

Cortex: Measurement-Gated Evolution and Pruning of Coding-Agent Skills and Rules

Samuele Giampieri

Independent researcher

October 2026

Abstract

AI coding agents such as Claude Code do not act on their training alone; they also follow text files that developers write for them. A project file describes how the code should be written, *skills* are short guides the agent may choose to read when it judges them relevant, and *rules* are notes tied to particular files, which the agent sees automatically whenever it opens one of those files. We call these files the agent’s *harness*. Developers add to it each time they have to correct the agent, but checking whether an addition really helps is harder than it looks: the same agent given the same task can succeed once and fail the next time, so seeing an instruction work once proves little.

We present Cortex, a tool that tests changes to the harness instead of trusting them. At the end of a working session it saves any fix the session made as a task it can replay; across sessions it spots the corrections the developer keeps making, and writes one new skill or rule at a time to prevent them. It then runs the agent on those tasks several times with the new text and several times without it, and keeps the text only if it helps overall and makes no single task noticeably worse. Text already in the harness is tested the same way before it is deleted. Throughout, Cortex records whether each piece of text reached the agent, and it neither adds nor deletes text that the agent never used.

We tested Cortex with Claude Haiku 4.5 on a repository built for the purpose, with a few coding conventions the model tends to break and one area with nothing to learn, kept as a comparison. A script played the developer, and Cortex was run several times from scratch. Each harness it built was then tested on new tasks it had never seen.

The harnesses Cortex built made the agent follow the conventions far more often than a harness with no skills or rules, left the comparison area unchanged, and added no text that the agent must read at every step. The clearest lesson is about how advice is delivered rather than what it says: rules, which the agent sees automatically, did most of the work, while skills, which the agent must choose to open, helped only when they were offered for the files a task touched; offered on every turn, they were rarely opened. We also fed Cortex useless and harmful suggestions on purpose. It kept none of them, but the agent never opened the useless ones or the harmful skills, and the harmful rules, which did load, were tried on tasks that mostly failed without them, where Cortex reached no verdict, even when a harmful rule broke a task that passed without it. This shows that Cortex keeps no text the agent never uses, not yet that it can spot text that does damage. Finally, in this study Cortex was worth as much as the model needed correcting: under the stronger Claude Sonnet 5, which already followed most of the conventions, a single run’s harness added little, and on a real open-source project where the agent was rarely corrected, Cortex found nothing to learn. We release the tool, the evaluation lab and the data behind every number in the paper.

1 Introduction

Coding agents are increasingly steered not only by their weights but by text their users edit. A project instruction file (CLAUDE.md, or AGENTS.md [16]), a folder of *skills* the agent may choose to invoke, and *rules* that load when the agent reads matching files [4, 5] together make up what this paper calls the agent’s *harness*: the part of its behaviour a developer can change in a minute, by editing text. The agent works in *turns*: at each step the model reads its *context*, the text in front of it (the harness text that is loaded, the request and the work so far), and then acts. Harnesses grow the way documentation grows. A correction a developer has to make twice becomes a line, the line stays, and every line that is always in context is paid for on every turn of every session.

Little of it is measured. Whether an instruction helps is often judged the way it was written: by someone who watched it work once. That is not evidence, because an agent’s success on a fixed task is a noisy quantity: the model samples its output, so the same agent can pass a task on one attempt and fail it on the next. Results in this paper are therefore given as a *pass rate*, the share of attempts that succeeded, and differences between pass rates in percentage

points. In the evaluation reported here, one and the same harness, measured twice on the same tasks in separate batches of rollouts, differed by -3.3 points (95% confidence interval, or CI, $[-7.3, 0.0]$, resampling tasks; Section 4 explains these intervals), a difference that, seen once, would pass for an effect. Recent studies report that adding skills can make an agent worse [54], that skill gains measured on the tasks one repository supplies could not be told apart from run-to-run noise [26], and that whether an updated harness helps depends on the model that uses it: weaker models often fail to activate, or to follow, what was added [31]. A harness curated by impression can accumulate instructions that do nothing, some that hurt, and a context bill for all of them.

A growing body of work optimises or evolves agent text automatically: prompts and programs [1, 25], agent designs [18, 66], and most recently the harness itself [10, 27, 65, 72]. These systems are typically optimised and evaluated against benchmarks. A developer’s problem is narrower and, in one respect, harder: one repository, its own conventions, a handful of tasks drawn from their own work, a fixed context budget, and no way to tell whether a given instruction earns its place.

We present **Cortex**, a tool for Claude Code that treats the harness as something to measure rather than to curate (Figure 1). Cortex calls each skill or rule an *item*, and the harness the developer is using now the *live* harness. It consists of three commands. `/harvest` runs at the end of a working session. When something in the session went from broken to working, it saves a replayable *task*: the broken commit, the developer’s request, and a *check*, a script that decides whether an attempt at the task succeeded. It also logs each correction the developer made along the way. `/evolve` reads the corrections that recur and writes *one* new skill or rule, a *candidate*, for one of them. It also chooses the candidate’s *loading tier*, which decides when its text enters the agent’s context (Section 2): if the fixes behind the problem sit in one folder, the candidate loads only when the agent works there. It then runs the agent on the harvested tasks several times each, with the live harness and with the live harness plus the candidate, in disposable clones of the repository. Each attempt is a *rollout*, and a batch of rollouts that compares two harnesses in this way is a *sweep*. The candidate is kept only if it helped, *fired* in at least one of its rollouts (a rule loaded into the agent’s context, or a skill opened), and made no task worse beyond a set tolerance, which is zero on a task the live harness never failed; a candidate killed only for such a drop is swept again first. `/prune` works the other way round: it re-measures live items by removing them one at a time, and never deletes one it has not seen fire. One principle runs through both measurements: *an item is credited or blamed only where it entered the agent’s context*.

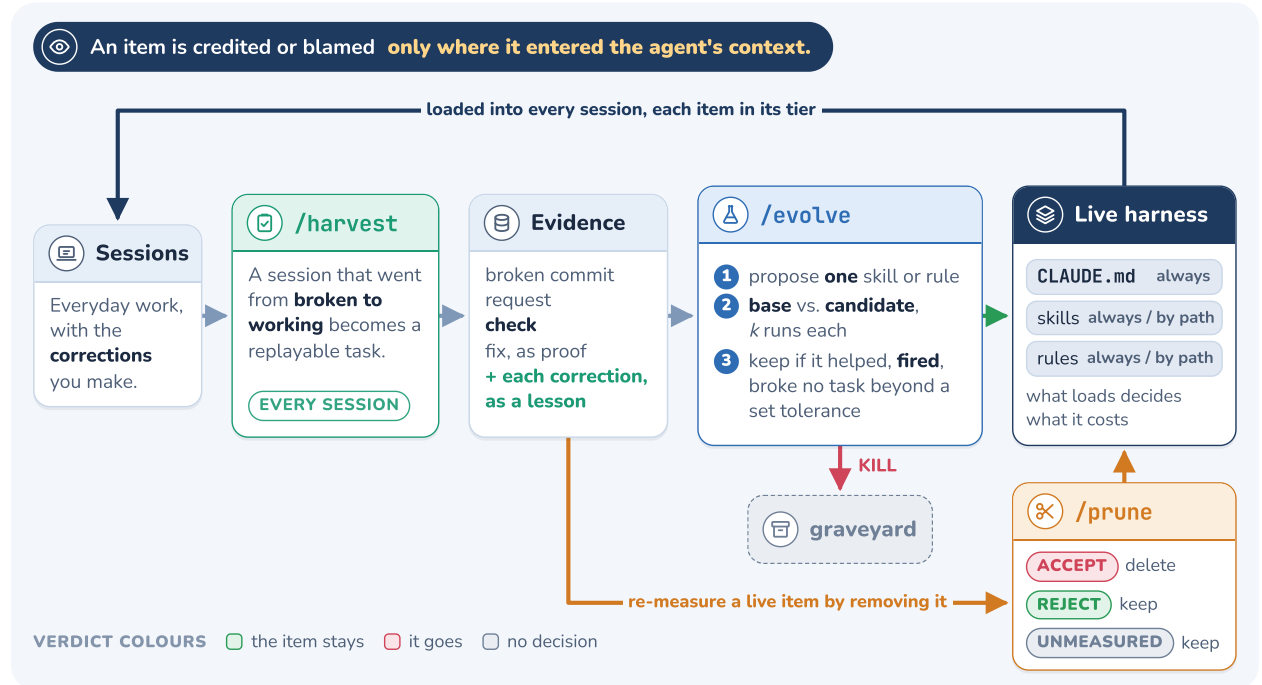


Figure 1: The Cortex loop. `/harvest` turns a session that went from broken to working into a task; `/evolve` measures one candidate against the live harness and keeps it only if it helped, fired, and cost no task more than a small tolerance; `/prune` re-measures live items by removal and keeps any item it never saw fire.

We evaluate Cortex in a replicated study (Section 4), with Claude Haiku 4.5 as both the agent that is measured and the one that runs Cortex’s commands. The study uses a small synthetic repository with four *house rules*: conventions of that one project, such as reading the current time only through its own clock module, which code can break while still working, and which the model often breaks unaided. Each house rule has a *family* of tasks built around it; a fifth family, plain bugs with no rule to learn, is the *control*. A script plays the developer. The 4 independent *runs*, R1 to R4, each start from a fresh copy of the repository, play the same 10 rounds of scripted sessions, and let Cortex evolve a harness from them. Every harness is then measured on 30 *held-out* scenarios: pieces of work of the same families that no session played. The loop never saw them, so a gain on them cannot come from remembering the tasks it was shown: what it learned has to carry over to new work. The main findings are these.

- **It works on held-out scenarios.** On the four rule families, the evolved harness beats a harness with no skills or rules by +55.0 points of pass rate (95% CI [+34.1, +74.1], permutation $p = 0.0001$). The interval allows for chance in both which runs and which scenarios were measured. The control family changes by -0.8 points, and no run kept an item for it.
- **It adds no always-on context.** Every item the loop kept has a `paths` field, which ties it to certain files, so that it loads only when the agent works on them. So the evolved harness adds nothing to what every turn carries: beyond the skill listing every arm shares, its only text present on every turn is the 559-character `CLAUDE.md` that the harness with no skills or rules has too. In the one run that measured it (R1), the evolved harness also scored 11.1 points higher on the rule families than a harness that instead appends the project’s contribution guide, the file that states the house rules, to `CLAUDE.md` (95% CI [0.0, +22.5]); the interval reaches zero, and the comparison crosses a batch boundary (Section 5.4).
- **It kept nothing known to be useless or harmful.** None of 20 *placebo* candidates (plausible advice about matters the repository does not care about) and none of 5 harmful ones was kept. The test was weaker than those counts suggest. Each candidate is first tried in a short, cheap sweep, its *screen*, which runs only those of the tasks the candidate was written for that the live harness does not always pass. The agent never opened the placebos or the 2 harmful skills on their screens, so none could be kept: Cortex’s scoring script refused the harmful skills and 12 placebos for never being used, and reached no verdict on the other 8 placebos, whose screens could show no gain. The 3 harmful rules did load, but harm shows only on a task that passes without the harmful item, and these screens ran on tasks chosen because the live harness had failed them, where no task could show a gain; so Cortex’s scoring script reached no verdict on them, even on the screen where a rule broke a task that passed without it (Section 5.3).
- **Rules are loaded; skills must be chosen.** Of the 18 items kept across the runs, 16 are rules. Claude Code adds a rule to the context itself, whenever the agent reads a file that matches the rule’s `paths`. Of a skill, only the description is listed to the agent; the body enters the context only if the agent decides to open it, and often it did not. A *path-gated* candidate, a skill listed only once the agent works with matching files, was listed in 66 attempts at a task, in its own sweeps and in the gate test (Section 5.3), and opened in 28. An *always-on* candidate, a skill listed on every turn, was listed in 198 attempts, likewise in its own sweeps and in the gate test, and opened in none of them.
- **Its value shrinks under a stronger model.** The harness of run R1, which gained +66.7 points under Claude Haiku 4.5, the model it was evolved with, gained +5.6 points under Claude Sonnet 5 (95% CI [−2.8, +13.9]). One `/prune` pass after the upgrade, which ran each item’s tasks 3 times with the item and 3 times without it, flagged 2 of its 4 items as removable.

We also report what did not work. Only 2 of 4 known-good candidates were kept. `/prune` reached the expected verdict on 6 of the 8 items it was tested on, but on only 2 of the 4 *planted* ones, items we added to a harness with a known right answer. It kept a useless always-on skill, which the agent never opened in its own sweep, because 2 tasks passed less often when it was removed. And an always-on skill that the agent never opens got no verdict where the test asks for *unmeasured*, the verdict for an item that never fired: `/prune`’s plan leaves such an item out, since no sweep could delete it, so the item stayed without being measured. Both results miss the margins of their hypotheses (Sections 5.3 and 5.5). On a real open-source repository the agent was rarely corrected, so nothing recurred and the loop learned nothing. Finally, Cortex can consult an optional *judge*, an outside model that advises which candidates to try (Section 3.7). It was switched off during the runs; replayed on their candidates afterwards, it did not anticipate the verdicts (Section 5.7).

Contributions.

1. **Exposure-aware measurement semantics** for harness items, where an item’s *exposure* is whether it entered the agent’s context in a rollout: the same rule governs adding and deleting, so that both verdicts compare the harness with and without the item only on the tasks where it entered the context, a candidate that never fired is not kept, an item that never fired is not deleted, and unknown exposure blocks the verdict (Section 3).
2. **A known-answer test of the acceptance pipeline**: 20 placebo, 5 harmful and 4 known-good candidates run through it. It shows that a candidate that never fires is never kept; it could not test harm, and the test’s other weaknesses are reported (Section 5.3).
3. **Evidence on loading tiers**: why an item the agent must choose fares differently under measurement from one that is simply loaded (Sections 5.2 and 5.4).
4. **A replicated evaluation** with a control run in which /evolve is off, an A/A arm (the same harness measured twice, to show how far two measurements of one harness can differ), ablations (other harnesses, each built to answer one question, such as the evolved harness with its paths removed), a model-change study and a real-repository attempt. Every number can be regenerated offline from the released data: rows exported from the runs’ records and from Claude Code’s session records (the records themselves are not released).
5. **The open-source tool and its evaluation lab**, at <https://github.com/samugit83/cortex>: tag v1.0-eval is the system as evaluated, and tag v1.0.5 adds the lab’s data and the scripts that regenerate every number in the paper.

2 Background

The harness and its loading tiers. Claude Code reads three kinds of text a developer writes [4, 5]. A `CLAUDE.md` at the root of a repository holds project instructions and is read at the start of every session; recent versions of Claude Code read an `AGENTS.md` in its place when a repository has no `CLAUDE.md`, and tailoring a coding agent to a repository with such a context file is widespread practice [16]. A *rule* is a Markdown file in `.claude/rules/`. It may carry a `paths` field, a list of file patterns such as `shop/billing/**` (every file under `shop/billing/`). A *path-less* rule, one without the field, is loaded at launch, like `CLAUDE.md`; a *path-scoped* rule enters the context when the agent reads a file its patterns match, and, as Cortex’s tests found on version 2.1.276, when the request names such a file with an @-mention. A *skill* is a folder in `.claude/skills/` holding a `SKILL.md`: a short description, then a body with the instructions. Only the description is kept in context, in a listing of the available skills; the body is loaded when the skill is *invoked*: the agent opens it with a call to Claude Code’s Skill tool, or the user types its name as a command. A skill without `paths` is *always-on*: its description is listed on every turn. A *path-gated* skill, one with `paths`, is listed only once the agent works with matching files. Every description in the listing costs context on every turn, whether or not its skill is ever used [5]. Table 1 summarises these five *loading tiers*. The difference that matters most in this paper is between an item that is *loaded* and one that must be *chosen*: Claude Code itself puts a rule into the context, but a skill acts only if the agent, reading its description among the others, decides to open it.

Table 1: Loading tiers in Claude Code [4, 5]. The rows for items with `paths` are as Cortex verified them on version 2.1.276, the oldest it accepts; the current documentation also counts writes and edits among the triggers of a path-scoped rule. The tier, not the text, decides how often an item is paid for, and whether it is *loaded* (rules) or must be *chosen* by the agent (skills).

Tier	What sits in context	When
root <code>CLAUDE.md</code>	the whole file	every turn
always-on skill	its description; the body when invoked	every turn
path-less rule	the whole rule	every turn
path-gated skill	its description; the body when invoked	after a matching file is read or written
path-scoped rule	the whole rule	after a matching file is read or @-mentioned

Tasks with a check. Coding agents are evaluated on tasks cut from real repositories: a codebase at a commit where something is broken, a request in natural language, and tests that decide whether the agent’s change fixes it. SWE-bench

keeps only task instances with at least one test that fails before the reference solution, the fix the project’s developers actually merged, and passes after it [22]; SWE-rebench extracts such tasks automatically and continuously, so that models can be scored on fresh tasks they cannot have seen [6]. Cortex keeps the format but takes its tasks from the developer’s own sessions rather than from public issues (Section 3.3).

One run is not a measurement. An agent’s attempt at a task is a random draw: the model samples its output, so two attempts at the same task can take different paths through the repository, and one may pass where the other fails. A task therefore has a pass *rate*, and a single pass or fail estimates it poorly. Three studies show how noisy one run is. On the retail tasks of τ -bench, a GPT-4o agent, then the state of the art, solved about three tasks in five on a single attempt, but solved the same task in all eight of eight attempts less than a quarter of the time [63]. In the published repeated trials of a coding-agent benchmark, fewer than one task in five had a one-run result that reliably tracked an agent’s score on the whole benchmark [13]. When the skill files of three repositories were optimised automatically, the larger of two optimisers’ average gains, about five points, could not be separated from the agent’s run-to-run variance with the tasks one repository supplies [26]. Treating an evaluation as an experiment, Miller recommends two remedies [36]: sample each question more than once, so that the luck of single attempts averages out; and compare two systems through their paired, question-level differences rather than their two overall scores, so that what makes a question easy or hard for both systems cancels. Cortex does both: it runs every task k times under the live harness and k times under the candidate, and compares the two task by task (Section 3.4).

3 Cortex

Cortex changes only text: the files that make up a Claude Code harness. The model is fixed. This section describes Cortex as it stands at the git tag `v1.0-eval`, a named snapshot of its code, with that snapshot’s default settings. Every experiment of Section 4 was made on this version, except the development run D0, which used an earlier one. No run changed a gate’s threshold, so the rules that decide whether an item is kept are the same in every experiment. The runs changed a few other settings, none of which decides a verdict; Section 4 lists them.

Section 3.1 follows the three commands from start to finish. The subsections after it give the reasons for each step: what an item costs (Section 3.2), where the tasks come from (Section 3.3), how a change is measured (Section 3.4), when a new item is kept (Section 3.5), and when an old one is deleted (Section 3.6). Table 2 collects the terms they use, for reference.

3.1 The three commands, step by step

Cortex consists of three Claude Code commands that the developer runs, each with a lifecycle of its own, joined into one loop (Figure 1). The commands are prompts that the model follows; for every step that must be exact (running a sweep, scoring it, recording a cycle), they call a command-line program, `cortex`, whose subcommands, such as `cortex cycle` or `cortex scope`, appear below. `/harvest` runs at the end of a working session and turns what happened into evidence: a replayable task when something went from broken to working, and a one-line lesson for each correction (Figure 2). `/evolve` runs one cycle at a time: it looks in that evidence for one recurring problem, writes one skill or rule for it, and measures it against the live harness before it is kept (Figure 3). `/prune` is meant to run monthly and after every model change: it measures what removing each item of its plan costs, one item at a time, and deletes only what costs nothing measurable (Figure 4). The three figures follow each command from start to finish, including every point where it stops early; the rest of this section gives the reasons for each step.

The three commands as algorithms. Algorithms 1 to 3 give the same three commands as short pseudocode, in the order a developer meets them. `/harvest` (Algorithm 1) only records: it turns a session into a task, lessons, or nothing, and never runs the agent. `/evolve` (Algorithm 2) adds: it picks one recurring problem, writes one skill or rule for it, and keeps it only if a measurement against the live harness says so. `/prune` (Algorithm 3) removes: with the developer’s approval, it measures what taking each item out costs and deletes only the items whose removal costs nothing measurable. The scores and thresholds they use (G, N, D, f, P, m, τ) are defined in Sections 3.4 and 3.5.

Table 2: The terms this paper uses for Cortex, in roughly the order a reader meets them, with the subsection that explains each in full.

Term	Meaning	See
harness	the text files that steer the agent: CLAUDE.md, skills and rules	Section 2
item	a skill or a rule, the harness files Cortex measures	Section 3.2
live harness	the harness the developer is using now	Section 3.4
loading tier	when an item’s text enters the agent’s context (Table 1)	Section 2
task	an episode /harvest saved so it can be replayed: a broken commit, the request, a check and the fix	Section 3.3
check	a script that exits zero when an attempt solved the task	Section 3.3
quarantine	set aside a task whose check no longer tells the broken state from the fixed one, or needs the fix’s own test code	Section 3.3
lesson	one logged line for each correction the developer made	Section 3.1
theme	a problem that recurs in the lessons or the failing tasks	Section 3.5
candidate	a new skill or rule under test, not yet live	Section 3.5
rollout	one attempt by the agent at one task, in a sandbox clone of the repository	Section 3.4
sweep	a batch of rollouts that runs each of a set of tasks k times under each of two harnesses	Section 3.4
run (of a task)	one of the k rollouts of a task under one arm of a sweep; not to be confused with an evaluation run, R1 to R4 (Section 4)	Section 3.4
pinned model	the one model that every rollout uses, when the developer names it in Cortex’s configuration	Section 3.4
<i>base, cand</i>	the two arms of a sweep: the live harness, and the live harness with the change under test	Section 3.4
exposed, fire	a task is <i>exposed</i> to an item when the item reached the context in one of its rollouts (a rule loaded, a skill at least listed); an item <i>fires</i> when it is used (a rule loaded, a skill opened)	Section 3.4
cycle	one iteration of /evolve: find a theme, write one candidate, measure it	Section 3.5
screen, confirm	the cheap first sweep, on the theme’s failing tasks; the stricter second one, on every task	Section 3.5
gate	one condition a candidate must meet to be kept, computed by the scoring script <code>score.sh</code> (Table 3); gates 2 and 3, which stop a candidate that makes a task worse, are the <i>regression gates</i>	Section 3.5
recheck (RECHECK)	a repeat sweep of the tasks that dropped, when a candidate failed only the regression gates, before its KILL is final	Section 3.5
KEEP, KILL	the candidate goes live; the candidate is buried	Section 3.5
BARREN	a cycle that found no theme strong enough and ran no rollouts	Section 3.5
RERUN	the sweep cannot decide, because its data cannot be trusted or none of the tasks the candidate reached could show a gain; it is repeated and nothing is decided	Section 3.5
ACCEPT, REJECT, UNMEASURED	/prune’s verdicts: delete the item; keep it, because removing it cost something; keep it, because it never fired, so its removal tested nothing	Section 3.6
graveyard	the folder of buried items, each with the reason it was buried	Section 3.5
journal	the file in which Cortex records every cycle and verdict	Section 3.1

3.2 Harness items and what they cost

Cortex calls each skill or rule a *harness item*; the harness also holds constant files such as CLAUDE.md, which Cortex never measures. When a file enters the context decides what it costs (Table 1): its price is its length multiplied by how often it is present, so Cortex counts the characters present on every turn (CLAUDE.md, always-on skill descriptions and path-less rules) and warns above a budget (3,000 characters by default). Its objective is informally pass rate $-\lambda \cdot$ always-on context, where λ is the pass rate a developer would give up to save one character read on every turn. Cortex never computes this as one number. It enforces it by two rules on the items it measures, skills and rules: add one only if it raises the pass rate (Section 3.5); remove or narrow one only if that does not lower it (Section 3.6). When /evolve decides that a theme belongs in CLAUDE.md, a setting, a hook or a tool rather than in a skill or rule, it writes the change and records it in the journal, but does not measure it: the gates apply only to skills and rules. The idea of context cost as an objective is shared with Meta-Harness and SkillReducer [15, 27]. Whether an item is loaded or must be chosen also shaped what survived in our evaluation (Figure 18).

