$ scale. Inspecting the underlying suggestions, both backends surface the same root causes (mismatched CLI flags, root-owned outputs, configuration-schema gaps); they disagree on which of those causes count as merge-blocking. Backend B promotes config-consistency findings into LV1 that Backend A leaves in LV2. Neither is wrong, since the LV1 definition ("causally implicated in first-attempt failure") admits both readings when a finding is causally adjacent to but not directly responsible for an assertion failure. The disagreement is nevertheless large, reproducible across runs of each backend, and consequential because LV1 is the merge-gating tier.

Direct measurement of self-correction activity. Backend A explicitly logs each in-place patch its executor applies. Across its short-workflow runs we observed five and six self-corrections respectively: direct empirical evidence of the bias pattern formalised in Section 3. Without the structural separation of Section 4, every one of those self-corrections would have been silently absorbed into a clean PASS grade.

Protocol fidelity is itself a measurable failure mode. The framework requires the executor to emit a locked output-template (callout block, structured verdict, ⟨details⟩ balance, tiered-suggestion split). Across the matched set, one Backend A run silently rolled its own format on the same prompt that produced its sibling's clean output; the other five runs were format-compliant. This is an agent-as-evaluator failure mode that the framework's rigid artifact schema let us *detect* by simple structural checks. It is a consequential failure, because a downstream uncertainty-quantification procedure consuming non-canonical artifacts would silently drop or misparse them. The fidelity column in Table 1 is therefore not a presentation detail; it is a per-run reliability metric.

Cost is dominated by cache. With ≈95–97% prompt-cache hit rates, the marginal token cost of replaying the protocol on each push is roughly an order of magnitude smaller than the gross figure suggests. The framework is consequently cheap enough to re-run on every change to a skill, a precondition for the persistent-harness property of Section 6.

The combined picture this gives is the form of evaluator uncertainty that statistical frameworks for agentic systems must accommodate: *the same input, the same protocol, two graders, a systematic disagreement on severity, and an occasional protocol-fidelity violation that the framework itself can detect*. The fact that this is all expressible as machine-readable per-run scalars (causal share, self-correction count, fidelity boolean) is, we believe, the framework's structural validation: its signal is well-defined enough that running it through independent graders surfaces a clean, reproducible calibration gap rather than incomparable noise. This is exactly the input downstream conformal or sequential procedures need.

8. Discussion

Scope of the guarantee. Our protocol does not provide a probabilistic coverage guarantee in the conformal sense; it provides a *structural* guarantee that the grader's reported signal is a function of the first-attempt execution rather than the post-correction state. Combined with pre-observation assertions and a non-participating grader, this gives a reproducible quality signal that can serve as the ground-truth input for statistical methods.

Calibration of LV1/LV2. The boundary between LV1 and LV2 is currently determined by the grader's classification of whether a finding traces to an assertion failure or an `eval_feedback.suggestions` entry. We summarise this boundary by the per-run *causal share* introduced in Section 7.1, which collapses the partition into a single scalar in $[0, 1]$ comparable across runs of different denominator. A natural extension is to apply conformal calibration over historical MR outcomes: learn a threshold on causal share such that a bounded fraction of advisory-only MRs ever become merge-blocking on a subsequent patch. The cross-runtime study shows that the partition is also *runtime-conditional*: backends from different families assigned the same root causes to opposite tiers, with a 0.41 gap in mean causal share. A single global threshold therefore under-fits; either a per-runtime threshold or a runtime-marginalised severity distribution is needed. We leave both to future work.

Limitations. The ban on simulated outputs is enforced by instruction; a sufficiently capable agent could in principle violate it. Our current mitigation is the transcript audit and the grader's independent verification, but sandboxing the executor's file system write scope to exclude the original skill directory is a cleaner structural defense. The first-attempt

Run	Wall (min)	Assertions (passed/total)	Causal share	Self-corrections	Protocol fidelity
Backend A (Claude-family)					
Short workflow, replicate 1	34.7	10/13 (77%)	0.33	6	× (drift)
Short workflow, replicate 2	34.3	10/13 (77%)	0.50	5	✓
Long workflow	61.6	16/18 (89%)	0.40	—	✓
Backend B (GPT-family)					
Short workflow, replicate 1	40.0	9/12 (75%)	0.80	—	✓
Short workflow, replicate 2	28.7	7/10 (70%)	0.86	—	✓
Long workflow	67.3	9/11 (82%)	0.80	—	✓
Backend A (mean)	**43.5**	**81%**	**0.41**	**5.5**	**2/3**
Backend B (mean)	**45.3**	**76%**	**0.82**	**n/a**	**3/3**

Table 1. Matched runs across two agent backends on the same skill and `eval.config`. *Causal share* = $\mathrm{LV1}/(\mathrm{LV1}+\mathrm{LV2})$, the fraction of fix suggestions the grader marks as merge-blocking rather than advisory; this is a per-run scalar in $[0, 1]$ that normalises across the variable assertion-list size each grader writes from the declared `expected_outcome`. *Self-corrections logged* are the per-run count of in-place skill patches the executor applied during the first-attempt trajectory (Backend A logs these explicitly; Backend B does not, marked "—"). *Protocol fidelity* is whether the run honored the locked output-template contract; one Backend A run silently drifted from it. Pass rates are reported as a passed/total ratio and not directly compared across backends because each grader writes its own assertion list. Mean tokens-in were ≈8.5M (97% cached) for Backend A and ≈12.4M (95% cached) for Backend B; with caching, marginal cost per merge-blocking finding is on the order of a few hundred thousand uncached tokens.

rule also depends on the grader's ability to correctly identify causal self-corrections in τ; complex multi-step corrections can be misattributed. Finally, per-agent evaluation surfaces compatibility issues but multiplies compute cost linearly in the number of runtimes.

