



Agentic Harness Engineering: Observability-Driven Automatic Evolution of Coding-Agent Harnesses

Jiahang Lin^{1*†}, Shichun Liu^{1*†}, Chengjun Pan^{2*†}, Lizhi Lin³, Shihan Dou¹,
Zhiheng Xi¹, Xuanjing Huang¹, Hang Yan³, Zhenhua Han^{3†}, Tao Gui^{1†}, Yu-Gang Jiang¹

¹Fudan University ²Peking University ³Shanghai Qiji Zhifeng Co., Ltd

 [china-qijizhifeng/agentic-harness-engineering](https://github.com/china-qijizhifeng/agentic-harness-engineering)

Abstract

Harnesses are now central to agent performance, mediating how models interact with tools and execution environments. Yet harness engineering remains a manual craft, because automating it faces a heterogeneous action space across editable components, voluminous trajectories that bury actionable signal, and edits whose effect is hard to attribute. We introduce **Agentic Harness Engineering (AHE)**, a closed loop that addresses these challenges through three matched observability pillars: ① *component observability* gives every editable harness component a file-level representation so the action space is explicit and revertible; ② *experience observability* distills millions of raw trajectory tokens into a layered, drill-down evidence corpus that an evolving agent can actually consume; and ③ *decision observability* pairs every edit with a self-declared prediction, later verified against the next round’s task-level outcomes. Together, these pillars turn every edit into a falsifiable contract, so harness evolution proceeds autonomously without collapsing into trial-and-error. Empirically, ten AHE iterations lift pass@1 on Terminal-Bench 2 from 69.7% to 77.0%, surpassing the human-designed harness Codex (71.9%) and the self-evolving baselines ACE and Training-Free GRPO. The frozen harness transfers without re-evolution: on SWE-bench-verified it achieves the highest aggregate success while using 12% fewer tokens than the seed, and on Terminal-Bench 2 it yields +5.1 to +10.1 pp cross-family gains across three alternate model families, indicating the evolved components encode general engineering experience rather than benchmark-specific tuning. Ablations further localize the gain to tools, middleware, and long-term memory rather than the system prompt. These results position observability-driven evolution as a practical pathway to keep coding-agent harnesses continually improving alongside their base models.

1 Introduction

Coding agents are increasingly deployed on long-horizon software-engineering tasks, with measurable progress on issue resolution over real-world code repositories [14, 46, 7] and multi-step terminal workflows [21]. In practice, such progress relies not only on the underlying language model, but equally on the surrounding engineering components: the system prompt that shapes work style, the tools that expose the file system and shell, and the middleware that controls context, execution, and recovery. This collection of model-external, editable components is collectively referred to as the agent’s *harness* [29, 18, 42, 45, 33, 31].

*Equal contributions. †Corresponding authors: hzhua201@gmail.com, tgui@fudan.edu.cn. ‡Work done during an internship at Shanghai Qiji Zhifeng Co., Ltd.

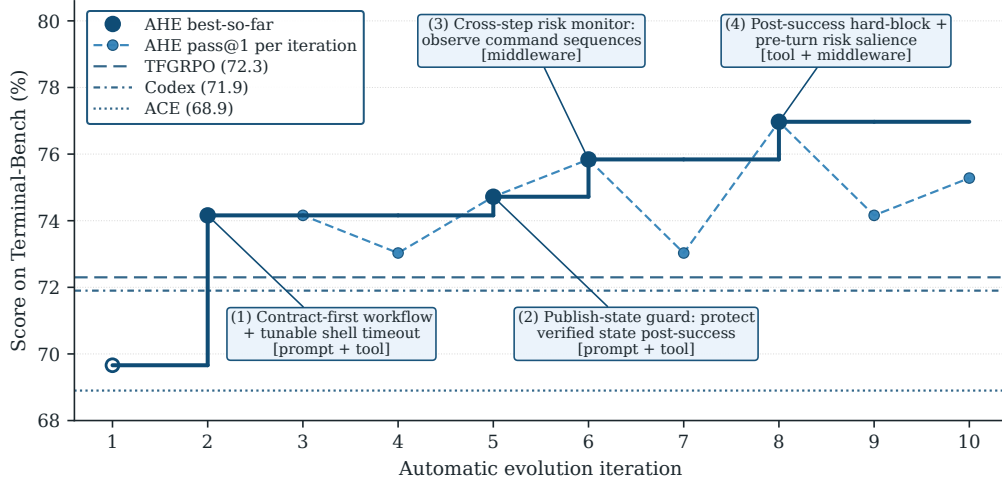


Figure 1: **AHE evolves a bash-only seed past every human-designed and self-evolving baseline on Terminal-Bench 2.** All three role agents share one base model, isolating the gain to harness edits rather than analyzer or editor capability.