/harvest, at the end of a working session

No rollouts: it only records what happened, and never writes a skill (that is /evolve's job).

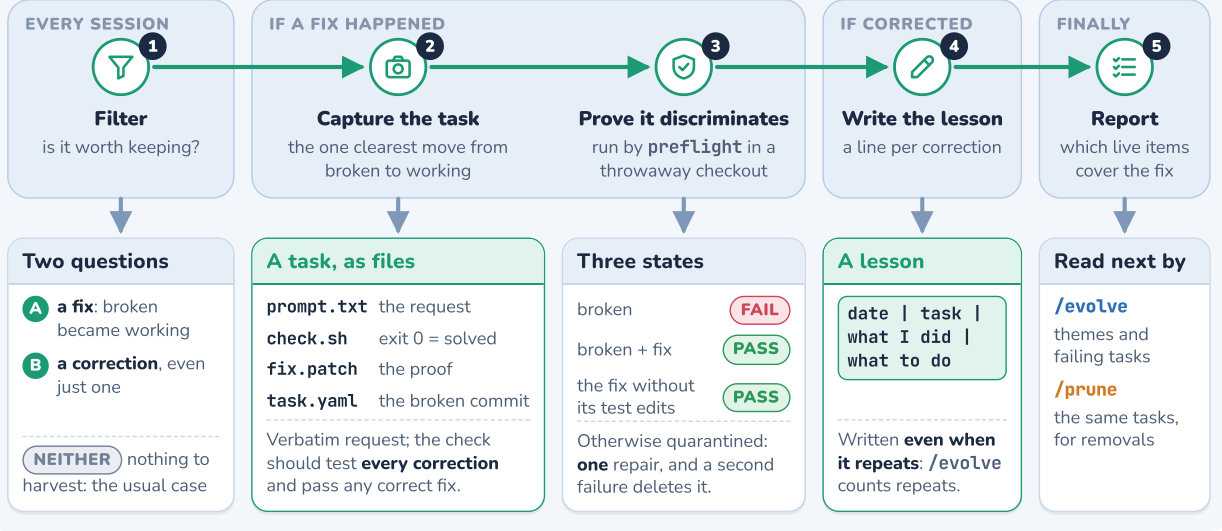


Figure 2: What /harvest does at the end of a working session. It asks two questions about the session (step 1). **A:** did something go from broken to working? **B:** did the developer correct the agent, even once? In most sessions the answer to both is no, and nothing is recorded. If the answer to A is yes, the command saves that change as a *task* that can be replayed later (step 2): the developer's request, word for word; the commit where the code was still broken; the fix that made it work; and a check, a short script that says whether an attempt solved the problem. Before the task is kept, *preflight* tests the check itself in a throwaway copy of the repository (step 3). The check must fail on the broken code, pass once the fix is applied, and still pass when the fix's changes to test code are left out, so that an agent is judged on the code it writes, not on reproducing the developer's own test. A task that fails any of the three is set aside and gets one repair; if it fails again, it is deleted (Section 3.3). If the answer to B is yes, the command writes one line per correction to a log of *lessons*: the date, the task, what the agent did, and what it should have done (step 4). A lesson is written even when the same one is already there, because /evolve counts the repeats to find the problems that recur. Finally, the command reports which items of the current harness already cover the files the fix touched (step 5). /harvest only records: it never runs the agent on a task and never writes a skill or rule, which is /evolve's job. The tasks it saves are used next by /evolve, to measure new candidates, and by /prune, to measure what removing an item costs.

Algorithm 1 One /harvest. It runs no rollouts and writes no skill or rule: it records a task, lessons, or nothing.

Require: the session that just ended; the repository, with the fix still in place

- 1: $A \leftarrow$ something went from broken to working during the session
 - 2: $B \leftarrow$ the developer corrected the agent at least once
 - 3: **if** neither A nor B **then**
 - 4: report "nothing to harvest" and stop ▷ the usual case
 - 5: **if** A **then**
 - 6: $t \leftarrow$ a task from the clearest move from broken to working: the broken commit, the fix, the request verbatim, and a check
 - 7: preflight every task, t included, in a throwaway checkout: fail broken, pass fixed, pass fixed minus its test edits ▷ Section 3.3
 - 8: **if** t is quarantined **then**
 - 9: repair the check once and preflight t again; delete t if it is quarantined again
 - 10: **if** B **then**
 - 11: append one lesson per correction, even one already logged ▷ /evolve counts repeats
 - 12: report which live items cover the files the fix touched
-

One /evolve cycle: one candidate, measured before it is kept

The model proposes and writes the candidate; every verdict comes from the scorer, never from the model.

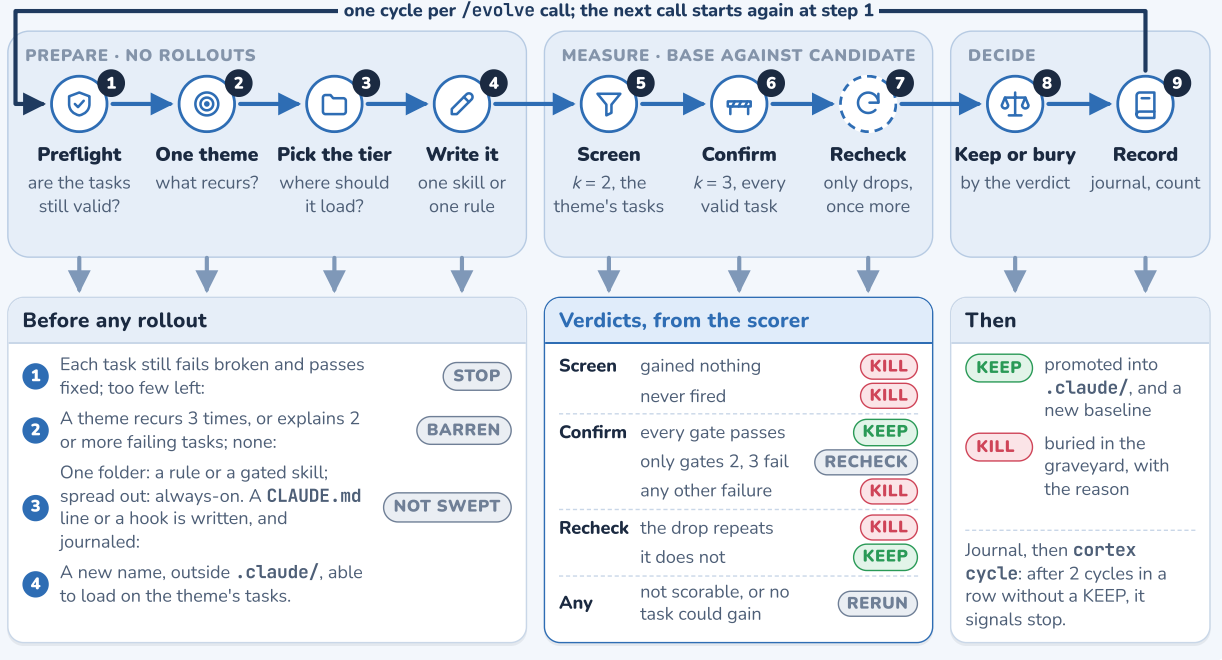


Figure 3: One /evolve cycle, which tests a single new skill or rule before it may join the harness. The model proposes and writes the candidate, but never decides its fate: every verdict comes from a scoring script (Table 3). Steps 1 to 4 prepare the cycle and run the agent on no task; the first three can end the cycle early (grey badges). Step 1 re-runs preflight to make sure the saved tasks still tell broken code from fixed code; if too few are left, the cycle stops. Step 2 looks for one *theme*, a problem that recurs: a lesson logged at least 3 times, or a cause behind 2 or more failing tasks. If none clears that bar, the cycle ends as **BARREN** and runs no rollouts. Step 3 chooses where the candidate should load: if the fixes behind the theme sit in one folder, it becomes a rule or a skill tied to that folder; if they are spread across the repository, it becomes a skill offered on every turn. A fact the agent breaks during unrelated work becomes a line in **CLAUDE.md**, and a check that must always run becomes a *hook*, a command Claude Code runs by itself at a fixed point of a session; such a change is written and recorded in the *journal*, Cortex's running log of every cycle and verdict (**.evolve/journal.md**), but not measured. Step 4 writes the candidate under a new name, outside the live harness. Steps 5 to 7 measure it: the agent attempts each task with the live harness (*base*) and with the harness plus the candidate (*cand*), each attempt in its worker's own clone of the repository, reset and verified before every attempt (Figure 5). The *screen* (step 5) is cheap, 2 runs per task on the theme's tasks only; a candidate that gained nothing, or that never fired (a rule never loaded, a skill never opened), is killed here. The *confirm* (step 6) is stricter, 3 runs per task on every valid task, so that it also sees what the candidate might break. The candidate is kept only if it wins overall, no task collapses, no task that never failed starts failing, and it fired at least once. Because one unlucky run can look like a broken task, a candidate that failed only on such a drop gets a *recheck* (step 7, drawn dashed): the tasks that dropped are run again, and the candidate is killed only if the drop repeats. A sweep whose data cannot be trusted, with runs missing or invalid, or in which no task the candidate reached could show a gain, is repeated rather than decided (**RERUN**). Step 8 acts on the verdict: a **KEEP** moves the candidate into the live harness and records the confirm sweep's scores as the new baseline, with no new rollouts; a **KILL** buries it in the *graveyard*, the folder of rejected ideas (**.evolve/graveyard/**), with its reason, so the same idea is not proposed again unchanged. Step 9 records the cycle in the journal; after 2 cycles in a row without a **KEEP**, cortex cycle signals the loop to stop. Badge colours follow Figure 1: green, the item stays; red, it goes; grey, no decision.

Algorithm 2 One /evolve cycle. The model writes the candidate; every verdict comes from `score.sh`. A sweep that is not scorable, or in which the candidate reached tasks but none could show a gain, is repeated, never decided.

Require: live harness H ; task store; logged corrections; graveyard

```

1: quarantine every task whose check no longer separates broken from fixed, or needs the fix's own test code      ▷ Section 3.3
2: if too few valid tasks remain, or the routing check finds an error (a name used by both a skill and a rule, or a symlink that leaves
   the repository or dangles) then
3:   stop                                                                                              ▷ no rollouts
4:  $\theta \leftarrow$  a theme logged at least 3 times, or behind 2 or more failing tasks, that no live item covers (a buried idea may return only
   revised, under a new name)
5: if there is no such  $\theta$  then
6:   record the cycle as BARREN and stop                                                                 ▷ no rollouts
7:  $x \leftarrow$  one skill or rule for  $\theta$ , in the tier its fixes point to                                  ▷ Table 1
8: screen: sweep  $H$  against  $H + x$  on the tasks behind  $\theta$ ,  $k = 2$ ; repeat it while it is not scorable, or while  $x$  reached some task but
   none could show a gain                                                                                              ▷ RERUN
9: if  $G = 0$  or  $f = 0$  then
10:  bury  $x$  and stop                                                                                   ▷ no confirm is paid for
11: confirm: sweep  $H$  against  $H + x$  on every valid task,  $k = 3$ 
12:  $v \leftarrow$  the gates of Table 3
13: if  $v = \text{RECHECK}$  then
14:  sweep the regressed tasks again;  $v \leftarrow \text{KILL}$  if one regresses again, else KEEP
15: if  $v = \text{KEEP}$  then
16:  make  $x$  live
17: else if  $v = \text{KILL}$  then
18:  bury  $x$  with the gates it failed

```

3.3 Tasks from the developer's own sessions

A task is a replayable episode: a broken commit (`base_sha`), the developer's original request, a `check.sh` that exits zero when the task is solved, and the `fix.patch` that proves it can be. `/harvest` writes one at the end of a session, and only when something went from broken to working. The request is the developer's original words, not a cleaned-up version with hints known only afterwards. `/harvest` is instructed to write a check that tests not only that the bug is fixed but also each correction the developer made during the session (for example, that the linter is clean), so that an item which prevents the corrected mistake can turn a fail into a pass, and show an effect; preflight confirms that the check separates broken from fixed, not that it covers every correction. Most sessions produce no task, by design.

As in the benchmarks of Section 2, a task is only a ruler if its check separates the broken state from the fixed one. preflight runs the check in three states before every cycle and every prune pass: the broken state must *fail*; the broken state plus the fix must *pass*; and, when the fix edits test code, the fix *without* those edits must still pass (test data it adds, such as a fixture, stays). Write σ_t for the broken commit, δ_t for the fix, δ_t^{test} for the fix's edits to test code, $\sigma \oplus \delta$ for the commit σ with the edits δ applied, and $c_t(\cdot)$ for the result of the check: *pass* when it exits zero, *fail* when it exits with any other code, and *hang* when it does not finish within its time limit. A task is valid when

$$\text{valid}(t) \iff c_t(\sigma_t) = \text{fail} \wedge c_t(\sigma_t \oplus \delta_t) = \text{pass} \wedge c_t(\sigma_t \oplus (\delta_t \setminus \delta_t^{\text{test}})) = \text{pass}, \quad (1)$$

so a check that hangs fails preflight in any state. When the fix edits no test code, δ_t^{test} is empty and the third condition repeats the second. The third state exists because a developer often writes a test while fixing a bug. If the check depended on that test, an agent replaying the task would have to write the same test again, which a replaying agent has no reason to do, and the task would fail in both arms forever. A task that fails any state is *quarantined*: it is set aside and left out of every sweep. One whose environment is down, as reported by an optional script that checks its services, is skipped instead, since an outage is not a bad task.

3.4 Measurement: against the live harness, with exposure

Every decision rests on a *sweep*: the agent runs each task k times under two harnesses. *base* is the harness as it is now, re-measured in every sweep; *cand* is the same harness plus the one change under test (or minus it, for a removal). Both arms load every existing item, so what is measured is the change's *marginal* value: a near-duplicate of an item the

One /prune pass: an item is deleted only when removing it costs nothing measurable

Monthly, and after every model change. Nothing runs until the developer approves the plan.

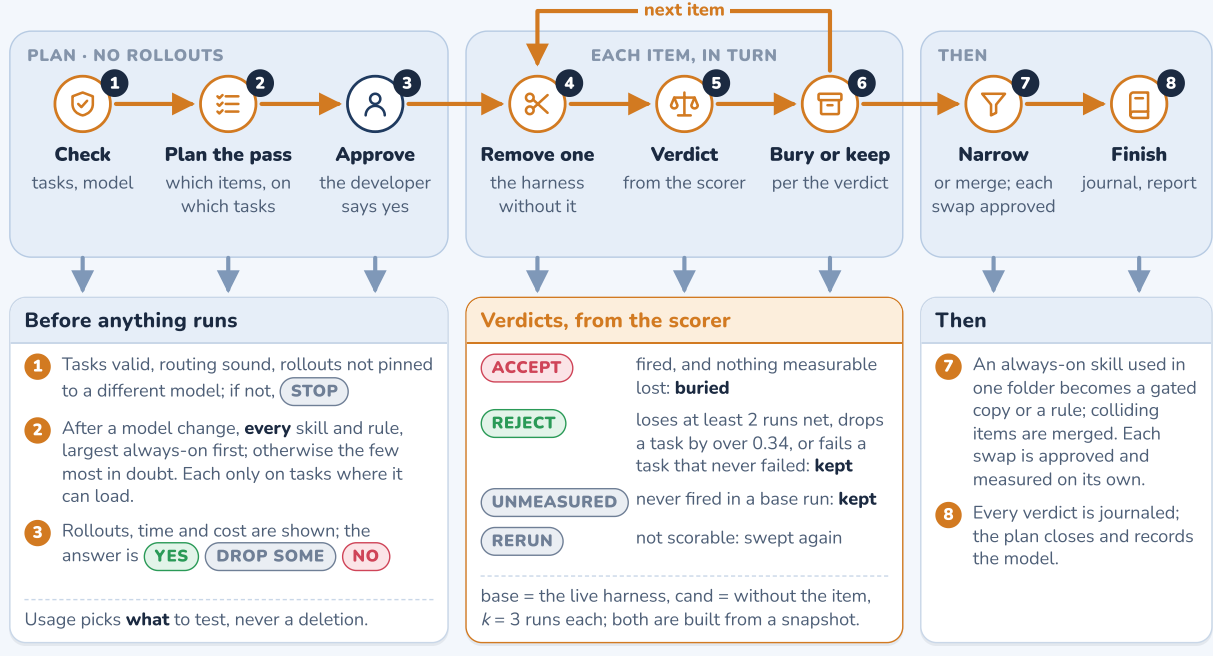


Figure 4: One /prune pass, which tests the items already in the harness and deletes one only when removing it costs nothing measurable. It is meant to run monthly and after every change of model, since a stronger model may already do what an item teaches. Steps 1 to 3 plan the pass and run the agent on no task. Step 1 checks that the tasks are still valid, that the routing is sound (no name is both a skill and a rule, no symlink leaves the repository or dangles), and that the rollouts are not pinned to a model other than the one in use; if any check fails, the pass stops. Step 2 chooses which items to test: after a model change, every skill and rule, the largest always-on items first; otherwise, the few most in doubt. Each item is tested only on the tasks where it can load; an always-on skill, only on the tasks where earlier sweeps saw it fire, and not at all if they never did. How often an item is used helps choose what to test, but never decides a deletion. Step 3 shows the developer the plan with its expected rollouts, time and cost, and nothing runs until the developer approves it, possibly dropping some items. Steps 4 to 6 then take one item at a time: the agent attempts each of its tasks 3 times with the live harness (*base*) and without the item (*cand*), and the scoring script gives the verdict. **ACCEPT**: the item fired in at least one base run, and removing it lost nothing measurable, so it is deleted and buried in the graveyard. **REJECT**: removing it lost too many runs, made a task collapse, or made a task that never failed start failing, so it stays. **UNMEASURED**: the item never fired in any base run, so the test says nothing about it, and it stays; an item for a rare situation looks exactly like dead weight until that situation comes. **RERUN**: the data cannot be trusted, and the sweep is repeated. A deleted item is removed before the next one is measured, so every test starts from the harness as it really is. Step 7 can also move advice to a better place: an always-on skill used in only one folder can become a rule or a skill tied to that folder, and overlapping items can be merged; each such change is approved and measured on its own. Step 8 records every verdict in the journal and closes the plan, noting the model it ran on (Section 3.6).

developer already has correctly shows no gain. The two arms' rollouts are interleaved (for each task and each run, *base* and then *cand*), so a slowdown, or a new model deployment in the middle of a sweep, lands on both arms alike. The harness and task definitions are snapshotted at the start. Rollouts run in parallel, one per *worker*, and each worker has its own clone of the repository (Figure 5), reset before every rollout to the task's broken commit with that arm's harness; every reset is verified by running the check, which must fail. The clone never has Cortex's own folder (*.evolve/*) on disk: that folder holds the *task store*, where Cortex keeps its saved tasks with every earlier task's fix, and the lessons and graveyard, which would give the answer away, so the clone is made without it (a sparse checkout, which leaves chosen folders out) and every reset deletes it again. It is still reachable through the clone's git history, which is not pruned; Section 7 discusses that route.

Finally, the model that plays the agent can be fixed: by default each rollout uses whatever model Claude Code is set to, but the developer can name one model in Cortex's configuration (*pin* it), so that every rollout of every sweep uses the same one.

Algorithm 3 One /prune pass. Nothing runs until the developer approves the plan, and every verdict comes from `score.sh`. Here *base* is the live harness and *cand* the harness without the item, so a negative *N* means the item was earning its place.

Require: live harness *H*; task store; each item’s use in recent sessions; the model of the last pass

```

1: quarantine every task whose check no longer separates broken from fixed, or needs the fix’s own test code      ▷ Section 3.3
2: if too few valid tasks remain, the routing check finds an error (a name used by both a skill and a rule, or a symlink that leaves
   the repository or dangles), or the rollouts are pinned to a model other than the one in use then
3:   stop                                                                                                                                            ▷ no rollouts
4: if the pinned model changed since the last pass then
5:    $X \leftarrow$  every skill and rule, the largest always-on first
6: else
7:    $X \leftarrow$  the few items most in doubt                                                                                                      ▷ use chooses, never deletes
8: drop from  $X$  every item no task can load, and every always-on skill that never fired in an earlier sweep ▷ no sweep could delete
   it
9: show the plan, with its rollouts, time and cost; stop unless the developer approves it
10: for each  $x \in X$ , in order do
11:   sweep  $H$  against  $H - x$  on the tasks where  $x$  can load (an always-on skill: where it fired before),  $k = 3$ 
12:   if the sweep is not scorable then
13:     sweep again                                                                                                                                ▷ RERUN: nothing is decided
14:   else if  $kN \leq -m$ , or  $D > \tau$ , or a task in  $P$  loses a run then
15:     keep  $x$                                                                                                                                     ▷ REJECT: removing it costs something
16:   else if  $f = 0$  then
17:     keep  $x$                                                                                                                                     ▷ UNMEASURED: it never fired
18:   else
19:     bury  $x$ ;  $H \leftarrow H - x$                                                                                                                 ▷ ACCEPT, before the next item
20: narrow always-on items, or merge overlapping ones, each swap approved and swept on its own
21: journal every verdict; close the plan and record the model

```

Exposure. An item can change a result only in a rollout where it entered the agent’s context. Cortex reads each rollout’s *event stream*, the record of every message and tool call that Claude Code emits while the agent works. From it, Cortex records for each skill whether it was *visible* (listed to the agent) and whether it was *invoked* (opened with a Skill tool call). A rule leaves no trace of its own, so Cortex records that it *fired* when the agent read a file its paths match, or the prompt @-mentioned one. When the stream cannot be read the value is *unknown*, never “did not fire”. A task is *exposed* to an item if the item reached the context in any of its rollouts: a rule that loaded, or a skill that was at least listed, since its description then sits in the context. An always-on skill is listed on every turn, so every task is exposed to it. On a task that was not exposed, the agent in both arms read the same text, and Cortex treats any difference between the arms as chance. Whether the item *fired*, a rule loaded or a skill opened, is a stricter question, which gate 5 asks (Section 3.5). This one principle drives four decisions: the gates read only exposed tasks; a candidate that never fired is never kept; a removal whose item never fired is never accepted; and unknown exposure in any valid rollout blocks the verdict, so that a tooling failure cannot look like dead weight. Crediting only what fired is shared with SkillMAS [39], and it guards against inert edits such as the never-triggered hook that DemoEvolve’s audit found in a harness chosen by self-rollout search [9]. It has been applied to deletion as well: SLIM retires a skill on a comparison with and without it over the validation tasks on which it was loaded, and only once it has been loaded often enough [47], while Bayesian-Agent and Ratchet retire a skill only on enough evidence from the runs that used it, scoring the skill’s own outcomes rather than the difference its removal makes [56, 70]. Cortex applies the same rule to adding and to deleting, and lets unknown exposure block either verdict.