Relation to conformal prediction. Our protocol is orthogonal to but compatible with distribution-free uncertainty quantification. The structural separation produces reliable binary labels; conformal prediction and prediction-powered inference (Angelopoulos et al., 2023) consume such labels to provide coverage under exchangeability assumptions. We view AEVAL as the upstream *label-generating* component in a reliable agentic workflow.

9. Conclusion

Evaluation of agentic skills is dominated by anecdotal practice: a developer runs a demo, watches an agent, and forms a subjective impression of whether the skill works. This produces neither reproducibility nor comparability and does not scale to multi-skill marketplaces. We present AEVAL, a CI-integrated framework that replaces this practice with a deterministic, change-triggered test pipeline: skills declare an evaluation contract, every change triggers an automated executor run, and the run emits a structured, evidence-grounded quality signal. The pipeline is grounded by a structural executor/grader separation that prevents self-correction bias, a first-attempt grading rule with explicit self-correction tracking, and grounded-evidence tiered fix suggestions delivered as inline MR comments. Deployed in a production skill marketplace and validated across multiple popular agent runtimes, the protocol converts self-fulfilling 100% pass signals into auditable first-attempt quality signals and produces fixes that a developer can apply in one click. We view deterministic skill testing, with structural separation as its core reliability ingredient, as a foundational building block for statistically rigorous monitoring and stopping of agentic workflows.

References

Angelopoulos, A. N. and Bates, S. A gentle introduction to conformal prediction and distribution-free uncertainty quantification. *Foundations and Trends in Machine Learning*, 16(4):494–591, 2023.

Angelopoulos, A. N., Bates, S., Fannjiang, C., Jordan, M. I., and Zrnic, T. Prediction-powered inference. *Science*, 382 (6671):669–674, 2023.

Anthropic. Agent skills: Progressive disclosure for claude agents. *Anthropic Documentation*, 2025. Accessed 2026-04-23.

Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A., et al. Constitutional AI: Harmlessness from AI feedback. In *arXiv preprint arXiv:2212.08073*, 2022.

Braintrust. Braintrust: The enterprise-grade stack for building AI products. https://www.braintrust.dev/, 2025.

Harbor. Harbor framework: Containerized evaluation for AI agents. https://www.harborframework.com/, 2025.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. SWE-bench: Can language models resolve real-world GitHub issues? *arXiv preprint arXiv:2310.06770*, 2023.

Kadavath, S., Conerly, T., Askell, A., Henighan, T., Drain, D., Perez, E., Schiefer, N., Hatfield-Dodds, Z., DasSarma, N., Tran-Johnson, E., Johnston, S., El-Showk, S., Jones, A., Elhage, N., Hume, T., Chen, A., Bai, Y., Bowman, S., Fort, S., Ganguli, D., Hernandez, D., Jacobson, J., Kernion, J., Kravec, S., Lovitt, L., Ndousse, K., Olsson, C., Ringer, S., Amodei, D., Brown, T., Clark, J., Joseph, N., Mann, B., McCandlish, S., Olah, C., and Kaplan, J. Language models (mostly) know what they know. In *arXiv preprint arXiv:2207.05221*, 2022.

LangChain. LangSmith evaluation. https://www.langchain.com/langsmith, 2025.

Lin, S., Hilton, J., and Evans, O. Teaching models to express their uncertainty in words. In *Transactions on Machine Learning Research*, 2022.

Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., Zhang, S., Deng, X., Zeng, A., Du, Z., Zhang, C., Shen, S., Zhang, T., Su, Y., Sun, H., Huang, M., Dong, Y., and Tang, J. AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations*, 2024.

OpenAI. OpenAI evals: A framework for evaluating LLMs. 2024. Open-source registry of model-capability benchmarks.

Panickssery, A., Bowman, S. R., and Feng, S. LLM evaluators recognize and favor their own generations. In *arXiv preprint arXiv:2404.13076*, 2024.

promptfoo. Promptfoo: Test your LLM app like software. https://www.promptfoo.dev/, 2025.

Ren, J., Zhao, Y., Vu, T., Liu, P. J., and Lakshminarayanan, B. Self-evaluation improves selective generation in large language models. In *arXiv preprint arXiv:2312.09300*, 2023.

Saunders, W., Yeh, C., Wu, J., Bills, S., Ouyang, L., Ward, J., and Leike, J. Self-critiquing models for assisting human evaluators. In *arXiv preprint arXiv:2206.05802*, 2022.

Vovk, V., Gammerman, A., and Shafer, G. Algorithmic learning in a random world. 2005.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Xing, D., Gonzalez, J. E., Stoica, I., and Zhang, H. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Advances in Neural Information Processing Systems*, 2023.