Harness design materially shifts task completion on long-horizon coding benchmarks, even with the base model held fixed [40, 42], making harness engineering a first-class lever for improving coding agents. Moreover, the optimal harness is model-specific: a harness tuned for one base model often underperforms on another and must be re-adapted as the base model changes. In current practice, this adaptation is performed manually—developers inspect trajectories, identify recurring failure patterns, and hand-craft edits across prompts, tools, middleware, and skills. Yet as base models advance rapidly [39, 38, 44, 6, 35, 36], this manual loop struggles to keep pace, creating a widening gap between model capability and the harness needed to realize it [33].

An intuitive direction is to automate this loop with an *evolution agent* that optimizes harness components based on experience [1, 49, 4]. However, few existing approaches jointly evolve the full set of editable components [16]; most focus on a single component, typically the prompt [32, 50, 20], skills [19, 43], or an in-context playbook [49]. Jointly evolving multiple components end-to-end faces two structural obstacles: long, unstructured trajectories yield little actionable signal, and tightly coupled harness frameworks make edits beyond the prompt error-prone. This leaves the central question of agent-driven harness evolution open: *How can an evolution agent jointly and stably evolve all editable components of a coding agent’s harness?*

Our central insight is that this question is bottlenecked by *observability*, not by agent capability: once the evolution agent receives structured context over a clear action space, it can reliably converge on better harness designs [34, 53]. We implement this in **Agentic Harness Engineering (AHE, Figure 2)**, a closed loop driven by three observability pillars: ❶ *component observability* via a decoupled harness that exposes seven editable component types as files, so each failure pattern maps cleanly to a single component class; ❷ *experience observability* via a layered, drill-down evidence corpus distilled from millions of raw trajectory tokens, so the evolver consumes structured root causes rather than raw logs; and ❸ *decision observability* via a change manifest that pairs every edit with a self-declared prediction, later verified against the next round’s task-level outcomes, so each edit becomes a falsifiable contract and ineffective ones are reverted at file granularity.

We empirically validate AHE on Terminal-Bench 2[21]: ten iterations lift pass@1 from 69.7% to 77.0%, surpassing the human-designed Codex [25] and the self-evolving baselines ACE [49] and Training-Free GRPO [4]. Without further evolution, the frozen harness transfers to SWE-benchmark-verified [14], and across three alternate base-model families it yields consistent pass@1 gains of +5.1 to +10.1 pp, with the largest on bases further from saturation, suggesting that AHE encodes coordination patterns that less-saturated models lean on more heavily. A component ablation pinpoints where this gain lives: tools, middleware, and long-term memory each carry the improvement on their own, while the system prompt alone regresses, indicating that factual harness structure transfers across tasks and models whereas prose-level strategy does not.

This paper makes three contributions:

- We formulate *agent-driven harness evolution* for coding agents and propose **AHE**, which identifies *observability across components, trajectories, and decisions* as the design pivot and turns every harness edit into a falsifiable, file-level contract through three observability pillars: a decoupled component substrate, a layered trajectory-distillation pipeline, and a change manifest whose self-declared predictions are verified by next-round task deltas.
- We empirically show that AHE lifts pass@1 on Terminal-Bench 2 from 69.7% to 77.0%, surpasses human-designed and automated baselines, and produces a frozen harness that transfers across benchmarks and base-model families.
- Our analysis reveals two limits of agent-driven evolution: harness components interact non-additively, so stacking effective edits caps the aggregate gain; and the loop’s self-attribution is reliable for fixes but blind to regressions, pinpointing regression foresight as the clearest direction for future self-evolution loops.

2 Related Work

2.1 Harness Engineering and Evaluation for Coding Agents

Harness engineering refers to the practice of designing the system surrounding the model, including its tools, interfaces, memory, execution constraints, and feedback loops, which together shape what an agent can do on long-horizon tasks [29, 18, 40, 3, 33, 31]. Concretely, the harness mediates how the model perceives and acts on its environment: it exposes the action and observation interfaces over which tool-augmented reasoning unfolds [3], custom agent-computer interfaces for repository navigation, file editing, and command execution [45], as well as sandboxed execution and orchestration support that keep long-horizon runs reproducible [42].