Scores. Inside a sweep, each of the k rollouts of a task under one arm is called a *run* of that task; this short sense of “run” is unrelated to the evaluation runs R1 to R4 of Section 4. Let k be the runs per task and arm, and $p_b(t)$ and $p_c(t)$ the fraction of the valid rollouts of task t that passed under *base* and *cand*. A task is *measured* when both arms have k valid rollouts on it, and *exposed* when the item under test was in context in at least one of them: the candidate in a *cand* rollout or, for a removal, the removed item in a *base* rollout. Write $n_b(t)$ and $n_c(t)$ for the numbers of valid rollouts of task t under each arm; Ω_t^* for its valid rollouts in the arm where the item x under test exists (*cand* for a candidate, *base*

One rollout, in its own sandbox

base and candidate take the same steps, alternating; only the installed harness differs

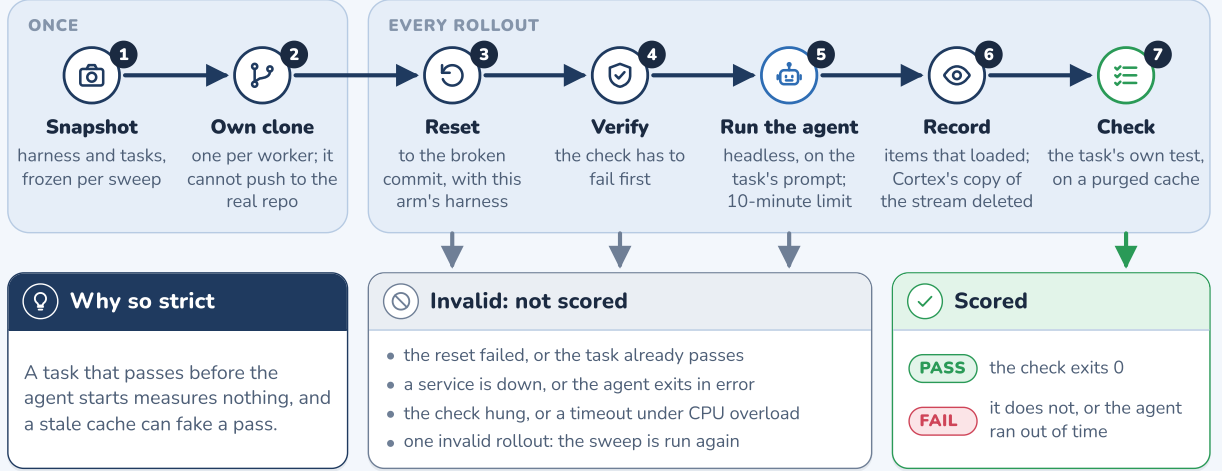


Figure 5: What happens in one rollout, a single attempt by the agent at a task. The *base* and *cand* arms take exactly the same steps, in alternation, and differ only in the harness that is installed. Once per sweep, Cortex freezes a copy of the harness and of the tasks (step 1), so that an edit made while the sweep runs cannot reach it, and gives each worker its own clone of the repository, which cannot push to the real one (step 2). Then, before every rollout, the clone is reset to the task’s broken commit and that arm’s harness is installed (step 3), and the task’s check is run once to confirm that it still fails (step 4): a task that already passes before the agent starts would measure nothing. The agent then works alone, with no human, on the task’s request, within a limit of 10 minutes (step 5). Cortex records which skills and rules entered the agent’s context during the attempt (step 6). Finally, the task’s own check decides the result, after the cached build files are deleted, since a stale cache could make old code look as if it passes (step 7). Each attempt ends in one of two ways. It is *scored*: a pass if the check succeeds, a fail if it does not or the agent ran out of time. Or it is *invalid*, because it could not run fairly: the reset failed, the task already passed, a service the task needs was down, the agent exited with an error (for example a login failure or a rate limit), the check hung, or the agent ran out of time while the machine was oversubscribed. Invalid is not the same as fail, and a sweep with even one invalid rollout is run again rather than scored.

for an item being removed); and, for a rollout ω , $\text{ctx}(\omega)$ for the items that were in the agent’s context and $\text{fired}(\omega)$ for those that fired, both read from its event stream. The measured tasks $\mathcal{M}$, the exposed tasks $\mathcal{X}$, the set S of tasks that are both, and the count f of rollouts in which the item fired, which gate 5 reads (Section 3.5), are

$$\begin{aligned}\mathcal{M} &= \{t : \min(n_b(t), n_c(t)) \geq k\}, & \mathcal{X} &= \{t : \exists \omega \in \Omega_t^*, x \in \text{ctx}(\omega)\}, \\ S &= \mathcal{M} \cap \mathcal{X}, & f &= \sum_t \sum_{\omega \in \Omega_t^*} \mathbb{1}[x \in \text{fired}(\omega)],\end{aligned}\tag{2}$$

where $\mathbb{1}[\cdot]$ is one when its condition holds and zero otherwise, and t in the sum runs over every task of the sweep. On S , the per-task change and the three scores are

$$\Delta_t = p_c(t) - p_b(t), \quad G = \sum_{t \in S} \max(\Delta_t, 0), \quad N = \sum_{t \in S} \Delta_t, \quad D = \max_{t \in S} \max(-\Delta_t, 0),\tag{3}$$

the *gain*, the *net* change and the *worst drop*. Each answers a different question about the change.

- Δ_t , the *per-task change*, is how much more often task t passed with the change than without it. It is +1 when the candidate passed every run and the base none, −1 in the opposite case, and 0 when both arms did equally well; in between it moves in steps of $1/k$, one run at a time. The other scores are built from it.
- G , the *gain*, adds the tasks that improved and ignores those that got worse: it is everything the change won, whatever it lost. A candidate with $G = 0$ helped on no task, so there is nothing to set against its cost, and the screen stops it before the confirm is paid for (or is rerun, if it reached tasks but none could show a gain).
- N , the *net* change, adds every Δ_t , so a loss on one task cancels a win on another: it says whether the change is still ahead once what it broke is counted. Every Δ_t is a multiple of $1/k$, so kN is a whole number, the runs won

net of those lost. The gates ask for $kN \geq m = 2$, because with $k = 3$ runs per arm a net of one run can come from luck alone.

- $G - N$, the *regression*, is what the change cost: taking the net away from the gain leaves only the losses.
- D , the *worst drop*, is the largest loss on any single task, and zero when S is empty. A net that is comfortably positive can hide one task the change broke; D exposes it.

For a removal, *cand* is the harness without the item, so the same scores read the other way: a negative N says the item was earning its place (Section 3.6).

Δ_t , N and $G - N$ are DarwinX’s per-task change, net gain and bounded regression [72]. The gates below use them differently in two ways. They keep DarwinX’s requirement that a change improve some task, and raise its requirement of a positive net to a net of at least m runs, since luck alone can win a single run. And they bound the single worst drop rather than the summed regression, because the sum cannot tell a broken task from scattered noise. Take $k = 3$. One task whose pass rate drops from $2/k$ to 0, and two tasks whose pass rates each drop from $2/k$ to $1/k$, both give $G - N = 2/k$, but only the first is a task the change broke. The first fails gate 2 ($D = 2/k$); the second passes it ($D = 1/k$), and its two lost runs still count against kN . In vector form, with $\Delta_t^+ = \max(\Delta_t, 0)$ the gains and $\Delta_t^- = \max(-\Delta_t, 0)$ the losses over $t \in S$, the gain and the regression are ℓ_1 norms and the worst drop is an ℓ_∞ norm:

$$G = \|\Delta^+\|_1, \quad G - N = \|\Delta^-\|_1, \quad D = \|\Delta^-\|_\infty, \quad D \leq G - N \leq |\{t \in S : \Delta_t < 0\}| \cdot D. \quad (4)$$

The same regression can therefore come from one task that lost $G - N$, where it equals D , or be spread over several tasks that each lost at most D ; D is what tells the two apart.

Two more quantities complete the gates. The *protected set* $P = \{t \in S : p_b(t) = 1\}$ holds the tasks the live harness never failed: the ones the developer already relies on, where a regression costs most. Gate 3 therefore gives them no tolerance: the candidate must pass every one of their runs too. Finally, f (Equation (2)) counts the valid rollouts in which the item *fired*: a skill was invoked, or a rule’s paths matched a file the agent read or the prompt @-mentioned. It is counted where the item exists, among the *cand* rollouts for a candidate and the *base* rollouts for an item being removed. The scores measure a difference between the arms, and f ties that difference to the item: a candidate that never fired cannot have caused its gain, and an item that never fired was not tested by its removal.

3.5 Adding an item: proposal and gates

Evidence before proposals. */evolve* improves the harness one *cycle* at a time (Algorithm 2): a cycle looks for one recurring problem, writes one item meant to fix it, and tests that item before the next cycle starts. It looks for the problem in four places: the corrections the developer made to the agent, which */harvest* logs as one line each; the tasks the live harness currently fails; the developer’s recent transcripts; and the *graveyard*, the items already tested and rejected, so that a rejected idea is not proposed again unchanged. A problem that recurs is a *theme*, and a cycle acts on one only when the evidence is strong enough: the theme recurs at least 3 times, or it explains 2 or more failing tasks. A single correction may be a one-off, and testing an item costs rollouts, so when no theme clears this bar the cycle is *BARREN*: it proposes nothing and runs no rollouts. This is an expected outcome, not a failure. When 2 cycles in a row end without a *KEEP*, whether barren or killed, *cortex cycle* signals that the loop should stop (the tasks may no longer tell a good harness from a bad one, the harness may already be good, or nothing new is being harvested), and */evolve* stops and tells the developer. The developer may still start another cycle; nothing enforces the stop.

When a theme clears the bar, the cycle proposes one candidate, and the evidence also chooses its tier (Section 3.2) from where the fixes of the failing tasks landed. If they cluster in one directory, the candidate loads only there: a path-scoped rule when the fixes edit files that already exist, since a rule is injected whenever the agent reads a matching file; or a path-gated skill when the fixes only create new files, since a skill is also offered when a matching file is written. Its paths use the narrowest glob that covers every fix, because an item costs context each time it loads. If the fixes are spread across the repository, no directory fits, and the candidate is an always-on skill tied to a moment rather than a place, such as “before reporting done”.

Two stages. A cheap *screen* runs the candidate at $k = 2$ on the theme’s currently failing tasks and stops if it gained nothing or never fired, or is rerun if it reached tasks but none could show a gain (below). It cannot show what a change breaks, because it never runs the tasks that pass; only candidates that survive it pay for the *confirm*, at $k = 3$ over every valid task. A cheap screen followed by a stricter confirmation is adapted from DarwinX [72]. How many rollouts each stage took across the evaluation is shown in Figure 15.

The gates. The confirm verdict is computed by a script, not by a model (Table 3); Figure 6 shows one confirm sweep, scored again from its rollouts. A change must win something *and* lose nothing that matters: DarwinX’s contract of extending without breaking what is preserved [72]. The net threshold in runs mirrors SkillGen’s [34], and the no-regression gates HarnessX’s refusal to regress any solved task and Self-Harness’s non-regressive acceptance [10, 65]. Gates 2 and 3, the *regression gates*, set two standards for a drop: a task already unstable may lose one run of its 3 ($\tau = 0.34$), but a task in P may lose none. Because at $k = 3$ one unlucky run on a reliable task looks exactly like a broken task, a KILL that rests *only* on gates 2 and 3 is not final: the tasks behind it are swept again (RECHECK), and the KILL stands only if one of them regresses again on the fresh runs; otherwise the candidate is kept. Figure 7 shows a candidate killed this way, although it won more runs than it lost.

A sweep is not scored at all (RERUN) when any of these holds: it is incomplete; more than 10% of its rollouts are invalid; some task lacks k valid runs in either arm, which a single invalid rollout already causes, since a sweep plans exactly k ; exposure is unknown in a valid rollout; the Claude Code version changed mid-sweep; or more than 10% of its rollouts ran on an overloaded machine, with more than 3 times as many runnable processes as CPUs. Written out, with Ω the sweep’s rollouts, Ω_{inv} the invalid ones, $\mathcal{T}$ the tasks it planned, n_{unk} the number of valid rollouts whose exposure is unknown, $\text{cli}(\omega)$ the Claude Code version that ran rollout ω , $L(\omega)$ the mean number of runnable processes sampled during it, n_{cpu} the number of CPUs of the machine and $\varepsilon = 10\%$, a sweep is scored only when

$$\begin{aligned} \text{scorable} \iff \text{complete} \wedge \frac{|\Omega_{\text{inv}}|}{|\Omega|} \leq \varepsilon \wedge \mathcal{M} = \mathcal{T} \neq \emptyset \wedge n_{\text{unk}} = 0 \\ \wedge \left| \{\text{cli}(\omega)\}_{\omega \in \Omega \setminus \Omega_{\text{inv}}} \right| \leq 1 \wedge \frac{|\{\omega \in \Omega : L(\omega) > 3 n_{\text{cpu}}\}|}{|\Omega|} \leq \varepsilon, \end{aligned} \quad (5)$$

where *complete* means that the sweep finished, was not cut short by its cap on rollouts, and recorded every rollout it planned; the last condition applies when rollouts run in parallel. A sweep of an addition in which the candidate reached at least one task is also a RERUN, not a KILL, when none of those tasks could show a gain, because each one either passed every base run, leaving no room to improve, or never passed in either arm, so that nothing there tells a useful candidate from a useless one (the scorer suggests repairing those tasks’ checks, or screening tasks the candidate can win). Such a sweep did not test the candidate, and burying it would record a verdict nobody earned. A candidate that reached no task at all is killed instead: it gained nothing and never fired (gates 1, 4 and 5). An agent that timed out is a valid failure, unless the machine was oversubscribed at the time (more runnable processes than CPUs, so that every rollout ran slower than it would alone); an agent that could not run (authentication, rate limits, a failed reset) is invalid, and invalid is not fail.

Gates, recheck and RERUN together decide a confirm sweep of an addition. Let $\mathcal{T}$ be the exposed tasks that could show a gain: those the base did not pass in every run, and that passed at least once in either arm. Let $\mathcal{F}$ be the gates of Table 3, 1 to 5, that the sweep fails, and $\rho(u, v)$ the test that gates 2 and 3 apply to a task whose pass rate went from u under *base* to v under *cand*. Let $\mathcal{Q}$ be the tasks behind a failure of gate 2 or gate 3, and p'_b, p'_c and $\mathcal{R}'$ the pass rates and the exposed tasks of the recheck, which sweeps both arms again on the tasks of $\mathcal{Q}$ alone; a recheck that is not scorable is itself a RERUN. Reading the cases from the top,

$$\begin{aligned} V_{\text{add}} = \begin{cases} \text{RERUN} & \text{if } \neg \text{scorable} \vee (S \neq \emptyset \wedge \mathcal{T} = \emptyset), \\ \text{KEEP} & \text{if } \mathcal{F} = \emptyset, \\ \text{RECHECK} & \text{if } \emptyset \neq \mathcal{F} \subseteq \{2, 3\}, \\ \text{KILL} & \text{otherwise,} \end{cases} \quad \begin{aligned} \mathcal{T} &= \{t \in S : p_b(t) < 1 \wedge \max(p_b(t), p_c(t)) > 0\}, \\ \mathcal{F} &= \{g \in \{1, \dots, 5\} : \text{gate } g \text{ fails}\}, \\ \rho(u, v) &= (u - v > \tau) \vee (u = 1 \wedge v < 1), \end{aligned} \\ V_{\text{recheck}} = \begin{cases} \text{KILL} & \text{if } \exists t \in \mathcal{Q} \cap \mathcal{R}' : \rho(p'_b(t), p'_c(t)), \\ \text{KEEP} & \text{otherwise,} \end{cases} \quad \mathcal{Q} = \{t \in S : \rho(p_b(t), p_c(t))\}. \end{aligned} \quad (6)$$

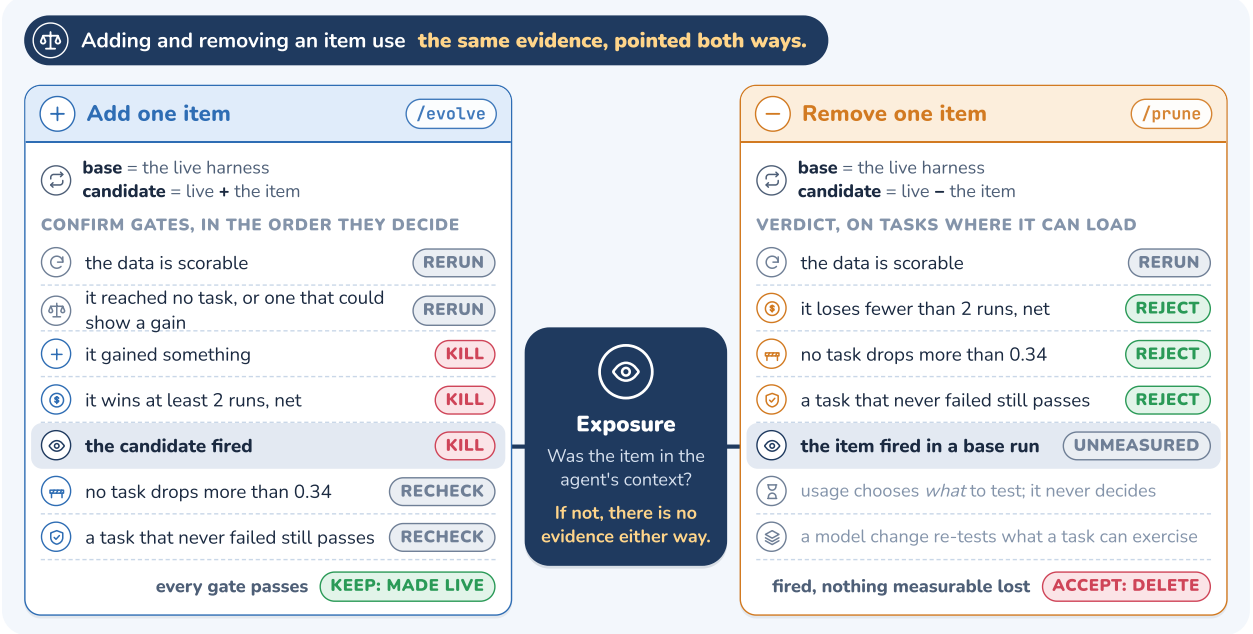


Figure 8: Adding and removing are one test, pointed both ways. /evolve compares the live harness with the harness plus a candidate; /prune compares it with the harness minus a live item. Both count only the tasks on which the item was in the agent's context: a candidate that never fired is never kept, and a live item that never fired is never deleted.

3.6 Removing and moving items

Deletion needs the same evidence as addition, pointed the other way (Figure 8). /prune removes one live item and runs a sweep with the item as *base* and without it as *cand*, on the tasks where the item can load. In the notation of Equations (2), (3) and (5), the verdict is

$$V_{\text{remove}} = \begin{cases} \text{RERUN} & \text{if } \neg \text{scorable}, \\ \text{REJECT} & \text{if } kN \leq -m \vee D > \tau \vee \exists t \in P : p_c(t) < 1, \\ \text{UNMEASURED} & \text{if } f = 0, \\ \text{ACCEPT} & \text{otherwise.} \end{cases} \quad (7)$$

REJECT keeps the item, because removing it cost runs, made a task collapse, or made a task in P lose a run. UNMEASURED keeps it too: here f counts the valid *base* rollouts, and $f = 0$ means the item fired in none of them. Only ACCEPT deletes it, to the graveyard. The cases are read from the top, so an item whose removal cost something gets REJECT even when it never fired (Section 5.5). UNMEASURED is the point: if nothing in the suite used an item, removing it changed nothing *because nothing needed it*, which is absence of evidence, not evidence of uselessness: a skill for a twice-a-year situation looks exactly like dead weight in between. This is the direct counterpoint to deletion rules that retire items whose measured effect is small [54]. For the same reason, usage data chooses *which* items to test and never decides a deletion.

A pass starts as a plan: the items to test, the tasks to test each one on, and the rollouts, time and cost this will take, estimated from the past rollouts in the same repository. Nothing runs until the developer approves it. Which items go into the plan depends on whether the model has changed. If the pinned model differs from the one the last pass ran on, *every* item is re-tested, because a stronger model may already do what an item teaches; always-on items come first, the largest first. Otherwise the pass tests only the few items most in doubt. Either way, an item is left out when no task can load it, since its removal could only come out UNMEASURED, or when it is an always-on skill the agent never opened in earlier sweeps, since no sweep could accept its removal: it could only keep the item, as REJECT or UNMEASURED. A pass can also move an item to a cheaper tier: an always-on skill whose use is concentrated in one folder can be rewritten as a skill or a rule tied to that folder. The swap is swept like a removal, the harness with the new item in place of the old against the live one, and decided by Equation (7). For a swap, a task is exposed when the old

item was in context in a *base* rollout or the new one in a *cand* rollout, and f counts the *base* rollouts in which the old item fired. The swap goes ahead only if it loses nothing measurable on those tasks and the old item fired, and the new item is made live only if it fired too.

3.7 The optional judge, Jev

Cortex can consult an outside model, Jev (TypeSafe’s System One), to help it choose what to try. Jev is sent some text and a typed question, and it returns a probability for a yes-or-no question, one option from a list with its confidence, or a score on a fixed scale. It writes no text and takes no action. It is off by default, and it was off in every run of this paper, so every theme, tier, candidate and prune order reported in Section 5 was chosen without it.

When it is on, Jev acts only in the *proposal* layer, the steps that decide which theme, candidate or order to try next: it can sort the logged lessons into themes, suggest a candidate’s tier, order a prune plan and tell the developer when a session is worth harvesting. One command, `cortex scope`, puts two questions to it about a candidate before a sweep is paid for. The first gives the candidate’s *relevance*: for each task the candidate would load on, Jev is asked whether the task is actually *about* the candidate’s subject, from the task’s title, request and one-line note, never the code or the fix, and the share of yes answers is the relevance. When it is below 35%, Cortex prints a warning, but it never blocks a sweep. Without Jev, Cortex reports only *breadth*, the share of all tasks the candidate would load on. The second, for a path-gated skill, is whether an agent doing the task, seeing only the skill’s description, would choose to open it. The tasks a candidate would load on are predicted from its paths; once the candidate has been swept, they are the tasks where its sweeps recorded it in context.

What the judge may not do. Jev may change which theme, candidate or order Cortex tries; it may never decide what a measurement found. The line is mechanical rather than a promise. The scorer, the sweep runner and `preflight` contain no reference to Jev, and a test in the shipped suite asserts that they do not. No Jev call runs inside a rollout sandbox, where it would become part of the harness under test. A wrong suggestion can waste rollouts, but it cannot turn a KILL into a KEEP, because the gates of Table 3 judge every candidate either way.

How the judge was used in this evaluation. Only the two `cortex scope` questions were exercised: after the runs, they were put to each run’s candidates, at no cost in rollouts (Section 5.7). Each candidate was rebuilt, from the graveyard or the live harness, in a scratch copy of its run’s repository that kept the run’s sweep records. Because every candidate had such a record, `cortex scope` took the tasks it would load on from the tasks where its own sweeps had recorded it in context; on those tasks, from the candidate’s text and each task’s title, request and note, never from its verdict, it estimated the candidate’s relevance and, for a path-gated skill, how often the agent would open it.

3.8 Implementation

Cortex’s scripts are 6,906 lines of shell and Python in 9 files (of which 1,243 lines are the optional judge client), plus 1,341 lines of prompts that define `/harvest`, `/evolve` and `/prune` as Claude Code commands. The prompts do what needs judgement (writing a task, proposing a candidate, choosing a tier), and every quantity a verdict depends on is computed by the scripts. The scripts refuse to make a candidate live without a KEEP from a confirm sweep or its recheck (or an ACCEPT for a swap); burying a candidate is done by the `/evolve` agent, the model session that runs the command. Sweeps run in parallel, one sandbox clone per worker, as many workers as the machine’s free memory and CPUs allow. Rollouts are *headless* Claude Code sessions, started by a script with no one at the keyboard, with a 10-minute limit. Cortex deletes the event stream it reads from each once exposure is known, because it holds everything the agent saw; Claude Code keeps its own record of every session, outside the repository.

4 Evaluation setup

The evaluation measures Cortex in a synthetic repository whose house rules a model often breaks. This section describes the repository (Figure 9), what one run does (Figure 10), how a run’s result is measured (Figure 11), two tests with known answers, two tests of how far the results carry (Figure 12), and the statistics.

The lab. *shopkit* is a small Python toolkit for orders and invoices (a catalogue, carts, invoices, refunds, exports, reports and a command line) with no dependencies and its own unit tests. Like many real codebases, it has *house rules*: conventions this one project has chosen, which code can break while still working. They are written in its `CONTRIBUTING.md`, and its continuous-integration check, which runs the unit tests and a linter, enforces part of them. Take the rule on time: code in `shop/` must read the current time only through `shop.clock`, never with `datetime.now()` or `date.today()`, so that tests can freeze the clock. Asked to give invoices a due date counted from today, an agent that computes it from `date.today()` produces code that works, yet the linter rejects it, and the reviewer replies: “Works, but CI is still red: `make lint` rejects `datetime.now() / date.today()` in `shop/`. We only read the time through `shop.clock` ...” (CI here is continuous integration). A model cannot learn such a rule from its training, only from the project. It is exactly what a harness exists to teach, and a model that keeps breaking it keeps being corrected, which is the evidence Cortex learns from. Four house rules each have a family of tasks built around them (Figure 9), named by a letter: A, the changelog (every change users can see gets a line in `CHANGELOG.md`); B, money (amounts are integer cents, scaled only with the project’s rate helpers); C, exporters (a new exporter needs a registry entry, a row in the documentation and a golden file, a stored sample of its expected output); and E, time (the clock rule above). A fifth family, D, holds plain bugs with no house rule, so the loop has nothing to learn there: it is the control, and it can only show harm.

Training and held-out scenarios. A *scenario* is one piece of work: a teammate’s commit that adds a failing test, and a request saying what is wanted; the due date above is one. The scenarios are split in two, the way a course keeps its exam apart from its exercises. The 26 *training* scenarios are the exercises: the runs play them as working sessions, the reviewer corrects the agent’s mistakes on them, and Cortex learns from those corrections. The 30 *held-out* scenarios are the exam: no session ever plays them, so Cortex never sees them, and they serve only to measure harnesses (the benchmark, below). The split is what lets a gain be read as learning. On a training scenario, a harness could help by recalling that one task, for instance that the due date is computed from `shop.clock`; on a held-out scenario, no task it recalls holds the answer, and only what carries over to new work, such as the rule, can help. Held-out scenario HE1 (held-out scenarios are named H, the family’s letter and a number), for one, asks for the orders of the last *N* days, counted back from today by default: a task no session plays, under the same rule on time. A gain on the held-out scenarios therefore shows that what Cortex learned carries over to work it has never seen, which is what a developer needs from a harness. Passing the exam also takes following the rule, not only making the test pass. Every held-out scenario of the four rule families comes with a rule-breaking fix, one that makes the scenario’s own test pass but breaks the rule: for HE1, it falls back on `date.today()`; in families A and C, it is the correct fix without its changelog line, or without its documentation row and golden file. Only the rule’s part of the check can tell it apart from the correct fix.

A run. A run is one complete, unattended replay of the project’s working life (Figure 10). It starts from a fresh copy of the repository and plays the 26 training scenarios as 26 working sessions, one per scenario, in 10 rounds; each family is introduced in a round of its own and returns in later ones. No one sits at the keyboard: a script types what a developer would type, and nothing in the repository is edited by hand.

One session. A session begins the way work reaches a developer: a teammate’s commit adds a failing test, for a bug or for a change in what the code should do, and a headless Claude Code session, one started by the script with no one watching, is asked to deal with it. When the agent stops, a scripted reviewer plays continuous integration. It checks, in order, that the teammate’s test is unchanged, that this test now passes, that the whole suite passes, that the linter is clean, and, for family A, that the changelog gained a line. The first check that fails gets the reply written for that failure, in the same words in every run, like the reply on time above, and the agent tries again, at most 3 times. The session then ends with `/harvest` (Section 3.1): if the session went from broken to working it saves a task, whose check also tests what the reviewer corrected, and it writes each correction down as a lesson. Because the replies are scripted, every run receives the same correction for the same mistake.

The loop, after each round. After the round’s sessions, the script runs `/evolve` (Figure 3). A cycle looks for a theme, writes one candidate, screens and confirms it, and keeps or buries it, or ends as `BARREN`, at no cost in rollouts, when no theme clears the bar. A sweep takes minutes and runs in the background, so the session that started it ends; when the sweep has finished, the script types `/evolve` again in a fresh session, which reads the result and goes on. It stops once a cycle has ended with no candidate pending, or when `/evolve` stops on its own; most rounds hold one cycle. The script runs `/evolve` in every round, even after Cortex has signalled that the loop should stop, so that every

The lab: shopkit, a small codebase with house rules

A house rule is a convention of this one project: code can break it and still work, and a model can learn it only from the repository.



Figure 9: The lab. *Top:* the repository, and its five task families: each rule family is built around one house rule written in `CONTRIBUTING.md`, with the mistake an agent typically makes, what catches it, and the counts of training and held-out scenarios; family D, plain bugs with no rule, is the control. *Bottom left:* one correction from start to finish, for the rule on time. *Bottom right:* the training scenarios are played as sessions; the held-out scenarios are kept for the benchmark.

run has the same shape. What `/evolve` changed, a kept item and the journal, is then committed, as each session's code and task were at the session's end, and the next round begins.

The runs. The 4 evaluation runs, R1 to R4, follow this procedure with Cortex in full; repeating it shows whether a result holds beyond a single run. A control run, C1, plays the same sessions, receives the same corrections and runs `/harvest` as usual, but never runs `/evolve`, so no skill or rule is ever added. Corrections can still fall in C1, because the agent can copy a rule from the fixes already in the code, and C1 measures how much they fall without `/evolve`. A development run, D0, used an earlier version of Cortex than `v1.0-eval`. It is reported apart and named wherever it is used (Sections 5.1, 5.7 and 7); every other number comes from the runs made at `v1.0-eval`.

One model in every role. Every evaluation and control run used Claude Haiku 4.5 (`claude-haiku-4-5-20251001`) through Claude Code 2.1.278, with Cortex's optional judge, Jev, off. The same model plays every role: it is the agent that works in the sessions; the model that runs `/harvest` and `/evolve`, and so writes the tasks and proposes the candidates; and the agent in every sweep and benchmark rollout. No stronger model teaches it. The runs depart from Cortex's defaults in the settings below, and in where the sandboxes sit on disk and how many run at once; none of them decides a verdict. The rollout model is pinned to the one the sessions use, so that every rollout uses that model (by default a rollout uses whatever model Claude Code is set to). The cap on rollouts per sweep, a guard on cost, is raised from 60 to 200, because a confirm sweep over every training task takes 156 rollouts (each of the 26 tasks 3 times under each of the two harnesses). The stop limit, the number of cycles in a row without a `KEEP` after which Cortex suggests

stopping, is raised from 2 to 3, because the lab introduces its families on a schedule, and quiet cycles are expected before the next family arrives. And Claude Code never updates itself mid-run.

The benchmark. After the last round, each run’s final harness sits an exam (Figure 11). Each held-out scenario is committed onto the run’s final code, so that it meets the codebase as that run left it, and the commit adds only the teammate’s test: the agent gets that commit and the scenario’s request, as in a session. Each scenario also has a *reference fix*, a correct solution written with it, and, in the rule families, its rule-breaking fix; both stay outside the repository and serve only to validate the scenario, below. The training scenarios are replayed from the commits their sessions started from. An *arm* is one harness: every arm faces the same scenarios, with the same model and the same oracle, and the arms differ only in the harness installed. The two primary arms are *none*, the minimal CLAUDE.md that Cortex’s installer, `cortex init`, writes, and nothing else, and *evolved*, what the run kept; for short, the text also calls *none* “no harness”, since it has no skills or rules. Each of the others answers one question (Figure 11, bottom). *none2*, the A/A arm, is *none* measured a second time, in a later batch, to show how far two measurements of one harness can differ. The rest are the *ablations*: *kitchen* is *none* with the whole contribution guide appended to CLAUDE.md; *flat* is *evolved*’s texts with every `paths` field removed; *desc-only* is *evolved* with each skill’s body replaced by neutral filler; *accept-all* is *evolved* plus every candidate the run buried; and *ideal* is written by hand: always-on skills for A and E, a rule on `shop/billing/**` for B, and a gated skill on `shop/plugins/**` for C. Every attempt, or rollout, runs in a clean sandbox, in Claude Code’s `acceptEdits` mode, in which file edits are accepted without asking and a command runs only if the settings already allow it, as the repository’s own `allow` list does for `make`, `python3` and the read-only `git` commands (`git status`, `git diff`, `git log`, `git show`; Section 7 says what a rollout could read through them). Success is decided by the lab’s own *oracle*, a fixed script that judges every attempt the way the scripted reviewer does, not by the checks Cortex harvested, so that Cortex never grades its own work. It makes the reviewer’s checks in the same order and stops at the first that fails: the teammate’s test is left unchanged, that test passes, the rest of the suite passes, the linter is clean (it checks the rules of B, C and E), and, for family A, the changelog gained a line. The agent does not behave the same way twice, so each scenario is attempted several times: *none* and *evolved* run each held-out scenario 5 times and each training scenario 3 times, and the other arms run the held-out scenarios 3 times. The primary arms run interleaved, taking turns, so that a slowdown lands on both; the other arms ran afterwards, in batches of their own, so every comparison with them crosses a batch boundary. Every arm alike also carried one shared skill listing on every turn: the descriptions of Claude Code’s own bundled skills, of Cortex’s three commands, which `cortex init` installs in the repository, and of the skills in the user-level Claude Code configuration; the agent invoked one of the bundled skills in 5.4% of *none* rollouts and 5.8% of *evolved* ones. Because every run’s code ends differently, a held-out scenario counts in a run only if it validates on that run’s final code: its test must fail there while the linter stays clean, its reference fix must pass the oracle, and, in a rule family, its rule-breaking fix must pass its own test and be caught only by the rule’s part of the check. One did not validate (HA6, in R4), and it is excluded from that run in every arm. The primary analysis uses the complete-case set, the 23 rule-family scenarios valid in every run; the set of all valid scenarios is secondary.

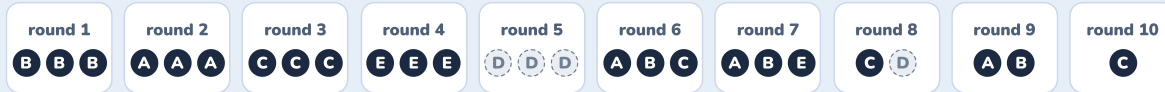
Tests with known answers, and of generality. Four more experiments complete the evaluation (Figure 12); all ran on Claude Haiku 4.5 except where the model change says otherwise. The first two measure Cortex’s own decisions against answers known before they ran; the last two ask whether its results carry beyond the lab and its model.

- **The gate test** asks whether the gates keep only what helps. It runs 29 candidates whose right verdict is known through the real pipeline, on a copy of C1’s final repository, which holds harvested tasks and no live item: 20 placebos, plausible advice that shares no keyword with a house rule, all written as always-on skills (10 about a coding topic, such as preferring `pathlib` to `os.path`, and 10 about a moment in the work, such as what to do after finishing an edit), which should be killed; 5 harmful items, advice that contradicts a house rule, 3 of them rules and the rest always-on skills, which should be killed; and the 4 items of *ideal*, which should be kept. Each is screened at $k = 2$ on tasks the live harness had failed (the same tasks for every placebo), and confirmed if it survives. The test’s procedure stops at the screen whenever the gain is not positive or the candidate never fired, and records that stop as a KILL; `score.sh`’s own verdict is recorded beside it, and the two can differ, since `score.sh` returns RERUN when no task on the screen could show a gain (Section 5.3).
- **The prune test** asks whether `/prune` deletes only dead weight. It runs `/prune` on a copy of R1’s final repository, with 4 items planted beside R1’s 4 and its limit on items per pass raised to cover them all. In its first pass, `/prune` chose which items to test and planned R1’s own, which it should keep. A second pass was given the planted

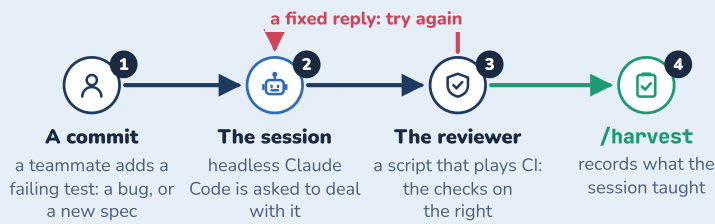
One run, end to end

No one at the keyboard and nothing edited by hand: a script types what a developer would type, and Claude Haiku 4.5 does all the work.

1 A RUN: 26 SESSIONS, ONE PER TRAINING SCENARIO, IN 10 ROUNDS, FROM A FRESH COPY OF SHOPKIT



2 EACH SESSION: WORK ARRIVES THE WAY IT REACHES A DEVELOPER



/harvest saves

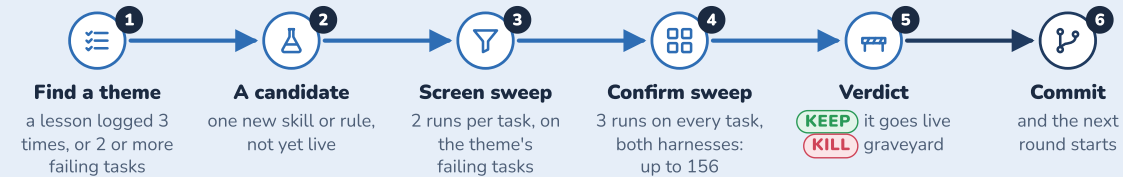
a task: the request, the broken commit, the fix, and a check that also tests the rule
lessons: one line per correction

The reviewer's checks, in order

- 1 the teammate's test is unchanged
- 2 that test now passes
- 3 the whole test suite passes
- 4 the linter is clean (rules B, C, E)
- 5 family A: a new changelog line

The first check that fails gets its own reply, in the same words in every run; the agent tries again, at most 3 times. Then /harvest runs.

3 THEN, AFTER THE ROUND'S SESSIONS: /evolve, UNTIL A CYCLE ENDS



No theme: the cycle is BARREN and runs no rollouts. **A sweep** runs for minutes in the background, so after each one /evolve is typed again in a fresh session, until a cycle ends; most rounds hold one. It runs in **every round**, even after Cortex suggests stopping.

4 THE EVALUATION AND CONTROL RUNS SHARE ONE SETUP; THE DEVELOPMENT RUN IS REPORTED APART

Evaluation runs **R1-R4**

Cortex on, as above. Repeating it in 4 runs shows whether the result holds beyond one lucky run.

Control run **C1**

the same sessions, with /evolve off: how much the agent improves from its own earlier fixes alone

Development run **D0**

an earlier version of Cortex than the evaluation tag: reported apart, and named wherever it is used

Claude Haiku 4.5 plays every role

- the agent that works in the sessions
- the model that runs /harvest and /evolve, so it writes the tasks and the candidates
- the agent in every sweep and benchmark rollout

Claude Code 2.1.278; the optional judge, Jev, off

Settings changed from Cortex's defaults

rollout model	pinned to the one the sessions use
sweep size cap	60 raised to 200: a full confirm sweep takes 156
stop suggestion	after 3 cycles in a row without a KEEP, not 2: the families arrive on a schedule
self-update	off: Claude Code cannot change mid-run
parallel workers	auto-sized, with more of the machine's memory

Figure 10: One run, end to end. (1) The rounds, and the family of each session. (2) One session: the teammate's commit, the agent, the scripted reviewer, whose fixed reply sends the agent back at most 3 times, and /harvest. (3) The /evolve cycle after each round's sessions (Figure 3). (4) The runs: evaluation and control share one setup; the development run is reported apart.

After the runs: the benchmark

The exam uses scenarios that no session ever played, and an oracle that Cortex never wrote.

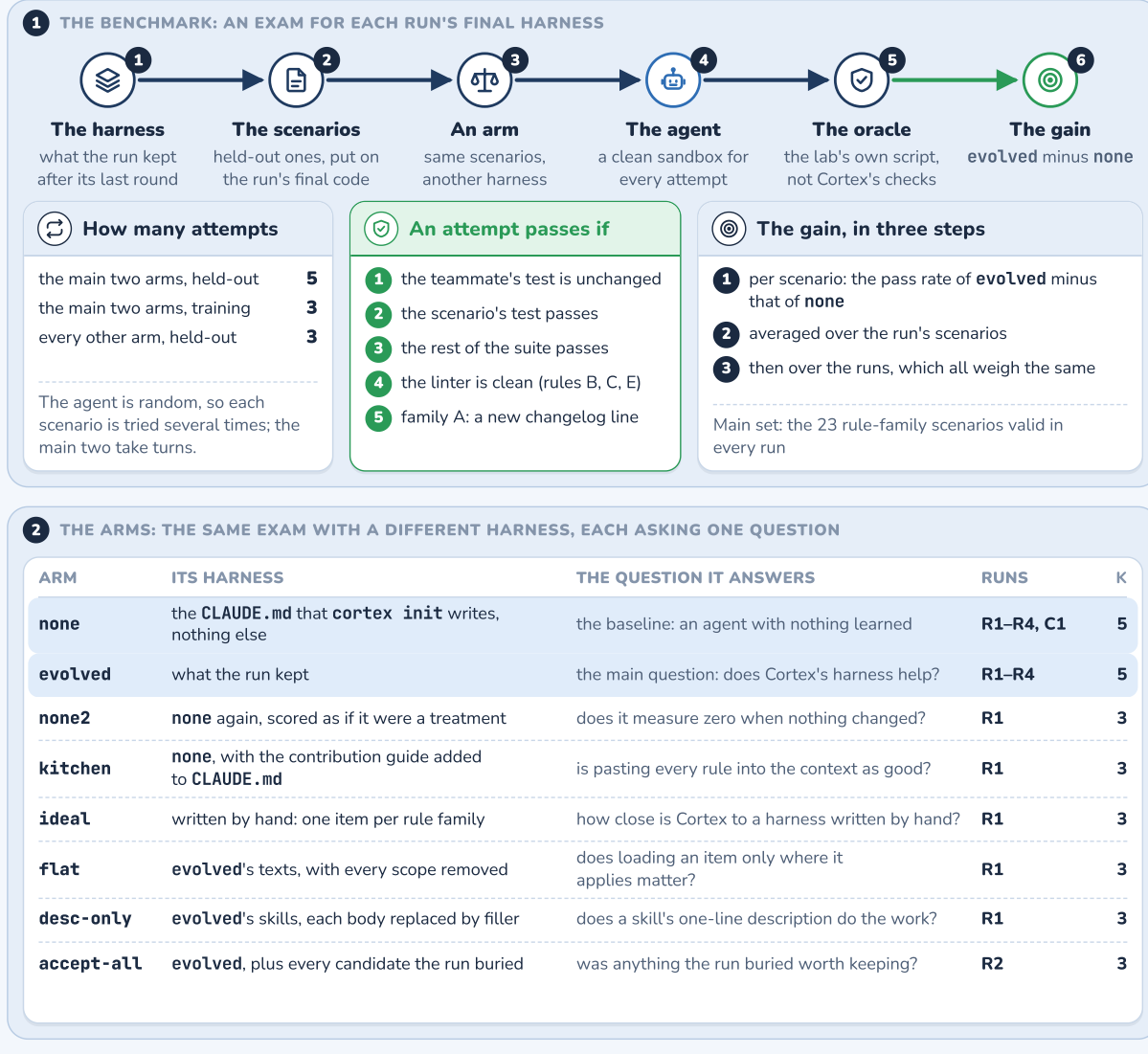


Figure 11: The benchmark. *Top:* how each run's final harness is examined, from the scenarios to the gain, which is the difference between the pass rates of **evolved** and **none**. *Bottom:* the arms, which share the scenarios, the model and the oracle and differ only in the harness, with the question each one answers, the runs it was measured in, and its number k of attempts per held-out scenario. **none** and **evolved** also run the training scenarios.

items it had left out: 2 placebo always-on skills, which it should delete; a gated skill that no task reaches, which it should skip; and an always-on skill that never fires, which it should report as unmeasured rather than delete.

- **The model change** asks whether the harness still matters under a stronger model. It benchmarks R1's harness with Claude Sonnet 5 (`claude-sonnet-5`) as the agent, at $k = 3$, and runs `/prune` on another copy of R1 with its rollouts pinned to that model, while the session that runs `/prune` is still Claude Haiku 4.5. The two models' results are analysed separately and never pooled.
- **The second repository** asks whether the loop works on a project nobody in this study built. `structlog` was chosen as a Python project with a permissive licence, a long history, a fast test suite that needs no network or credentials and a convention newcomers break, absent from SWE-bench [22], SWE-Gym [38], SWE-smith [60] and Aider's

coding benchmark. The repository is hynek/structlog, taken at commit 73393f34b4. It gets one loop on tasks mined from its own history, then none against evolved on its 4 newest tasks at $k = 5$.

Four more experiments: two with known answers, two of how far the results carry

The first two check Cortex's decisions against answers known before they ran; the last two ask whether the results hold elsewhere.

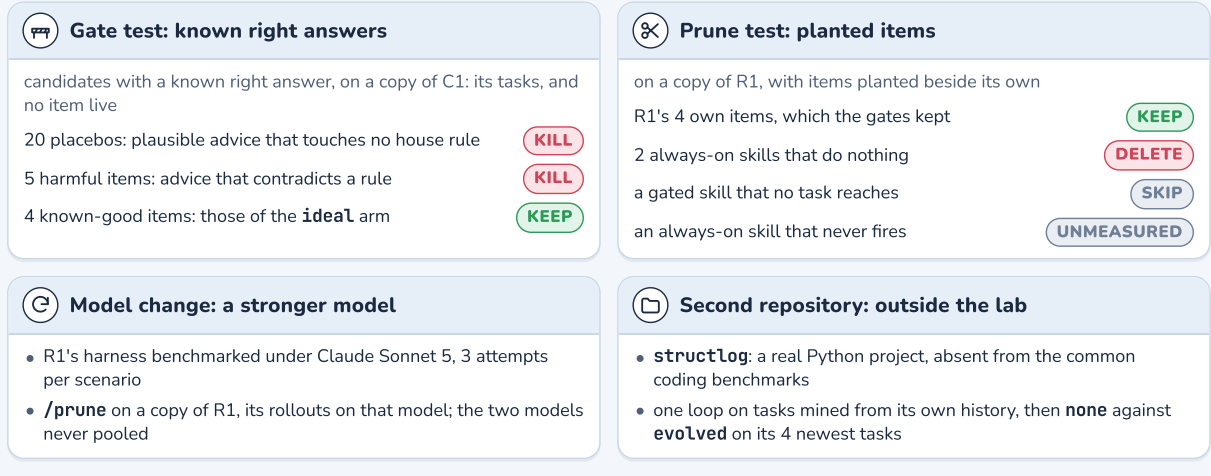


Figure 12: The four further experiments. The gate test and the prune test run the real pipeline on copies of finished runs, and each item's right verdict is known before it runs; the model change and the second repository ask how far the results carry. The badges give each item's right verdict, not the verdict the pipeline reached (Sections 5.3 and 5.5).

Statistics. The primary comparison is *evolved* against *none* on the held-out scenarios of the four rule families, paired by scenario within run. For each run and each scenario we take the difference between the two arms' pass rates, average it over the run's scenarios and then over the runs, so that a run that measured more scenarios does not weigh more; this is the gain reported in Section 5.1. With $\mathcal{R}$ the runs, $\mathcal{V}_r$ the scenarios of the analysed set that are valid in run r , and $y_{rsi}^a \in \{0, 1\}$ the outcome of the i -th of the n_{rs}^a valid rollouts of scenario s under arm a (*evo* for *evolved*) in run r , the gain is

$$\hat{\Delta} = \frac{1}{|\mathcal{R}|} \sum_{r \in \mathcal{R}} \frac{1}{|\mathcal{V}_r|} \sum_{s \in \mathcal{V}_r} d_{rs}, \quad d_{rs} = \bar{y}_{rs}^{\text{evo}} - \bar{y}_{rs}^{\text{none}}, \quad \bar{y}_{rs}^a = \frac{1}{n_{rs}^a} \sum_{i=1}^{n_{rs}^a} y_{rsi}^a, \quad (8)$$

and every other comparison between two arms is computed the same way. The study states its questions as hypotheses, each with a margin that decides it (Section 5); the primary one, H1, is about this gain. Two calculations then say how far that number can be trusted.

- **The confidence interval decides the hypothesis.** A two-way bootstrap builds 10,000 alternative versions of the study, each drawing the runs and the scenarios at random with replacement, independently of each other, and recomputes the gain on each. Drawing with replacement means that a run or a scenario can be drawn twice and another left out, as if the study had happened to sample differently; the spread of the recomputed gains shows how much the result depends on that luck. The middle 95% of those values, with the most extreme cut off equally at both ends, is the 95% CI; its two ends are the pair of numbers printed in brackets throughout the paper. In replicate b , the bootstrap draws $|\mathcal{R}|$ runs r_{bi}^* from $\mathcal{R}$ and $|\mathcal{V}|$ scenarios s_{bj}^* from $\mathcal{V} = \bigcup_r \mathcal{V}_r$, each uniformly; J_{bi} holds the draws j whose scenario is valid in run r_{bi}^* , and I_b the draws i for which J_{bi} is not empty. Then

$$\hat{\Delta}_b^* = \frac{1}{|I_b|} \sum_{i \in I_b} \frac{1}{|J_{bi}|} \sum_{j \in J_{bi}} d_{r_{bi}^* s_{bj}^*}, \quad r_{bi}^* \stackrel{\text{iid}}{\sim} \mathcal{U}(\mathcal{R}), \quad s_{bj}^* \stackrel{\text{iid}}{\sim} \mathcal{U}(\mathcal{V}), \quad \text{CI} = [Q_{\alpha/2}, Q_{1-\alpha/2}](\{\hat{\Delta}_b^*\}_{b=1}^B), \quad (9)$$

with $B = 10,000$, Q_q the q -quantile of the B replicates, and $1 - \alpha$ the 95% level. On the complete-case set every scenario is valid in every run, so each J_{bi} holds every draw. The primary hypothesis is supported when the lower end of this interval is above zero. Every other hypothesis has its own margin, and Section 5 gives each verdict with it.

- **The permutation test gives the p -value.** It asks how often a gain at least this large, in either direction, would appear if the harness made no difference. If it made none, the arm label on each rollout would carry no information, so within each scenario of each run the labels are shuffled at random, which keeps every scenario’s own difficulty fixed, and the gain is recomputed. This is repeated $B = 10,000$ times. Each shuffle π_b reorders, uniformly at random and independently, the $n_{rs}^{\text{evo}} + n_{rs}^{\text{none}}$ outcomes of each scenario s of each run r , and gives them back to the two arms in their original numbers; $\mathfrak{S}_n$ is the set of the $n!$ orders of n outcomes, and $\hat{\Delta}(\pi_b)$ is the gain recomputed after the shuffle. Then

$$p = \frac{1 + \sum_{b=1}^B \mathbb{1}[|\hat{\Delta}(\pi_b)| \geq |\hat{\Delta}|]}{B + 1}, \quad \pi_b \sim \mathcal{U}\left(\prod_{(r,s)} \mathfrak{S}_{n_{rs}^{\text{evo}} + n_{rs}^{\text{none}}}\right). \quad (10)$$

The 1 added above and below counts the observed labels as one of the possible arrangements, so p is never zero: it is at least $1/(B + 1)$.

Every comparison measured in a single run, the A/A arm, the ablations and the model change, has an interval that resamples scenarios only. When the gain is tested in each rule family on its own, Holm’s correction adjusts the four p -values for having made four tests, since the more tests are made, the likelier one of them is to look significant by chance. With the four sorted from smallest to largest, $p_{(1)} \leq \dots \leq p_{(4)}$, the adjusted values are

$$\tilde{p}_{(i)} = \min\left\{1, \max_{j \leq i} (5 - j) p_{(j)}\right\}, \quad (11)$$

so the smallest is multiplied by four, the next by three, and so on, and no adjusted value is smaller than the one before it. Proportions of candidates get Clopper–Pearson intervals, an exact interval suited to small counts, which for x successes in n trials is

$$\left[\mathcal{B}_{\alpha/2}^{-1}(x, n - x + 1), \mathcal{B}_{1-\alpha/2}^{-1}(x + 1, n - x) \right] \xrightarrow{x=0} \left[0, 1 - \left(\frac{\alpha}{2}\right)^{1/n} \right], \quad (12)$$

with $\mathcal{B}_q^{-1}(a, b)$ the q -quantile of the beta distribution with parameters a and b , the lower end taken as 0 when $x = 0$ and the upper end as 1 when $x = n$; when no candidate of a kind was kept, it reduces to the closed form on the right, which gives the bound on kept placebos in Section 5.3. Fisher’s exact test, which suits small counts too, asks whether the known-good candidates were kept more often than the placebos. The A/A arm carries its own rule: if its interval excludes zero, every other result is suspect until explained. Every random draw uses a fixed seed, so rebuilding the analysis gives the same intervals and p -values. Invalid rollouts are excluded and counted. Every verdict of the loop reported in Section 5 is the gates of Section 3.5 applied to the recorded rollouts, by a re-implementation of `score.sh` that agrees with it on all 51 sweeps of the evaluation runs.

5 Results

Each question of the evaluation is stated as a hypothesis with a *margin*: the result a measurement must reach for the hypothesis to count as supported. Table 4 lists the hypotheses tested, each with its margin, its result, its verdict and the subsection that reports it. Each is reported where its question arises, stated in words, with its verdict, including those that failed. A hypothesis is supported or not supported as its margin gives; it is inconclusive when the measurement could not decide the question, even where the letter of the margin is met, and the text then says why. Differences are in percentage points of pass rate, evolved minus none unless stated otherwise. Each interval is a 95% CI, the range of values the data leave plausible (Section 4).

5.1 Held-out gain

The baseline. Without a harness the agent passes the control family and often breaks the house rules. On the held-out scenarios, none’s pass rate was 100% on family D, but 0% on family A, 26% on B, 2% on C and 55% on E. H0, that the model without a harness often breaks the house rules but passes the control family, is supported. Its margin asks that family D pass at least 90% of the time and that at least three of the four rule families pass less than 50%: D passed 100%, and 3 rule families are below 50%; E is the one above it. H0 checks the lab rather than Cortex: its scenarios were built so that the model breaks the rules unaided (Section 7). The rules are written down, in the repository’s contribution guide (`CONTRIBUTING.md`), but the agent opened that file in only 0.5% of none’s held-out rollouts.

Table 4: The hypotheses tested. The margin is the result a measurement must reach for the hypothesis to count as supported, and the verdict is what the margin gives, or inconclusive when the measurement could not decide the question. none is the harness with no skills or rules, and evolved the one a run built; A, B, C and E are the four rule families, and D is the control. Differences are in percentage points of pass rate, evolved minus none unless the row says otherwise, and the two numbers in brackets are the lower and upper ends of a 95% CI. The last column is the subsection that reports each one.

	Hypothesis	Margin	Result	Verdict	Sec.
H0	A check of the lab, not of Cortex: with no skills or rules (none), the agent often breaks the house rules, so families A, B, C and E often fail, but it fixes the plain bugs of the control family, D.	On the held-out scenarios under none, D passes at least 90% of the time, and at least three of A, B, C and E pass less than 50%.	Pass rates under none: D 100%; A 0%, B 26%, C 2%, E 55%. D is above its bar, and 3 of the four rule families are below 50%; E is the one above it.	supported	5.1
H1	Primary. What Cortex learns carries over to new work. On the held-out scenarios of A, B, C and E together, which no session played, the agent passes more often with the harness a run evolved (evolved) than with none.	The gain, evolved minus none, has an interval whose lower end is above zero: even the lowest value the data leave plausible is a gain.	On the 23 rule-family scenarios valid in every run, the pass rate rose from 21.5% to 76.5%: a gain of +55.0 [+34.1, +74.1]. Permutation $p = 0.0001$, the smallest the test can return.	supported	5.1
H2	Cortex does no harm where there is nothing to learn. D has no house rule and already passes under none, so the evolved harness cannot help there; it should not make D worse, and no run should keep an item written for D.	The change in D’s pass rate has an interval whose lower end is above –10 points, so that a loss that large is not plausible; and no run keeps an item for D.	D changed by –0.8 [–5.0, 0.0]: the lower end is above –10. The upper end cannot be above zero, since none passes D 100% of the time and leaves nothing to gain. No run kept an item for D.	supported	5.1
H3	Cortex puts each kept item in a sensible loading tier, which decides when its text enters the agent’s context, and the item fires (a rule loads, a skill is opened) mainly in the work of its own family.	Every kept item meets three criteria: its form (rule or skill) fits how it loads; its scope is limited; it fires more often on its own family’s rollouts than on the others’.	15 of the 18 kept items meet all three. The rest are rules on shop/**, a scope wide enough to load on almost every task of every family, and they fail only the third criterion.	not supported	5.2
H4	The harness spares the developer corrections while it is being built. In the evaluation runs Cortex’s items go live as they are kept, so the scripted developer should correct the agent less often there than in the control run, which plays the same sessions with /evolve off.	Fewer corrections per session in the evaluation runs than in the control run, counting every session, including those before any item was kept.	60 corrections in 104 sessions (0.58 per session) in the evaluation runs, against 23 in 26 (0.88) in the control run. With 1 control run, a permutation test cannot return p below 0.2, so the difference is descriptive, not a tested effect.	supported	5.1
H5	The gates, the rules that decide whether a candidate is kept, give the right verdict on candidates whose right verdict is known: 20 placebos (plausible advice that touches no house rule), 5 harmful ones (advice that contradicts a house rule) and 4 known-good ones (the items of the hand-written harness ideal). They should rarely keep a placebo, kill every harmful candidate and keep the known-good ones.	The rate at which placebos are kept has an interval whose upper end is below 25%; every harmful candidate is killed; and at least 3 of the 4 known-good ones are kept.	Placebos: 0 of 20 kept; with so few, the rate could still be as high as 17%, below 25%: met. Harmful: none kept, but only 2 of 5 killed, with no verdict on the other 3: not met. Known-good: 2 of 4 kept: not met.	not supported	5.3
H6	The gates add value: what the loop buries would not have helped. accept-all is a run’s evolved harness plus every candidate that run buried, and it should score no better than evolved.	The difference, accept-all minus evolved, has an interval whose upper end is below +5 points, so that an advantage that large is not plausible.	In R2, accept-all minus evolved is –8.9 [–16.1, –1.7], so the margin is met. But the arm adds a single candidate, and no gate buried it: the scoring script reached no verdict, and the /evolve agent buried it as overlapping a live rule. The result rests on that burial, not on the gates.	inconclusive	5.3
H7	The scopes earn their place. Every kept item has a paths field, which loads it only when the agent works on matching files. flat is the same texts with every paths field removed; it should put more text into every turn’s context and score no better than evolved.	flat has more always-on characters (text present on every turn) than evolved; and the difference, flat minus evolved, has an interval whose upper end is below +5 points.	In R1, flat has 1,688 always-on characters against evolved’s 559, and flat minus evolved is –12.5 [–28.3, +1.7]: the upper end is below +5. Both parts are met.	supported	5.4

Table 4: The hypotheses tested, continued.

	Hypothesis	Margin	Result	Verdict	Sec.
H8	Two harnesses state all four house rules: kitchen, which appends the contribution guide to CLAUDE.md, and ideal, written by hand with the rules known. evolved should come close to them on the rule families, with less always-on text.	evolved minus the better of the two has an interval whose lower end is above -10 points, and evolved has fewer always-on characters than either.	In R1, pass rates: evolved 87.5%, kitchen 76.4%, ideal 48.6%. evolved leads both: kitchen, the better of the two, by 11.1 points [0.0, +22.5], an interval that reaches zero. Always-on characters, in that order: 559, 2,519 and 728.	supported	5.4
H9	What the loop learns is repeatable. The 4 evaluation runs are independent, each starting from a fresh copy of the repository; they should learn the same families, as items of similar form.	A run has learned a family when it keeps an item for it. Of the families learned in any run, at least half are learned in every run.	All four rule families were learned in every run: each of the 4 runs kept an item for A, B, C and E, and none for D. Every run but R3 chose the same scopes.	supported	5.2
H10	/prune, which re-measures live items by removal, deletes only dead weight. Beside R1’s own 4 items, which it should keep, the prune test plants 4 items: 2 placebo always-on skills, to delete; a gated skill that no task reaches, to skip; and an always-on skill that never fires, to report as unmeasured.	Every planted item gets the verdict expected of it, and no item that never fired is deleted.	/prune kept R1’s own 4 items. Of the 4 planted items, 2 got the expected verdict (6 of 8 overall): one placebo was deleted and the unreachable skill skipped. The other placebo was kept: 2 tasks lost runs without it, and the agent never opened it. The skill that never fires got no verdict. No item that never fired was deleted.	not supported	5.5
H11	A skill is offered to the agent by its description, one line in a listing; its body enters the context only if the agent opens it. The description should carry much of the skill’s effect by itself. desc-only is evolved with each skill’s body replaced by neutral filler.	desc-only recovers at least 50% of evolved’s gain over none, with the lower end of that share’s interval above zero.	It recovered 79% [58%, 100%], which meets the letter of the margin. But only 1 of R1’s 4 items is a skill and could be rewritten; the rest are rules and kept their words. In C, the rewritten skill’s family, it recovered 16%. One skill cannot settle the question.	inconclusive	5.4
H12	The development run, D0, used an earlier version of Cortex than v1.0-eval, the version evaluated. Its harness, measured on the same held-out scenarios with the evaluation’s tooling, should gain in line with the evaluation runs.	D0’s gain lies inside the interval of the evaluation runs’ gain, taken over every valid scenario.	D0’s harness gained +42.5 points, inside the evaluation runs’ interval, [+35.4, +75.0], and within the range of the single runs’ own gains, +32.5 to +66.7.	supported	5.1
H13	A harness evolved for one model may be partly redundant under a stronger one. /prune, run on R1’s harness with its rollouts on Claude Sonnet 5 instead of Claude Haiku 4.5, should find items the weaker model needed and the stronger one no longer does.	/prune finds at least one such item removable.	2 of the harness’s 4 items are removable under Claude Sonnet 5: the money rule and the exporter checklist. The money rule is the clear case, since the stronger model passed every held-out money scenario with no harness; the exporter checklist is less certain.	supported	5.5
H14	The loop also helps outside the lab. structlog is a real open-source project that nobody in this study built; the loop runs once on tasks mined from its history, and the harness it evolves should raise the pass rate on the project’s newest tasks.	A positive gain, evolved minus none, on the 4 held-out tasks.	Nothing was learned. In 7 of the 8 sessions the agent passed the project’s tests and linter at the first attempt, so nothing recurred and no item was kept. evolved was identical to none, which left nothing to compare.	inconclusive	5.6
H15	The optional judge rates a candidate’s <i>relevance</i> : the share of the tasks it would load on that it is actually about. Relevance should separate the candidates the gates keep from those they kill on a regression (a task made worse); <i>breadth</i> , the share of all tasks it would load on, which needs no judge, should not.	To separate two groups, every kept candidate must score above every killed one. Relevance must separate them, and breadth must not.	Relevance did not separate them: the lowest kept candidate scored 17% and the highest killed one 75%. Neither did breadth (8% against 58%). Only 2 candidates were killed on a regression, which makes a thin comparison.	not supported	5.7
H16	For a skill, the judge also predicts its invocation rate, how often the agent will open it. The prediction should track the rate measured in the rollouts.	With so few skills, only the order is asked for: the predictions rank the skills in the same order as the measured rates.	The order is preserved for the 4 skills tested, 2 of them kept by D0, which meets the letter of the margin. But 3 of the predictions are tied at the top, so the order rests on the remaining skill.	inconclusive	5.7

The primary result. H1, the study's primary hypothesis, is that evolved beats none on the held-out scenarios of the four rule families taken together; its margin is an interval whose lower end is above zero. On the 23 rule-family scenarios valid in every run, evolved beat none by +55.0 points (95% CI [+34.1, +74.1], permutation $p = 0.0001$), so H1 is supported (Table 5 and Figure 13). With 10,000 shuffles, that p is the smallest the test can return: no shuffle of the arm labels produced a gain as large as the one observed. Every rule family gained: A by +97.0 [+91.0, +100.0], B by +53.3 [+17.5, +88.3], C by +50.0 [+25.0, +73.3] and E by +26.7 [+1.7, +60.0]. The permutation test gives each family $p = 0.0004$ after Holm's correction, which raises each p -value to allow for having tested four families instead of one; as for the four together, no shuffle reached any family's gain. That test holds the measured scenarios and runs fixed, while the intervals, which are not adjusted for the four families, also allow for which ones were drawn, and the clock family's lower end lies close to zero. H2 is that the harness does not harm the control family (the lower end of its interval stays above -10 points) and that no run keeps an item for it. The control family moved by -0.8 [-5.0, 0.0], and no run kept an item for it, so H2 is supported. Counting every rule-family scenario in each run where it was valid, so also the one excluded from a single run (Section 4), the gain on the four rule families together is +56.5 [+35.4, +75.0]. Every rollout behind these numbers was valid.

Table 5: The held-out gain, family by family: evolved against none on the scenarios valid in every evaluation run, R1 to R4. Pass rates are over rollouts, and *Rollouts* counts both arms. The gain is the paired difference in pass rate, in points, averaged over the scenarios of each run and then over the runs; its 95% CI is the two-way bootstrap and p the permutation test (Section 4). Holm's correction adjusts the four rule families' p -values for having made four tests. In the rule-family rows no shuffle reached the observed gain, so each p is the smallest the test can return with 10,000 shuffles; the test keeps the measured scenarios and runs fixed, while the interval also allows for which ones were drawn. Family D is the control.

Family	Scenarios	Rollouts	Pass rate		Gain	95% CI	p	Holm p
			none	evolved				
A, changelog	5	200	0%	97%	+97.0	[+91.0, +100.0]	0.0001	0.0004
B, money	6	240	26%	79%	+53.3	[+17.5, +88.3]	0.0001	0.0004
C, exporters	6	240	2%	52%	+50.0	[+25.0, +73.3]	0.0001	0.0004
E, time	6	240	55%	82%	+26.7	[+1.7, +60.0]	0.0001	0.0004
A, B, C, E together	23	920	22%	77%	+55.0	[+34.1, +74.1]	0.0001	
D, plain bugs (control)	6	240	100%	99%	-0.8	[-5.0, 0.0]	1.000	

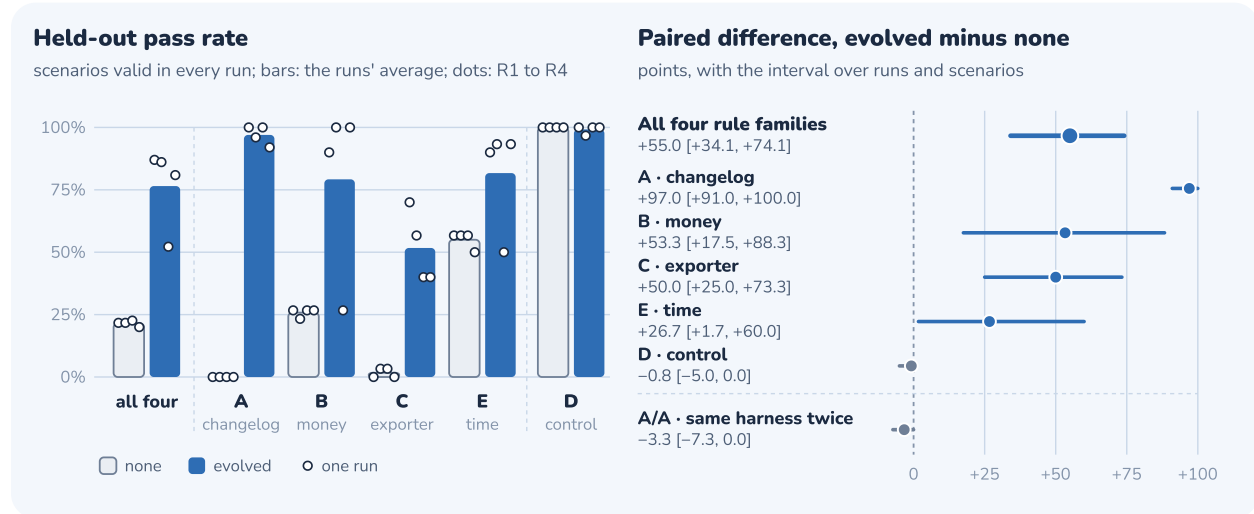


Figure 13: Held-out results on the scenarios valid in every run. Left: pass rate; each bar averages the 4 evaluation runs, and each dot is one run, R1 to R4 from left to right. Right: evolved minus none, with the two-way bootstrap interval over runs and scenarios. The bottom row is the A/A check, the none harness measured twice in R1 over all five families, whose interval resamples scenarios only. Family D is the control.

The noise floor. The A/A check measures how far two measurements of one and the same harness can differ. none2 is the none harness again, measured as if it were a different one; on all five families of R1, the two differed by -3.3 points $[-7.3, 0.0]$, with an interval that resamples scenarios (permutation $p = 0.13$). By the rule of Section 4, an A/A interval that excluded zero would put every other result in doubt. This interval ends at zero, so the rule does not apply, but only just: on each of the 3 scenarios where the two measurements differ, none passed more often than none2. none2 ran in a later batch than none, together with the ablation arms of Section 5.4, so the difference may hold a shift between batches (a change in conditions between the two periods in which the rollouts ran) as well as noise.

The runs. Every run gained on the four rule families taken together, by different amounts (the dots in Figure 13), though not on every family: R3 gained nothing on money and lost on the clock (Figure 17). R3 gained least: its pooled pass rate on the 23 scenarios valid in every run rose from 22.6% to 52.2%, where the other runs reached between 80.9% and 87.0%. Section 5.2 traces why. Every held-out scenario under every arm of R1 to R4 is shown in Figure 22, in Section A.

Training against held-out scenarios. The benchmark also replayed the training scenarios, the ones the sessions had played, under none and evolved (Section 4). Measured on the 22 of them in the four rule families, the gain would have been +78.8 points $[+68.9, +86.7]$ instead of +55.0. More of that difference lies in the baseline than in the harness: without a harness the agent passed 6.4% of the training rollouts in these families and 21.5% of the held-out ones, so the training scenarios left more to gain, while the evolved harness's own pass rate went from 85.2% to 76.5%. By family, the evolved rate went from training to held-out as follows: A from 90.3% to 97.0%, B from 88.9% to 79.2%, C from 76.4% to 51.7%, and E from 85.4% to 81.7%. It fell most for exporters, and fell there in every run. For money, the fall is R3's, the run that narrowed its money rule (Section 5.2); without it, the rate rose. For the clock, the rate held but the gain did not, because the held-out clock scenarios were easier without a harness: the agent alone passed 55.0% of their rollouts, against 10.4% of the training ones. The comparison is descriptive and tests no hypothesis: the two sets are different scenarios, attempted 3 times each against 5; a training scenario starts from the commit its session started from, not from the run's final code; and its session's fix is in the repository's history, which a rollout can read (Section 7).

Why rollouts fail. The oracle, the lab's own script that scores a rollout (Section 4), makes the reviewer's checks in order and stops at the first that fails, so each failed rollout has exactly one cause in Figure 14. First, the teammate's test, the test file the scenario starts from, must be unchanged: a rollout that edits it has moved the goalposts. Second, that test must pass: otherwise the task is not solved. Third, the rest of the test suite must still pass. Last, the house rule must hold: the linter, which checks the rules of families B, C and E, must be clean, and in family A the changelog must have gained a line. None of these rollouts failed on the rest of the suite. Without a harness, 94% of the rule families' failed rollouts (341 of 361) passed the tests and broke the house rule: the agent solved the task and missed the convention, the failure the lab was built around. The evolved harness turned most of them into passes. Of its 108 failures, 86 still broke the rule, 43 of them in the exporter family. A rollout that changes the teammate's test fails before anything else is checked; that happened in 9 none rollouts and 13 evolved ones, all in families C and E.

Corrections, and the development run. H4 is that the scripted developer has to correct the agent less often in the evaluation runs, where Cortex's items go live as they are kept, than in the control run, where /evolve is off; the count covers every session, including those before any item was kept. The evaluation runs' sessions needed 60 corrections in 104 sessions (0.58 per session), and the control run's 23 in 26 (0.88). The margin, fewer corrections per session than in the control run, is met, so H4 is supported. But with a single control run beside the 4 evaluation runs, a permutation test that shuffles which run is the control has too few arrangements to give p below 0.2, so we read the difference as description, not as a tested effect. H12 is that the result of the development run, D0 (Section 4), holds at $v1.0\text{-eval}$. Its harness, measured on the same held-out scenarios with the evaluation's tooling, gained +42.5 points, inside the evaluation runs' interval over every valid scenario, $[+35.4, +75.0]$, so H12 is supported. That interval is for the mean of the 4 runs, and a single run varies more: on the same scenarios the runs' own gains ranged from +32.5 to +66.7, and the development run's gain lies within that range as well.

Why held-out rollouts fail

every rollout of the scenarios valid in every run, R1 to R4; the oracle stops at the first check that fails: the teammate's test left unchanged, then passing, then the house rule

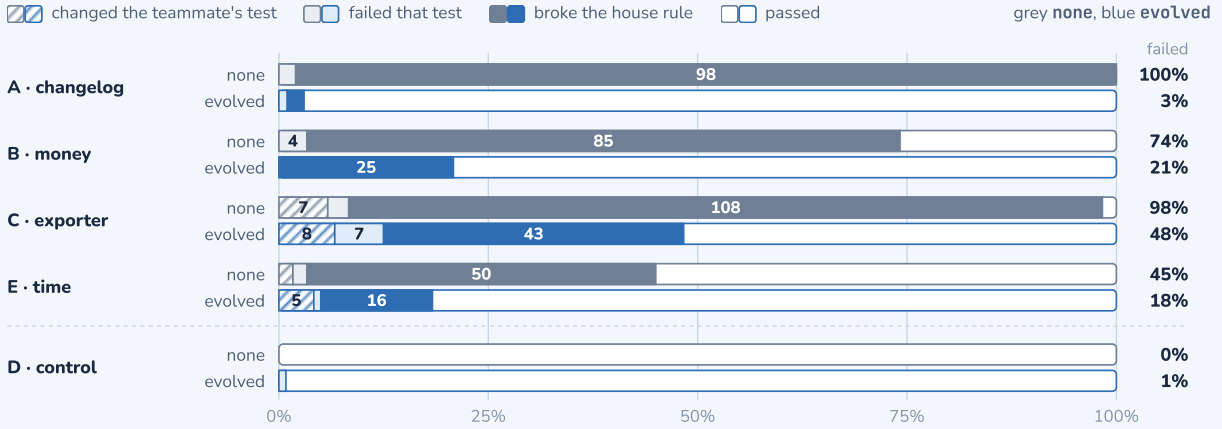


Figure 14: Why held-out rollouts fail, in the 4 evaluation runs, on the scenarios valid in every run. Each bar is all of an arm's rollouts on a family, split by the first check the oracle found failing: the agent edited the teammate's test (hatched), left it unchanged but did not make it pass (light), or passed the tests and broke the house rule (solid); white is a pass. The number inside a segment counts its rollouts, and the number at the right is the share that failed. Without a harness, nearly every rule-family failure passed its tests and broke the house rule.

Every /evolve cycle

all 4 evaluation runs: what was proposed, measured, kept

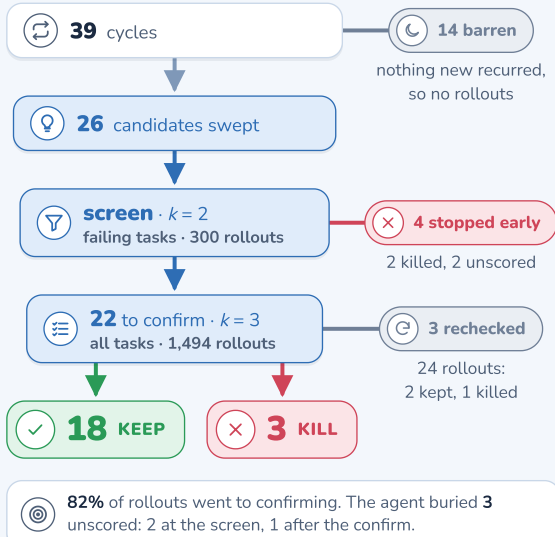


Figure 15: Every recorded /evolve cycle of the 4 evaluation runs, each candidate's verdict computed again from its rollouts by the rules of Section 3.5. A cycle in which nothing new recurred runs no rollouts; the screen, on the failing tasks ($k = 2$), stops weak candidates before the confirm sweep over all tasks ($k = 3$), where most rollouts go. The cycle drawn split in Figure 16 swept 2 candidates. score.sh returned RERUN for 3 candidates, which the /evolve agent buried without a verdict.

Every cycle, run by run

the 4 evaluation runs, each cycle in the order it ran

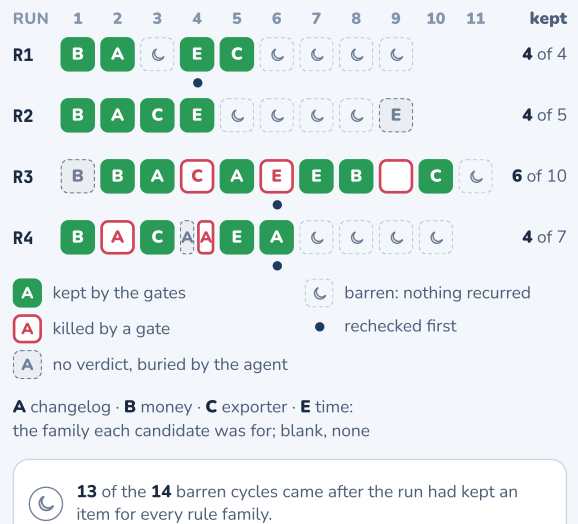


Figure 16: The same cycles, run by run; barren cycles, which sweep nothing, are placed where the journal recorded them. Each cell is one cycle, numbered in the order it ran: its colour is the verdict the gates give, and its letter the family the candidate was written for. Every run kept an item for each rule family, and most barren cycles came after that, when nothing new recurred. The cycle of R4 drawn split swept 2 candidates.

What each run kept, and whether it reached the held-out scenarios

the items each evaluation run kept, by house rule; the bar is the share of the family's held-out *evolved* rollouts in which the items fired (a rule loaded, a skill opened), and the number is the run's held-out gain there, in points



Figure 17: What each evaluation run kept, by house rule, and whether it reached the held-out scenarios: the share of the family's held-out evolved rollouts (scenarios valid in every run) in which the run's items for that family were fired (a rule loaded, a skill opened), and the run's gain on the family. R3's money rules, which name 5 files, loaded in none.

5.2 What the loop did

Cycles and verdicts. In the 4 evaluation runs /*evolve* recorded 39 cycles (Figure 15; run by run in Figure 16). Of these, 14 were barren, most of them after the run had kept an item for every rule family; the other 25 swept 26 candidates between them, one cycle sweeping 2. *score.sh* kept 18 and killed 5; for the other 3 it returned RERUN, and the /*evolve* agent buried them without a verdict rather than sweep them again. A recheck was needed for 3 candidates, and 2 of them were kept after it. Of the 1,818 sweep rollouts, 1,494 were spent on confirm sweeps, which only candidates that had passed a screen paid for; every confirm sweep, with the runs it won net and its worst drop on one task, is drawn in Figure 21.

What the runs learned. H9 is that independent runs learn the same families, in similar forms. It is supported: every run kept an item for each of the four rule families, and none for the control (Figure 17, and every kept item in Table 7). Every run but R3 chose the same scopes: *shop/*** for the changelog and the clock, *shop/billing/*** for money and *shop/plugins/*** for exporters; they differed in form only for exporters, a gated skill in two runs and a rule in the third. R3 narrowed its money and clock rules: instead of a folder pattern, their paths name a short list of individual files, so the rules load only when the agent reads one of those files. Its held-out result shows what that cost. Its first money rule covered *shop/billing/***. The confirm sweep, over the 3 tasks then in the suite, won 5 runs and lost none, but exposure was unknown in 7 of its rollouts, so *score.sh* returned RERUN (Section 3.4). The /*evolve* agent buried the rule and, in the next cycle, proposed a narrower one. It and a later rule name 5 files between them, and the gates kept both. They loaded in 0 of the run's 30 held-out money rollouts, where the run's money pass rate stayed at 26.7% in both arms; the other runs' money rules loaded in at least 28 of 30. Its clock rule on *shop/*** was killed when a task the live harness passed in all its runs lost one run in the confirm and another in the recheck (gate 3). The rule that replaced it names 3 files and loaded in 10 of 30 held-out clock rollouts, and the run's clock pass rate fell from 56.7% to 50.0%. The gates judge a candidate on the tasks the loop has harvested, where these rules did load. Nothing in them can see that a scope is too narrow for tasks the loop has not yet met.

The tier rubric. H3 is that every kept item lands in a sensible tier and fires mainly in its own family’s work. The lab scored each kept item on three criteria: its form fits how it loads (a skill fired at least once, a rule has paths); its scope is limited (it has paths, or is a skill); and it fires more often on its own family’s rollouts than on the others’. The margin was every item scoring three of three. Of the 18 kept items, 15 did, so H3 is not supported. Every item that failed is a rule on shop/** and failed only the third criterion. Rules on shop/** load on almost every task of every family (in 93% to 100% of rollouts), and the 4 that passed it did so narrowly, firing on their own family at most 1.25 points more often than on the others.

Loaded or chosen. Of the 18 kept items, 16 are rules (Figure 18). The other 2 are skills, each an exporter checklist gated on shop/plugins/**. A gated candidate, in its own sweeps and in the gate test, was listed to the agent in 66 rollouts and opened in 28; an always-on candidate, likewise in its own sweeps and in the gate test, was listed in 198 and opened in 0. Elsewhere such a skill was opened, but rarely: the gate test’s placebos that were planted unchanged in the prune test were listed there 1,263 times and opened 46. A rule needs no such decision: it is in the context whenever the agent reads a file it covers. The gate test and the ablations below meet the same asymmetry.



Figure 18: A rule is loaded; a skill must be chosen. Claude Code injects a path-scoped rule whenever the agent reads a matching file. A skill is offered by its description, and its body enters the context only if the agent opens it: in their own sweeps and in the gate test, path-gated candidates were opened in some of the rollouts that listed them, and always-on ones in none. Of the items the loop kept, 16 are rules and 2 are path-gated skills; most candidates were rules too.

5.3 The gates against known answers

The verdicts. The gate test ran 29 candidates with known answers through the real pipeline (Section 4). H5 is that the gates are calibrated: they rarely keep a placebo, kill every harmful candidate, and keep at least 3 of the 4 known-good ones. None of the 20 placebos was kept; with so few candidates, the rate at which the gates keep placebos could still be as high as 17% (the upper end of a two-sided Clopper–Pearson 95% CI). None of the 5 harmful candidates was kept either, but score.sh killed only 2 of them and reached no verdict on the other 3 (below). And 2 of the 4 known-good candidates were kept, short of the 3 the margin required. H5 is not supported. Fisher’s exact test, which asks whether the known-good candidates were kept more often than the placebos, gives $p = 0.022$.

What decided them. Table 6 gives each kind of candidate’s verdicts. No placebo was opened on its screen: each was listed in all 8 candidate rollouts and invoked in none. score.sh killed 12 of them for never firing (gate 5) and returned RERUN for the other 8, whose screen tasks failed in every run of both arms, so that no gain was possible; the test’s procedure, which stops at the screen, recorded all 20 as KILLs. In all, 10 candidates (7 placebos, 2 harmful skills and 1

known-good skill) showed a positive gain on their screen although the agent never opened them, 9 of them by a single run on a single task. Gate 5, which asks that a candidate fired, turned them away, not their gain.

Harm was not tested. The 2 harmful skills were never opened and fell to gate 5, like the placebos. The 3 harmful rules loaded in every candidate rollout of their screens, but a harmful item shows itself only on a task that passes without it, and their screens ran, as screens do, on failing tasks. One screen included a task that passed without the rule, and the rule broke it; `score.sh` still returned RERUN, because no task in that screen could show a gain. The test therefore shows that the gates never keep what never fires, not that they catch harm. In the loop itself, 2 of the 5 KILLs failed a regression gate, and 1, the clock rule of R3 above, failed nothing else.

The positives. The known-good candidates split along the tier line: the rule and the gated skill were kept, and the 2 always-on skills, never opened on their screens, were lost.

Do the gates add value? H6 is that they do: a harness that also keeps what the loop buried should be no better than evolved, and the margin asks that the upper end of its difference's interval be below +5 points. `accept-all`, R2's harness plus the one candidate that run buried, scored -8.9 points against evolved [-16.1, -1.7], an interval that resamples scenarios only, since the arm ran in a single run. That meets H6's margin, though the permutation test, at $p = 0.075$, does not by itself rule out chance. That candidate, a second clock rule, was not buried by a gate: both arms passed every run of its screen, `score.sh` returned RERUN, and the `/evolve` agent buried it as overlapping a live rule. The loss falls on families A (-13.3) and C (-17.8) more than on E (-4.4), the family the added rule is about. The rule, on `shop/**`, was in context in every A and C rollout, so it could have caused the loss there, but the arm also ran in a later batch, so part of the difference may be the shift between batches. The margin is met, but it rests on one burial the `/evolve` agent made, not on the gates, so H6 is inconclusive.

5.4 What the harness costs

Always-on context. Two hypotheses concern this cost. H8 is that evolved comes within 10 points of the better of two harnesses that know all four rules, `kitchen` and `ideal`, while putting fewer characters into every turn's context. H7 is that the scopes earn their place: `flat`, the same texts without their paths, puts more text into every turn and is no better (the upper end of its difference's interval below +5 points). Figure 19 places each arm of R1 by its pass rate on the rule families and by the characters it puts into every turn's context. The rates are over every scenario valid in that run, one more than the set valid in every run, so they differ slightly from those of Section 5.1. `evolved`'s pass rate was 87.5%, with no always-on text beyond the 559-character `CLAUDE.md` that every arm has. `kitchen`, which appends the contribution guide to that file (2,519 characters in all), reached 76.4%, 11.1 points lower [0.0, +22.5]; `flat`, `evolved`'s texts with every scope removed (1,688 characters), 75.0%, 12.5 lower [-1.7, +28.3]; and `ideal`, written by hand with the four rules known (728 characters), 48.6%, 38.9 lower [+20.0, +58.1]. `evolved` scored higher than both harnesses that knew the rules, though against `kitchen` the interval reaches zero, and it put less always-on text into every turn than either, so H8 is supported; `flat` put more text into every turn and was no better, so H7 is supported. These arms ran in one later batch together with `none2`, after the primary arms, and their intervals resample scenarios only. Across that same boundary, the A/A pair differed by -4.2 points on the rule families [-9.2, 0.0], in the direction that would widen every gap above.

Money and time. Costs are in US dollars, as Claude Code reported them for each call. In R1 to R4 a held-out rollout cost \$0.121 on average with no harness and \$0.149 with the evolved one, 23% more, and took 22.1 turns instead of 18.2 and 68 s instead of 57 s. The rise was largest on the exporter family (38%), smallest on the control family (5%), and 12% to 22% on families A, B and E. Under Claude Sonnet 5 the harness of R1 made a rollout cheaper, \$0.205 against \$0.228. Running the loop cost each evaluation run \$32.76 to \$128.78 in sweep rollouts, R3 the most, and \$2.39 to \$4.09 for the `/evolve` agent's own turns, beside \$10.08 to \$11.78 for the scripted sessions it learned from.

Where the words reached the agent. The families show the tier asymmetry again. `flat` passed every A, B and E rollout but 0% of C: without its paths, the exporter checklist became an always-on skill, listed in every C rollout and opened in none (evolved passed 70% of C). `ideal` wrote its advice on the changelog and on the clock as always-on skills; each was listed in all 90 of the arm's held-out rollouts and opened in none, and families A and E stayed where

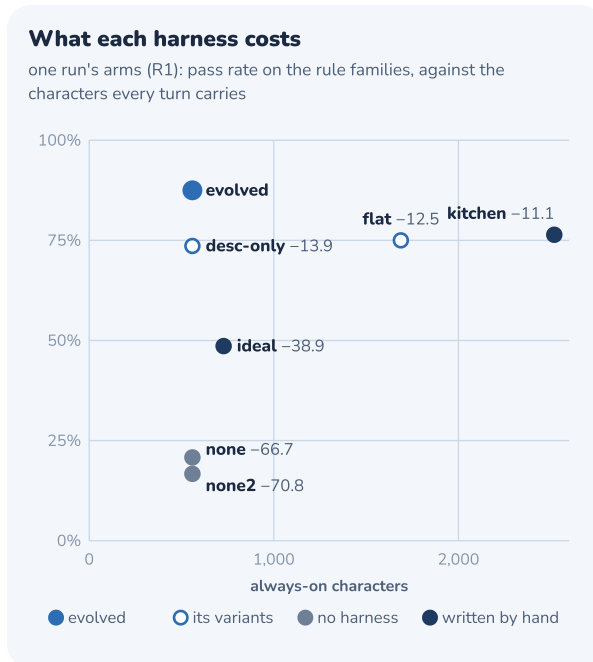


Figure 19: What each harness costs, in R1: pass rate on the four rule families against the characters present in every turn's context. Filled blue is the harness the loop evolved; hollow blue, its texts with every scope removed (flat) or every skill body replaced (desc-only); grey, no harness (none), measured twice; navy, the two harnesses that state the rules outright: kitchen, the contribution guide appended to CLAUDE.md, and ideal, written by hand. Beside each arm, its difference from evolved, in points.

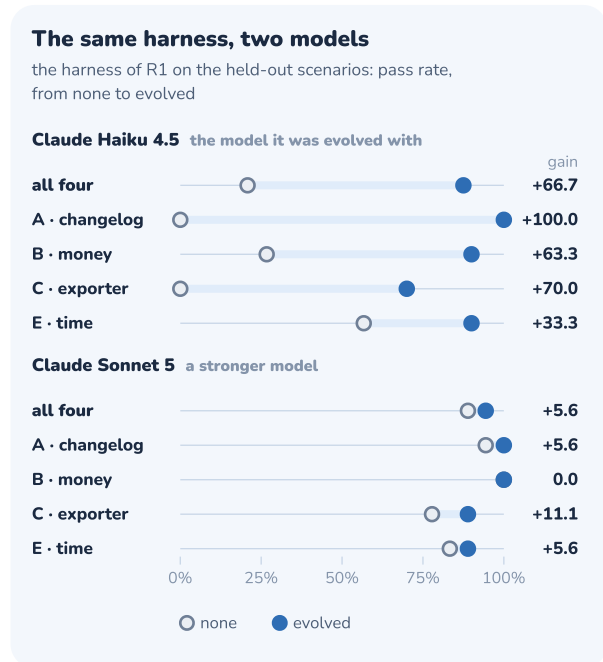


Figure 20: The harness of R1 on the held-out scenarios, under the model it was evolved with and under a stronger one. Each line runs from none (grey) to evolved (blue), and the number beside it is the gain in points. Under the stronger model, little is left to gain.

none left them (0% and 56%, against 0% and 57%). In R1, what mattered was not who wrote the advice but how it was offered: tied to the files a task touches, as a rule or a gated skill, it reached the agent; offered on every turn as an always-on skill, it was not opened.

The description alone. H11 is that a skill's description, the line the agent sees in the listing, carries much of the skill's effect by itself; the margin puts "much" at 50% or more of evolved's gain. desc-only tests it: it keeps each skill's description and replaces its body with neutral filler. Over the four families together it recovered 79% of evolved's gain [58%, 100%], but it could rewrite only 1 of R1's 4 items; the rest are rules and kept their words. In C, the one family whose skill was rewritten, the arm recovered 16% [0%, 56%], below half. The description still got the rewritten skill opened, in 16 of 18 C rollouts, but with filler in place of its body C's pass rate was 11%, against 70% for evolved, and its 2 passing rollouts were among the 6 that printed the replaced body through git (Section 7). One rewritten skill cannot settle the question, and H11 is inconclusive.

5.5 A stronger model, and pruning

The same harness under two models. Under Claude Sonnet 5, the harness of R1 has little left to do (Figure 20). none's pass rate on the rule-family scenarios was already 88.9%, against 20.8% under Claude Haiku 4.5, and the harness raised it to 94.4%: a gain of +5.6 points [-2.8, +13.9], against +66.7 [+51.7, +80.8] under the model it was evolved with; each comes from a single run, over every scenario valid in it, so both intervals resample scenarios only. H13 is that after a model upgrade, re-measuring the harness finds items the weaker model needed that the stronger one no longer does. /prune, in one pass at $k = 3$ with its rollouts pinned to the stronger model, found 2 of the harness's 4 items removable, the money rule and the exporter checklist, so H13 is supported. Its margin asks for one such item, and the

money rule is one on both measures: the stronger model passed every held-out money scenario with no harness at all. Removing the changelog rule cost 5 runs; removing the clock rule cost 1, on a task the live harness had passed in every run, where losing even one run keeps an item (Section 3.6). The exporter checklist that `/prune` removed is less certain: it still showed the largest gain on the held-out scenarios, +11.1 [-11.1, +38.9] on C, and one pass at this k cannot tell a small effect from none. Under a stronger model, part of a harness evolved for a weaker one may be redundant, and one pass of re-measurement flagged which part, within the limits of that pass.

Pruning against known answers. H10 is that `/prune` deletes planted useless items that fired, keeps the items that are needed, and never deletes an item that never fired; the margin asks every planted item to get its expected verdict. Under Claude Haiku 4.5, `/prune` kept all 4 of R1’s own items, as expected: removing any of them cost at least 9 runs net. Of the 4 planted items it matched 2 (6 of 8 overall), so H10 is not supported. It deleted one placebo always-on skill and skipped the gated skill that no task reaches, as expected. It kept the other placebo: in its sweep the agent opened it in none of the 18 rollouts that listed it, so it could not have been deleted. The placebo got REJECT rather than UNMEASURED because the conditions for REJECT are checked first (Algorithm 3), and 2 tasks lost runs when it was removed. The last planted item, the always-on skill that the agent never opens, got no verdict. The margin asks for UNMEASURED, which only a sweep can return, and the plan leaves out an always-on skill that never fired in earlier sweeps, since no sweep could delete it (Section 3.6); the item stayed in the harness, but without the verdict the margin asks for. No item that never fired was deleted.

5.6 A second repository

H14 is that the loop also helps on a repository that nobody in this study built. On structlog the loop found nothing to learn. Its training harvested 8 tasks but only 1 lesson: in 7 of the 8 sessions the agent passed the project’s own tests and its linter, at the version the project pins, at the first attempt. Nothing recurred, every `/evolve` cycle was barren, and the loop kept nothing, so evolved was byte-identical to none and the benchmark on the 4 held-out tasks had nothing to compare. With the two harnesses identical, H14 is inconclusive. At this correction rate the loop had nothing to do, and the run says nothing about whether a learned harness transfers.

5.7 Predicting a verdict from a candidate’s text

The judge was off in every run; its predictions were replayed afterwards on each run’s candidates, never from a verdict, but over the tasks on which each candidate’s own sweeps had recorded it in context (Section 3.7), so they are not the predictions cortex scope would have made before those sweeps. H15 is that the judge’s *relevance*, the share of the tasks a candidate would load on that the candidate is actually about, separates the candidates that are kept from those killed on a regression gate, while *breadth*, the share of all tasks a candidate would load on, which needs no judge, does not. To separate means that every kept candidate scores above every killed one. On the evaluation runs, *relevance* did not separate the candidates *score.sh* kept from the 2 it killed on a regression (lowest kept 17%, highest killed 75%), and neither did *breadth* (8% against 58%), so H15 is not supported. So few regression kills make a thin comparison. Skipping the evaluation runs’ candidates below Cortex’s warning floor of 35% would have avoided sweeping 3 candidates that were later buried, and lost 6 that were kept. H16 is that the judge’s predicted invocation rate, how often it expects the agent to open a skill, tracks the rate measured in the rollouts; with so few skills, its margin asks only that the predictions rank the skills in the same order as the measurements. The evaluation runs kept only 2 path-gated skills, so the test also takes the 2 that the development run kept. For these 4 skills the judge predicted a higher invocation rate than was measured for 3, by up to 54 points. The rank order is preserved, which meets the letter of the margin; but 3 of the 4 predictions are tied at the top, so the order rests on the remaining skill alone, and H16 is inconclusive.

6 Related work

Cortex sits among systems that write and revise the text an agent reads, studies of whether such text helps a coding agent, and the methods used to evaluate noisy agents and to build their tasks. Most of its parts have precedents; this section says where, and what differs.

Evolving what an agent reads. Agents that learn from experience keep what they learn as text: reflections on a failed attempt [49], programs admitted to a library once a model judges the task done [51], insights kept or dropped by a model’s votes [73], workflows induced from past trajectories [55], and itemised playbooks grown by small edits [68]. Optimisers search over prompts and programs against a metric [1, 25], and over agent designs and an agent’s own code [18, 42, 66]. The newest systems evolve the harness around a fixed model [9, 10, 12, 27, 30, 59, 65, 72] or synthesise skills from trajectories [3, 34, 37, 45, 50, 53, 61]; some of them run in Claude Code itself, on public benchmarks [3, 12, 61]. Cortex adapts DarwinX’s contract of extending without breaking what is preserved (Section 3.5), and differs from much of this work in two ways. Its tasks come from one repository and one developer, not from a benchmark. Its baseline is the live harness, measured again in the same sweep as the candidate, as HDSO’s control arm is [45]; many systems instead compare a candidate with a score recorded earlier, such as cached outcomes without the skill [34], the results of the current state reused until an update is accepted [53], or the best score so far [50, 59]. Its verdict is computed by a script, as the acceptance rules of SkillGen, SkillOpt and LiteEvo are [12, 34, 61], while a model proposes what is tried, much as HarnessX’s critic chooses which candidates reach its deterministic checks [10]. In some systems a model takes part in the verdict itself: DarwinX’s promotions are adjudicated by a verifier agent [72], and HDSO’s promotions include a review by its curator model [45]; model judges carry known biases, such as favouring the answer shown first [75]. Noise makes these differences matter. In HarnessDev, when models evolved their own harnesses, the score of one and the same version could move by nearly five points either way, only two of sixty-four changes gained clearly beyond that noise, and the scores on the feedback tasks and on held-out tasks moved in the same direction about half the time [58]. DarwinX’s score on its training subset saturated while its held-out score did not [72]. R3’s narrowed money and clock rules are a small instance of the same gap: they loaded on the tasks the loop had harvested, but in 0 of the run’s 30 held-out money rollouts and 10 of its 30 clock ones (Section 5.2).

Crediting and retiring an item. That an edit can be selected without ever reaching the model is known. In Demo-Evolve’s experiments, the harness that a self-rollout baseline selected was credited with a gain, yet the hook it added appeared in none of the model requests the authors inspected [9]; and in HarnessDev’s coding domain, about one in six of the components that models wrote into their own harnesses was never observed to trigger [58]. Some systems and studies record exposure. HarnessX’s change manifests name a trace feature that must appear if an edit fired, which its critic checks after shipping [10], and HDSO logs each request for a skill’s full text [45]. SkillMAS credits only skills that were invoked or whose steps appear in the trace, uses the credit to rank skills, and holds a new skill in a validation pool until later rounds show enough use of it [39]. Skill-Use credits a skill’s execution only once it is triggered, and measures its effect on the runs that triggered it [17]. Skill Following compares matched runs with and without skills only on the tasks where the agent retrieved one, and on coding tasks finds this effect negative for several models that appear to gain when tasks with and without retrieval are compared [11]; a review of such designs cautions that restricting a paired comparison to the tasks where the treated run triggered a skill does not by itself identify the effect of invoking it [69], and Cortex’s exposed tasks are likewise defined by what happened in the arm where the item exists (Section 3.4). LiteEvo drops a new processor that never fires when replayed, much as Cortex refuses a candidate that never fired, and shows its curating model which deployed components never fired [12]. WikiSkill sets the question aside by placing every skill in the prompt in full [50]. SLIM decides whether to retire a skill from a comparison with and without it restricted to the validation tasks on which it was loaded, and leaves a skill alone until it has been loaded often enough [47]; we found no system that applies the same restriction when deciding whether to add an item, or that lets unknown exposure block a verdict. Deletion rules mostly retire what looks weak: a small measured effect [54], failure evidence from enough runs that used a skill [56, 70], a negligible effect of removing it on the tasks that loaded it [47], low utility in recent calls [41], an insight whose votes fall to zero [73], low-utility entries once the context grows too long [68], or a component whose edits have stopped gaining [59]. A Shapley valuation of a skill’s parts, which separates content from context cost with length-matched padding, prunes the parts of lowest value first [28]. SkillReducer, which compresses skills rather than deleting them, reads a skill that never triggers as a sign of obsolescence [15]: the opposite reading to UNMEASURED. SLIM comes closest to UNMEASURED in mechanism [47] (Section 3.4); the argument behind it is made most explicitly by a study of skill retirement for physical agents after model upgrades. When clauses were deleted because the tasks that authorised an action still passed without them, safety conditions those tasks never exercised, such as consent, were left untested: an “unmeasured support gap” [64]. Its remedy is to build counterfactual tasks that exercise the condition; Cortex keeps the item and suggests harvesting a task that needs it. An analysis of why instruction files only grow makes a related point: age and rare triggering alone do not make an instruction outdated [8]. Two observations bear on /prune’s one-at-a-time removals: of two redundant instructions, each looks removable alone [8], and an item judged in isolation can mislead, because its effect depends on what else is

loaded [71, 74].

Context files, rules and skills. Whether repository context helps a coding agent is contested. Context files written by a model lowered task success slightly and files written by the repositories’ developers raised it marginally, neither change significant, while both raised cost, the model-written ones by about a fifth [16]; an ablation with Claude Code and Codex on real repositories, which delivered repository guidance either always on or as topic files the agent retrieves on demand, found that the delivery strategy had no measurable effect on correctness [24]. In Claude Code, on the SWE-bench Verified tasks it solved only sometimes, randomly drawn community rule files did as well as curated ones, which the authors read as the presence of rules mattering far more than their content [71]. A comparison with and without an item can therefore reward text as such, and a placebo candidate is the control for that. Cortex’s placebos, all always-on skills, were never opened on their gate-test screens (Section 5.3), so its gates never faced a placebo that loads. A repository’s guidance file, tuned with synthetic probes, raised the resolve rate of the model that tuned it; for a second model, in one run each, the guidance tuned on the first more than halved the resolve rate, and its own tuned guidance did no better than none [48]. Skill Issue, the closest in setting, optimised one skill file per repository for Claude Code on tasks mined from the repository’s history, with gains that its tasks could not separate from noise [26]; DeltaSelect’s author tested edits to the skills and instructions of a Codex-based harness on eight tasks, one run each, and kept changes by judgement [13]. On loading, Skill-Use finds triggering a skill and following it to be separate bottlenecks, and that placing a skill’s full text in context mainly helps the agent recognise it as relevant [17]. An agent given workflows as extra actions called them in fewer than a fifth of tasks [55]. In a benchmark whose skill library grows, GPT-5-based systems, Codex among them, invoked almost none of the skills offered at the final stage; given only each task’s own skills, Codex invoked most of them, and Claude Code, counted by the files it read, about one in seven [23]. In Claude Code, on one coding task, an agent given a memory store seeded with facts relevant to the task never chose to consult it, while the same facts, injected by the harness on matching triggers such as file paths, arrived in every run [43]. Text placed in context can also hurt: on web development tasks, injecting a relevant skill lowered the average pass rate of all four models tested, and for two of them an irrelevant skill of the same length accounted for most of the loss [62]. Cortex meets the same division, between text that is loaded and text the agent must choose, inside its acceptance loop, where 16 of the 18 items kept were rules (Section 5.2).

The harness and the model. The benefit of an evolved harness need not grow with the model’s strength: in one study it peaked at a model of middling strength, and the strongest gained little, consistent with its already solving many of the tasks [31]. Planning acts as a scaffold for weaker coding models and mainly saves cost for stronger ones, and predefined tools help weaker models, while models fluent in the shell do as well, more cheaply, with the shell alone [14]. A harness is argued to be specific to its model [30, 65] and to need re-adapting when the model changes [30], although AHE’s own evolved harness, moved unchanged to other base models and to other reasoning settings of its own model, still gained on each [30]. For data agents, many layers that compensate for a model’s weaknesses are expected to be absorbed by the model, while knowledge of the environment keeps its value [40]. DarwinX notes that a behaviour the model internalises leaves the corresponding harness edit as dead weight, and that how much of a harness survives a change of model has not been measured [72]. Cortex’s items hold knowledge of the second kind, the conventions of one repository; yet under a stronger model, which already passed most rule-family scenarios unaided, the harness gained little, and one pass of re-measurement flagged 2 of its 4 items as removable (Section 5.5).

Measuring under noise. Run-to-run variance in agent evaluations can be as large as the improvements papers report: ten runs of each of six agent configurations on SWE-bench Verified, at two temperatures, spread their single-run scores over two to six points [7], and a byte-identical re-run at temperature zero changed six of seventy outcomes [44]. The remedies are repeated samples and paired comparison question by question [36, 52], which Cortex’s sweeps apply. For accepting a change, PACE shows that accepting every prompt edit that raises the score commits many that an audit later labels false, and proposes sequential tests that remain valid whenever they are stopped [46]; EvoPathBench finds that the selection of candidates, more than their generation, limits self-evolving agents [29]. Some loops estimate a noise band from repeated runs: RRSI does not let a gain inside it admit a coding candidate on its own [59], and LiteEvo reverts a candidate only when it falls below the best score by more than a margin that includes the band [12]. PACE also tests its acceptor against a known answer, a harmful instruction planted for the proposer to remove, in prompt evolution on arithmetic and science questions; its other runs, like EvoPathBench’s, score real proposals on held-out data the acceptance rule never saw [29, 46]. Irrelevant skills of matched length have served as comparison arms [20, 62]; we

know of no study that plants placebo candidates in a coding agent’s acceptance pipeline to count how many it keeps.

Tasks, and what a sandbox leaks. Cortex’s tasks follow SWE-bench’s format, with a check that fails before the fix and passes after it [22]; benchmarks now refresh such tasks continuously [6, 67], and build them for training from real issues [38] or synthesise them at scale [60]. Tasks drawn from real use are recent: REAP builds them from developers’ sessions with a production coding assistant in one monorepo [21], and SWE-Together reconstructs them from public user–agent sessions, replaying the users’ corrections through a simulated user [57]. Both serve to compare agents. Skills distilled from each developer’s own sessions have been tested by replaying that developer’s held-out sessions with a simulated developer [19], and a user’s corrections have been compiled into rules and runtime checks for Claude Code and Codex [77]; neither decides what to keep by a measured comparison. Cortex harvests one developer’s sessions to improve that developer’s harness, and writes the corrections into each task’s check. A task can give its answer away: in about a third of the SWE-bench instances one agent resolved, the solution was stated in the issue or its comments [2], and the agentic benchmark checklist asks that agents be kept from the ground truth [78]. The git-history route a rollout can take (Section 7) is known. Terminal-Bench requires that a task’s repository not contain its future commits [35]. Audits of agent trajectories find agents reading future fixes in the local git history, less often but still under an explicit instruction not to [32], and single out git history as the main recurring route to passes not earned, in runs under Claude Code among others [33]. Deleting a repository’s branches and tags leaves later commits recoverable, which is why SWE-Bench Pro Verified rebuilds each repository as a single commit [76]; Skill Issue’s sandboxes go further and remove the repository’s `.git` directory [26]. A related route has been noted for solutions: a tool that shows the local, uncommitted diff can reveal a solution backed out of the working copy [21]; Section 7 reports the same route for harness items.

What is new. Paired gates with a net threshold [34, 45], a no-regression guard [10, 65], context cost as an objective [15, 27], measured deletion [47, 54], retiring a skill only after enough exposure [47, 70], an acceptor tested against a planted known answer [46], and the case for re-testing a harness when its model changes [30, 72] all have precedents. To our knowledge, what this paper adds is their combination for one developer’s repository and three things we did not find elsewhere: the same exposure rule for adding an item and for deleting one, so that both verdicts compare the harness with and without the item only on the tasks where it entered the context, with unknown exposure blocking either verdict (SLIM applies such a comparison to retirement alone [47]); a coding agent’s acceptance pipeline tested with planted placebo and harmful candidates as well as known-good ones; and Claude Code’s loaded rules and agent-chosen skills compared as tiers inside the loop that decides what to keep. AHE’s evolving agent also chooses between a system prompt that applies to every task and skills loaded on demand, but its analysis of where the gain lies does not compare the two [30].

7 Limitations

One repository, built for the purpose. The lab, its house rules, its tasks and its checks all come from the same hand, and the lab was calibrated so that the model had room to improve: before its first session, scenarios the model already solved unaided were redesigned until it broke the rule. The four house rules are ones a machine can check, and the one with the largest gain, family A’s, is checked only by whether the changelog gained a line. Of the 30 held-out scenarios, 15 were written after the development run (Section 4), when the items it had kept were known; they were written from CONTRIBUTING.md and the family definitions alone, never from an evolved item. Conventions that need judgement to check are not tested at all. The results show that the loop learns conventions of this kind, in a repository of this kind; they do not show that it learns others.

A small model’s gain. Every primary number is Claude Haiku 4.5’s. Under Claude Sonnet 5, the harness of R1 gained +5.6 points [−2.8, +13.9] where it had gained +66.7, because the stronger model already followed most of the house rules without a harness (Section 5.5). In this study, what the loop was worth followed how often the model broke the conventions; a single harness under a single stronger model is too little to say how it varies across models.

No result on a real repository. On the one repository we did not build, the agent was rarely corrected, so nothing recurred and the loop kept nothing, and its benchmark had two identical harnesses to compare (Section 5.6). Whether a harness that Cortex learns in a real repository helps on tasks drawn from real work there is untested.

Scripted corrections, and few runs. The developer in the lab is a script: it replies to each mistake in the same words every time, and never misreads a change or changes its mind. There were 4 evaluation runs and 1 control run, so the variation between runs rests on few runs: the bootstrap can draw only a few distinct sets of runs, and its intervals may understate how far a further run could fall, as R3 fell below the others (Section 5.1). With 1 control run, a test that permutes runs cannot give H4 a p below 0.2, so its verdict is descriptive. The loop’s confirm sweeps ran each task 3 times per arm, and the ablation arms each held-out scenario 3 times, where a single lost run can be luck; the recheck exists for that reason, and the ablations’ intervals are wide.

What measurement costs. Every verdict is paid for in rollouts (Section 5.4). The costs reported there are Claude Haiku 4.5’s prices on this repository’s small tasks; a larger repository, a dearer model or a larger k would cost more.

What a rollout could read. A sandbox isolates what is on disk, not what is in git. The clone keeps the repository’s history, so a rollout could find earlier tasks’ fixes in past commits, and a replayed training scenario could find the later commit that fixed it in its session. And when an item’s file is already tracked at the commit a rollout starts from, an arm that lacks the item sees `git status` list that file as deleted, and `git diff` prints its text. In the lab neither route reached a held-out scenario’s fix, which was never committed to the lab’s repository. Under Claude Haiku 4.5 the first route was rare: 2 of the 1,809 loop rollouts that could be checked printed a harvested fix or the task store. The second was not: of the 595 held-out none rollouts, 112 saw the run’s items named by `git status` and 2 printed an item’s text, and on the rule families the rollouts that saw the names passed less often (9.7% of 93, against 23.6% of the 382 that did not). Under Claude Sonnet 5 reading an item’s text was common, in 67 of the 168 none rollouts, held-out and training together, and in 72 of the 300 removal rollouts of `/prune`, but the none rollouts that read one did not pass more often than those that did not, and for the items found removable, the removal rollouts that read the removed item passed less often (28 of 35, against 110 of 115). On structlog, 1 of the 8 training sessions and 19 of the 40 benchmark rollouts read the upstream fix through git, so that run does not show an agent working unaided. A sandbox repository that holds a single commit, with the harness installed as untracked files, would close both routes.

No independent replication. No one else has yet repeated the study.

8 Conclusion

A coding agent’s harness is text that a developer edits by impression, and a single run of the agent is too noisy to say whether an edit helped. Cortex treats the harness as something to measure. It turns the developer’s own sessions into tasks, keeps a skill or rule only if a repeated comparison with the live harness, task by task, shows a net gain and no task that collapses, and deletes one only if removing it costs nothing measurable. One principle runs through both directions: an item is credited or blamed only where it entered the agent’s context, so a candidate that never fired is not kept, and an item that never fired is not deleted.

In a replicated evaluation with Claude Haiku 4.5, in a repository built so that the model had conventions to learn, the loop learned them. On held-out scenarios it never saw, the harnesses evolved by 4 independent runs beat no harness by +55.0 points on the four rule families (95% CI [+34.1, +74.1]) and moved the control family by −0.8 points. They added no text to what the agent reads on every turn. In the run that measured it, how the advice was offered mattered more than who wrote it. A rule enters the context whenever the agent reads a file it covers, and most candidates, and 16 of the 18 items kept, were rules. A skill’s body enters only if the agent opens it: the always-on skills written by hand for the `ideal` arm were listed in all 90 of its held-out rollouts and opened in none, and their families stayed where no harness left them. Advice reached the agent when it was tied to the files a task touches, but not so narrowly that new tasks missed it (Sections 5.2 and 5.4).

The results are as clear about where the loop stops. Its gates kept none of the 20 placebos and none of the 5 harmful candidates. But on their screens the agent never opened the placebos or the harmful skills, and the screens of the harmful rules could show no gain, so the test shows that the gates never keep what never fires, not that they catch harm. Only 2 of the 4 known-good candidates were kept; the 2 always-on skills, never opened on their screens, were lost. Under Claude Sonnet 5, which already followed most of the house rules unaided, the harness of R1 gained +5.6 points (95% CI [−2.8, +13.9]) where it had gained +66.7 points under Claude Haiku 4.5. One pass of `/prune` at $k = 3$, with its rollouts on that model, flagged 2 of the harness’s 4 items as removable, though one pass at that k cannot tell a small effect from none. In this study, what the loop was worth followed how often the model broke the conventions.

On structlog, the one repository we did not build, the agent was rarely corrected, nothing recurred, and the loop kept nothing. The positive result therefore rests on one synthetic repository with house rules a machine can check, a scripted developer and one small model. Cortex’s optional judge, replayed afterwards, did not anticipate the verdicts, on a thin comparison (Section 5.7).

The next steps follow from these limits. The first is a real repository whose conventions the model does break, the test this study could not run. The second is a test of harm, in which harmful candidates are swept, as in a confirm, over tasks that pass without them as well as tasks they could help, so that a gate can see what they break. The third is to run each rollout in a repository that holds a single commit, the broken state without the task store, with the harness installed as untracked files, which closes both routes by which rollouts read what their harness did not give them (Section 7). The fourth is conventions that need judgement to check, which a task’s script cannot decide alone; measuring them without letting a model decide the result (Section 3.7) is an open problem. Cortex and the lab are at <https://github.com/samugit83/cortex>: tag v1.0-eval is the system as evaluated, and tag v1.0.5 adds the lab’s data and the scripts that regenerate every number in the paper from it, offline. The data are rows exported from the runs’ records and from Claude Code’s session records; the records themselves are not released.

Acknowledgement

AI participation was substantial and continuous across design, implementation, evidence review, statistical verification, critical review, and editing.

References

- [1] Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziemis, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. In *International Conference on Learning Representations (ICLR)*, 2026.
- [2] Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. SWE-Bench+: Enhanced Coding Benchmark for LLMs. arXiv:2410.06992, 2024.
- [3] Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Christian Alexander Calvo, Weiyuan Chen, and Tu Vu. EvoSkill: Automated Skill Discovery for Multi-Agent Systems. In *Conference on Language Modeling (COLM)*, 2026.
- [4] Anthropic. How Claude remembers your project. Claude Code documentation, 2026. URL <https://code.claude.com/docs/en/memory>. Accessed 2026-09-27.
- [5] Anthropic. Extend Claude with skills. Claude Code documentation, 2026. URL <https://code.claude.com/docs/en/skills>. Accessed 2026-09-27.
- [6] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025.
- [7] Bjarni Haukur Bjarnason, André Silva, and Martin Monperrus. On Randomness in Agentic Evals. arXiv:2602.07150, 2026.
- [8] Kushal Chakrabarti. Why Does CLAUDE.md Keep Growing? Catastrophic Remembering in Agentic Coding. arXiv:2608.11095, 2026.
- [9] Lirong Che, Yuzhe Yang, Peiwen Lin, Xu Cao, Chuang Wang, Xueqian Wang, and Jian Su. DemoEvolve: Demonstration-Guided Harness Evolution under Sparse Feedback. arXiv:2605.24539, 2026.
- [10] Tingyang Chen, Shuo Lu, Kang Zhao, Weicheng Meng, Hanlin Teng, Tianhao Li, Chao Li, Xule Liu, Jian Liang, Zhizhong Zhang, Yuan Xie, Heng Qu, Kun Shao, and Jian Luan. HarnessX: A Composable, Adaptive, and Evolvable Agent Harness Foundry. arXiv:2606.14249, 2026.

- [11] Seonghyeon Cho and Chanjun Park. Skill Following: Evaluating Actual Skill Use in Retrieval-Enabled LLM Agents. arXiv:2609.00549, 2026.
- [12] Euntae Choi, Sumin Song, and Sungjoo Yoo. LiteEvo: Automated, Cost-Efficient Harness Evolution for Generalization to Unseen Tasks. arXiv:2609.33146, 2026.
- [13] Nicholas J. Conn. DeltaSelect: Affordable A/B Testing for Coding Agents. arXiv:2609.19607, 2026.
- [14] Run-Ze Fan, Zihao Zhang, Simin Ma, Yebowen Hu, Shouju Wang, Kaiqiang Song, Fei Liu, Hamed Zamani, and Xiaoyang Wang. An Empirical Study of Harness Design for Coding Agents. arXiv:2609.20804, 2026.
- [15] Yudong Gao, Zongjie Li, Yuanyuan Yuan, Zimo Ji, Pingchuan Ma, and Shuai Wang. SkillReducer: Optimizing LLM Agent Skills for Token Efficiency. arXiv:2603.29919, 2026.
- [16] Thibaud Gloaguen, Niels Mündler-Sasahara, Mark Niklas Müller, Veselin Raychev, and Martin Vechev. Evaluating AGENTS.md: Are Repository-Level Context Files Helpful for Coding Agents? arXiv:2602.11988, 2026.
- [17] Jinyi Han, Yuanjian Xu, Ying Liao, Xinyi Wang, Zishang Jiang, Zixiang Di, Fanyang Lu, Zhichao Hu, and Yanghua Xiao. Skill-Use: Can LLMs Actually Use Skills in Agentic Harnesses? arXiv:2608.04828, 2026.
- [18] Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems. In *International Conference on Learning Representations (ICLR)*, 2025.
- [19] Shuyan Huang, Kai Du, and Andrew Lan. Do Personalized Skills Help Coding Agents? An Empirical Study of Developer Interaction Histories. arXiv:2608.10319, 2026.
- [20] Wei-Jung Huang. Do LLM-Generated Skills Make Better AI Data Scientists? A Component Ablation Across Data-Science Workflows. arXiv:2607.07504, 2026.
- [21] Smriti Jha, Matteo Paltenghi, Chandra Maddila, Vijayaraghavan Murali, Shubham Ugare, and Satish Chandra. REAP: Automatic Curation of Coding Agent Benchmarks from Interactive Production Usage. arXiv:2604.01527, 2026.
- [22] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *International Conference on Learning Representations (ICLR)*, 2024.
- [23] Zixuan Ke, Vaidehi Patil, Haizhou Shi, Yang Li, Ye Liu, Sarath Shekkizhar, Anurag Koul, Jiayu Wang, Xuan Phi Nguyen, Semih Yavuz, Mohit Bansal, and Shafiq Joty. EvoHarnessBench: Can Your Agents Keep Pace with an Evolving Harness? arXiv:2609.04280, 2026.
- [24] Prakhar Khatri. Do Context Files Help Coding Agents? A Two-Agent Ablation Study on Real Repositories. arXiv:2607.27250, 2026.
- [25] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling Declarative Language Model Calls into State-of-the-Art Pipelines. In *International Conference on Learning Representations (ICLR)*, 2024.
- [26] Mykhailo Kozyrev, Andrei Kozyrev, and Anton Podkopaev. Skill Issue: Lessons from Optimizing Repository SKILLS for Coding Agents. arXiv:2609.12742, 2026.
- [27] Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-End Optimization of Model Harnesses. In *Conference on Language Modeling (COLM)*, 2026.
- [28] Tao Li, Junfeng Liu, Qinghua Zhao, Yifan Li, Lei Wang, Bo Shao, Xuejun Liu, and Linjun Shou. What Is a Skill Worth? Structure-Aware Shapley Valuation of Agent Skills. arXiv:2608.04562, 2026.
- [29] Hongqiang Lin, Chao Liu, Xiaofan Bai, Xuan Jin, Yuhong Li, Nenggan Zheng, and Xipeng Cao. Beyond Endpoint Performance: Process-Level Evaluation of Self-Evolving Agents. arXiv:2609.24663, 2026.

- [30] Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic Harness Engineering: Observability-Driven Automatic Evolution of Coding-Agent Harnesses. *arXiv:2604.25850*, 2026.
- [31] Minhua Lin, Juncheng Wu, Zijun Wang, Zhan Shi, Yisi Sang, Bing He, Zewen Liu, Tianxin Wei, Zongyu Wu, Zhiwei Zhang, Dakuo Wang, Xiang Zhang, Benoit Dumoulin, Cihang Xie, Yuyin Zhou, Suhang Wang, and Hanqing Lu. Harness Updating Is Not Harness Benefit: Disentangling Evolution Capabilities in Self-Evolving LLM Agents. *arXiv:2605.30621*, 2026.
- [32] Nikolai Ludwig, Wasi Uddin Ahmad, Somshubra Majumdar, and Boris Ginsburg. Shortcutting the Fix: Identifying and Categorizing Agentic Exploits in Software Engineering Benchmarks. *arXiv:2609.06780*, 2026.
- [33] Weijun Luo, Kelvin Luu, Xinyi Liu, Guangze Luo, Miguel Romero Calvo, Soham Dan, Daniel Yue Zhang, Ying Liu, and Mohamed Elfeki. Maintaining Benchmarks Against Increasingly Capable Agents: Detection and Remediation of Unearned Passes. *arXiv:2609.34262*, 2026.
- [34] Yuchen Ma, Yue Huang, Han Bao, Haomin Zhuang, Swadheen Shukla, Michel Galley, Xiangliang Zhang, and Stefan Feuerriegel. SkillGen: Verified Inference-Time Agent Skill Synthesis. *arXiv:2605.10999*, 2026.
- [35] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, et al. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. In *International Conference on Learning Representations (ICLR)*, 2026.
- [36] Evan Miller. Adding Error Bars to Evals: A Statistical Approach to Language Model Evaluations. *arXiv:2411.00640*, 2024.
- [37] Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2Skill: Distill Trajectory-Local Lessons into Transferable Agent Skills. *arXiv:2603.25158*, 2026.
- [38] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training Software Engineering Agents and Verifiers with SWE-Gym. In *International Conference on Machine Learning (ICML)*, 2025.
- [39] Shuai Pan, Yixiang Liu, Jiaye Gao, Te Gao, Weiwen Liu, Jianghao Lin, Zhihui Fu, Jun Wang, Weinan Zhang, and Yong Yu. SkillMAS: Skill Co-Evolution with LLM-based Multi-Agent System. *arXiv:2605.09341*, 2026.
- [40] Liana Patel, Siddharth Jha, Negar Arabzadeh, Carlos Guestrin, Ion Stoica, and Matei Zaharia. What Happens When the Model Eats the Stack? Rethinking the Research Agenda for Data Agents to Withstand the Bitter Lesson. *arXiv:2609.03141*, 2026.
- [41] Hongji Pu, Xinyuan Song, and Liang Zhao. SkillOps: Managing LLM Agent Skill Libraries as Self-Maintaining Software Ecosystems. *arXiv:2605.13716*, 2026.
- [42] Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A Self-Improving Coding Agent. *arXiv:2504.15228*, 2025.
- [43] Swapnanil Saha. Delivery, Not Storage: Cue-Anchored Working Memory as a Harness Property for Coding Agents. *arXiv:2607.20972*, 2026.
- [44] Brian Sam-Bodden. What Context Does a Coding Agent Actually Need to Act? *arXiv:2607.09691*, 2026.
- [45] Fangxin Shang and Yehui Yang. Hypothesis-Driven Skill Optimization for LLM Agents. *arXiv:2606.22330*, 2026.
- [46] Zayx Shawn. PACE: Anytime-Valid Acceptance Tests for Self-Evolving Agents. *arXiv:2606.08106*, 2026.
- [47] Junhao Shen, Teng Zhang, Xiaoyan Zhao, and Hong Cheng. Dynamic Skill Lifecycle Management for Agentic Reinforcement Learning. *arXiv:2605.10923*, 2026.

- [48] Asa Shepard and Jeannie Albrecht. Probe-and-Refine Tuning of Repository Guidance for Coding Agents. *arXiv:2606.20512*, 2026.
- [49] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- [50] Liyan Tang, Cyrus Rashtchian, Chun-Sung Ferng, Andrew Tomkins, Da-Cheng Juan, and Tu Vu. WikiSkill: Compiling Agent Experience into Persistent Knowledge for Skill Evolution. *arXiv:2608.27454*, 2026.
- [51] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models. *Transactions on Machine Learning Research (TMLR)*, 2024.
- [52] Sida Wang. Measuring all the noises of LLM Evals. *arXiv:2512.21326*, 2025.
- [53] Yifan Wang, Hao Cheng, Xiaomin Li, Yuexing Hao, Hemanth Neelgund Ramesh, Dongwon Jung, Hao Tang, Keru Wang, Chenliang Zhou, Qianhui Wu, Wenlin Yao, Ananth Grama, Andrzej Banburski-Fahey, Baolin Peng, Jaron Lanier, and Jianfeng Gao. Certified Long-Horizon Code Agent Evolution via Validation-Gated Skill Optimization. *arXiv:2609.32990*, 2026.
- [54] Yixuan Wang, Yiyang Zhou, Yiming Liang, Congyu Zhang, Fuxiao Liu, Jiawei Zhou, and Huaxiu Yao. Not All Skills Help: Measuring and Repairing Agent Knowledge. *arXiv:2606.15390*, 2026.
- [55] Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent Workflow Memory. In *International Conference on Machine Learning (ICML)*, 2025.
- [56] Xiaojun Wu, Cehao Yang, Honghao Liu, Xueyuan Lin, Wenjie Zhang, Zhichao Shi, Xuhui Jiang, Chengjin Xu, Jia Li, and Jian Guo. Bayesian-Agent: Posterior-Guided Skill Evolution Across LLM Agent Harnesses. *arXiv:2606.08348*, 2026.
- [57] Yifan Wu, Zhuokai Zhao, Songlin Li, Ho Hin Lee, Jiacheng Zhu, Shirley Wu, Tianhe Yu, Serena Li, Lizhu Zhang, Xiangjun Fan, and Shengzhi Li. SWE-Together: Evaluating Coding Agents in Interactive User Sessions. *arXiv:2606.29957*, 2026.
- [58] Yuhao Wu, Jingyuan Zhang, Jiajun Shi, Xinping Lei, Qingshui Gu, Yuxuan Zhang, Zexuan Wang, Chen He, Chen Huang, Maojia Song, Zhiyuan Zeng, Shaowen Wang, Jinkai Liu, Yunfeng Shi, Jiaheng Liu, Shen Yan, Wenhao Huang, Ge Zhang, and Wenxuan Zhang. HarnessDev: Can LLMs Create and Evolve Their Own Agent Harness? *arXiv:2609.01437*, 2026.
- [59] Peng Xia, Rujun Han, Zifeng Wang, Yanfei Chen, Yufan Zhuang, Yoonho Lee, Chengsong Huang, Han Yu, Zhongying CuiZhu, Yifei Ming, Huaxiu Yao, Burak Gokturk, Tomas Pfister, and Chen-Yu Lee. RRSI: Regularized Recursive Self-Improvement of Agent Harnesses. *arXiv:2609.24972*, 2026.
- [60] John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling Data for Software Engineering Agents. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025.
- [61] Yifan Yang, Ziyang Gong, WeiQuan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. SkillOpt: Executive Strategy for Self-Evolving Agent Skills. *arXiv:2605.23904*, 2026.
- [62] Ziyue Yang and Fan Ding. Signal or Noise? A Benchmark Study of Agent Skills in Web Development. *arXiv:2608.23067*, 2026.
- [63] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ -bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. In *International Conference on Learning Representations (ICLR)*, 2025.

- [64] Zhonghao Zhan, Xiao Ma, and Hamed Haddadi. Safe Skill Retirement for Physical Agents. arXiv:2609.29543, 2026.
- [65] Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-Harness: Harnesses That Improve Themselves. arXiv:2606.09498, 2026.
- [66] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents. In *International Conference on Learning Representations (ICLR)*, 2026.
- [67] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. SWE-bench Goes Live! In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025.
- [68] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. In *International Conference on Learning Representations (ICLR)*, 2026.
- [69] Shuyang Zhang. What Does a Skill Actually Do? Estimands and Evaluation Validity for Tool and Skill Use in LLM Agents: A Critical Review. arXiv:2609.33153, 2026.
- [70] Xing Zhang, Yanwei Cui, Guanghui Wang, Ziyuan Li, Wei Qiu, Bing Zhu, and Peiyang He. Ratchet: How Reliable Must an LLM Judge Be to Retire a Skill? arXiv:2605.22148, 2026.
- [71] Xing Zhang, Guanghui Wang, Yanwei Cui, Wei Qiu, Ziyuan Li, Bing Zhu, and Peiyang He. Guardrails Beat Guidance: A Large-Scale Study of Rules, Skills, and Persistent Configuration for Coding Agents. arXiv:2604.11088, 2026.
- [72] Yifan Zhang, Yutong Dai, Juntao Tan, Luyu Yang, Rishi Mullur, Thai Hoang, Zhiyuan Hu, James Zhu, Phil Mui, Silvio Savarese, Ran Xu, and Zeyuan Chen. DarwinX: Evolving Agent Harnesses Through Natural Selection. arXiv:2608.07545, 2026.
- [73] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM Agents Are Experiential Learners. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2024.
- [74] Qiyao Zhao, Xiaofeng Zhang, Bo Liu, Minda Chen, Wei Xiong, Jingyang Chen, Guanting Ye, Wenhao Yu, Xiaosong Yuan, Shijie Han, Da-Han Wang, Jianmin Ji, Fei Huang, and Xu-Yao Zhang. Coalition-Aware Skill Reliability for Self-Evolving Agents. arXiv:2608.22610, 2026.
- [75] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023.
- [76] Pujun Zheng, Zixin Shang, Shufan Jiang, Wenhui Tian, Dongsheng Zhu, Zerun Ma, Dingbo Yuan, and Qi Zhang. SWE-Bench Pro Verified: A Reliable Benchmark for Software Engineering Agents. arXiv:2609.08149, 2026.
- [77] Yujun Zhou, Kehan Guo, Haomin Zhuang, Xiangqi Wang, Yue Huang, Zhenwen Liang, Pin-Yu Chen, Tian Gao, Nuno Moniz, Nitesh V. Chawla, and Xiangliang Zhang. Getting Better at Working With You: Compiling User Corrections into Runtime Enforcement for Coding Agents. arXiv:2606.13174, 2026.
- [78] Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, Sasha Cui, Sayash Kapoor, Shayne Longpre, Kevin Meng, Rebecca Weiss, et al. Establishing Best Practices for Building Rigorous Agentic Benchmarks. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025.

A Further results

This appendix gives the detail behind Section 5: the gate test by kind of candidate (Table 6), every confirm sweep of the loop (Figure 21), every item the loop kept (Table 7), and every held-out scenario under every arm of the evaluation runs (Figure 22).

Table 6: The gate test by kind of candidate. n is the number of candidates of that kind. *Fired* counts the candidates that fired at least once on their screen (a skill opened, a rule loaded). KEEP, KILL and RERUN are `score.sh`’s verdicts, after the confirm sweep for the candidates that reached one; the last column is the verdict the test’s procedure recorded, which stops at the screen when the gain is not positive or the candidate never fired. The harmful rule `small-changes-need-no-changelog` has no paths and loads on every turn.

Candidate	Tier	n	Fired	KEEP	KILL	RERUN	Recorded KILL
placebo, about a topic	always-on skill	10	0	0	7	3	10
placebo, about a moment	always-on skill	10	0	0	5	5	10
harmful	rule	3	3	0	0	3	3
harmful	always-on skill	2	0	0	2	0	2
known-good	rule	1	1	1	0	0	0
known-good	gated skill	1	1	1	0	0	0
known-good	always-on skill	2	0	0	1	1	2

Every confirm sweep of the loop

R1 to R4: the runs each candidate won net, and its worst drop on one task; the mark is the candidate’s final verdict

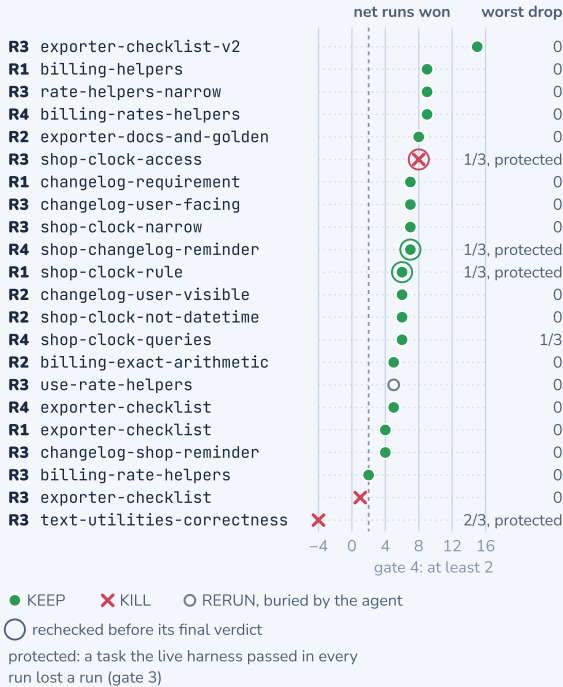


Figure 21: Every confirm sweep of `/evolve` in the 4 evaluation runs, scored again from its rollouts: the runs the candidate won net (kN in Equation (3)) and its worst drop on one task (D), marked by the candidate’s final verdict (KEEP a dot, KILL a cross, RERUN a hollow circle), after a recheck where one ran. The dashed line is gate 4’s threshold. `use-rate-helpers`, whose sweep `score.sh` left unscored and the `/evolve` agent buried, sits among the KEEPs.

Table 7: Every item the loop kept in the 4 evaluation runs. *Chars* is the item’s length; none is always-on, since every item has paths. *Fires* is the share of the run’s evolved benchmark rollouts, held-out and training, on the item’s own family and on the others, in which it fired (a rule loaded, a skill opened): the tier rubric’s third part. *Score* is how many of the rubric’s three criteria the item meets, out of three (Section 5.2). *Held-out* counts the family’s held-out evolved rollouts, on the scenarios valid in every run, in which the item fired.

Run	Fam.	Item	Form	Scope	Chars	Fires (%)		Score	Held-out
						own	other		
R1	A	changelog-requirement	rule	shop/**	518	98	97	3	25/25
R1	B	billing-helpers	rule	shop/billing/**	317	96	21	3	28/30
R1	C	exporter-checklist	skill	shop/plugins/**	1,135	100	0	3	30/30
R1	E	shop-clock-rule	rule	shop/**	327	98	97	3	29/30
R2	A	changelog-user-visible	rule	shop/**	529	98	97	3	25/25
R2	B	billing-exact-arithmetic	rule	shop/billing/**	564	98	18	3	30/30
R2	C	exporter-docs-and-golden	rule	shop/plugins/**	427	100	0	3	30/30
R2	E	shop-clock-not-datetime	rule	shop/**	353	93	98	2	27/30
R3	A	changelog-shop-reminder	rule	shop/**	261	100	99	3	25/25
R3	A	changelog-user-facing	rule	shop/cli.py, shop/formatting.py, shop/report.py	334	100	8	3	25/25
R3	B	billing-rate-helpers	rule	shop/billing/shipping.py, shop/billing/refunds.py	510	13	0	3	0/30
R3	B	rate-helpers-narrow	rule	shop/billing/tax.py, shop/billing/discount.py, shop/billing/fx.py	380	19	0	3	0/30
R3	C	exporter-checklist-v2	rule	shop/plugins/*_export.py	621	100	0	3	30/30
R3	E	shop-clock-narrow	rule	shop/billing/invoice.py, shop/cart.py, shop/orders.py	335	43	28	3	10/30
R4	A	shop-changelog-reminder	rule	shop/**	616	98	98	2	25/25
R4	B	billing-rates-helpers	rule	shop/billing/**	823	100	18	3	30/30
R4	C	exporter-checklist	skill	shop/plugins/**	940	83	0	3	26/30
R4	E	shop-clock-queries	rule	shop/**	574	93	99	2	27/30

Every held-out scenario, in every arm of R1 to R4

pass rate of each held-out scenario in the run that measured the arm; **none** and **evolved** at $k = 5$, the other arms at $k = 3$

	R1		R2		R3		R4		R1, later batch					R2
	none	evolved	none	evolved	none	evolved	none	evolved	none2	kitchen	ideal	flat	desc-only	accept-all
A • changelog														
HA1	0	100	0	80	0	100	0	80	0	67	0	100	100	33
HA2	0	100	0	100	0	100	0	80	0	67	0	100	100	67
HA3	0	100	0	100	0	100	0	100	0	33	0	100	100	100
HA4	0	100	0	100	0	100	0	100	0	67	0	100	100	100
HA5	0	100	0	100	0	100	0	100	0	67	0	100	100	100
HA6	0	100	0	100	0	100	excl.	excl.	0	33	0	100	100	100
B • money														
HB1	0	60	0	100	0	0	0	100	0	100	100	100	100	100
HB2	40	100	0	100	40	40	40	100	0	100	100	100	100	100
HB3	0	80	0	100	0	0	20	100	0	100	100	100	67	100
HB4	20	100	20	100	20	20	0	100	0	100	33	100	67	100
HB5	100	100	100	100	100	100	100	100	100	100	100	100	100	100
HB6	0	100	20	100	0	0	0	100	0	67	100	100	100	100
C • exporter														
HC1	0	40	0	60	0	0	0	20	0	33	33	0	0	33
HC2	0	100	0	60	0	40	0	60	0	67	33	0	0	33
HC3	0	80	0	60	0	20	0	60	0	67	100	0	0	33
HC4	0	100	0	80	0	100	0	80	0	100	67	0	0	67
HC5	0	60	20	80	20	80	0	20	0	100	33	0	67	67
HC6	0	40	0	0	0	0	0	0	0	0	33	0	0	0
E • time														
HE1	100	100	100	100	100	100	80	100	100	100	100	100	100	100
HE2	0	40	20	60	0	0	0	80	0	67	0	100	100	100
HE3	100	100	80	100	80	80	100	100	100	100	100	100	100	100
HE4	40	100	40	100	60	20	20	100	0	100	33	100	100	67
HE5	0	100	0	100	0	0	0	80	0	100	0	100	67	67
HE6	100	100	100	100	100	100	100	100	100	100	100	100	100	100
D • control														
HD1	100	100	100	100	100	100	100	100	100	100	100	100	100	100
HD2	100	100	100	100	100	100	100	100	100	100	100	100	100	100
HD3	100	100	100	100	100	100	100	100	100	100	100	100	100	100
HD4	100	100	100	100	100	100	100	100	100	100	100	100	100	100
HD5	100	100	100	80	100	100	100	100	100	100	100	100	100	100
HD6	100	100	100	100	100	100	100	100	100	100	100	100	100	100

Figure 22: Every held-out scenario under every arm of the 4 evaluation runs: the pass rate of each scenario, in percent, in the run that measured the arm, darker for a higher rate. HA6 was excluded from R4 in every arm (Section 4). R3's money scenarios scored the same under both arms: the run's money rules loaded in none of those rollouts (Section 5.2).