Verifying that such systems actually help has driven the parallel maturation of coding-agent evaluation along two axes: task horizon and environmental realism. Coverage extends from short-horizon function-level benchmarks focused on contamination and freshness control [52, 12], through repository-scale executable patch resolution [14, 46, 7], to multi-hour, terminal-driven workflows that exercise long-horizon, realistic execution [22, 5, 21]. A parallel infrastructure track packages executable runtimes and verifiers around these benchmarks [28, 13, 47], whose attention to reproducible, traceable, and verifiable execution directly motivates the observation system AHE builds on.

2.2 Automated Optimization of LLM Agents

Approaches to automated agent optimization differ in what evidence the optimizer observes and what it can edit. Some revise the agent’s own outputs through episodic critique and reflection [20, 32, 9]. Others target prompts and instructions [15]: structured playbooks [49], semantic-advantage priors [4], jointly optimized instruction-demonstration pipelines for multi-stage programs [27], and reflective updates driven by Pareto-frontier traces [1]. A separate line edits program structure itself, in the form of skill libraries [41], scored program and agent archives evolved through mutation [24, 11], and graph-structured workflows searched or learned from rollouts [48, 51].

AHE tunes the full harness as a combinatorial whole rather than a single editable surface, so cross-component trade-offs become legible to the optimizer. It also keeps the human prior minimal, leaving methodology for the optimizer to discover from rollouts rather than fixing it by hand. We describe the substrate, trajectory analysis, and iteration that realize these choices in Section 3.

3 Method

AHE turns harness optimization into a closed loop driven by another agent, with the base model held fixed and only the explicit harness edited. Our design principle is that every phase of this loop must be *observable*: AHE faithfully records the artifacts each phase produces (the harness components an iteration writes, the rollout trajectories it generates, the edit decisions it commits) and represents them in structured, layered forms that another agent can read and act on.

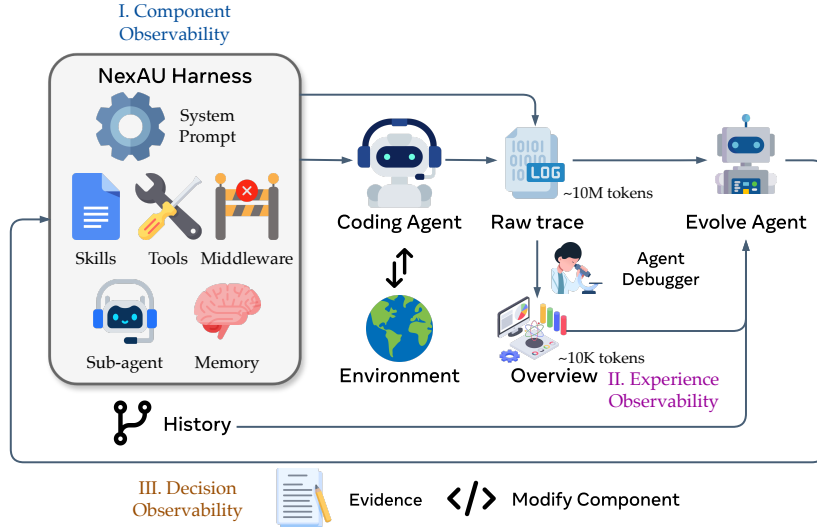


Figure 2: **The AHE pipeline links three observable surfaces into one closed loop.** Components, rollout experience, and edit decisions each surface as structured artifacts another agent reads, and every edit becomes a falsifiable prediction the next round verifies.

Three observability layers implement this principle. **Component observability** (§3.1) is realized by a decoupled, file-level harness substrate that maps each failure pattern to a single component class. **Experience observability** (§3.2) is realized by a layered evidence corpus distilled from raw rollouts and indexed for drill-down access. **Decision observability** (§3.3) is realized by a change manifest that pairs every edit with a self-declared prediction the next round verifies. The three layers compose into the iteration of Algorithm 1, which runs unattended round after round.

3.1 NexAU: an editable, decoupled harness substrate

We instantiate the harness H on the NexAU framework [23, 37], which exposes seven orthogonal component types as explicit files at fixed mount points in a single workspace: system prompt, tool description, tool implementation, middleware, skill, sub-agent configuration, and long-term memory. The component types are loosely coupled, so adding a middleware does not require editing the system prompt, and adding a skill does not require touching any tool.

This decoupling is what realizes **component observability**: each failure pattern maps to a single component class, giving the evolve agent a clean action space and localizing every pass-rate change to one file rather than scattering it across hundreds of lines of unstructured prompt prose. Each logical edit becomes one commit on the workspace’s git history, which yields file-level diffs and rollback granularity for free.

Our seed harness H_0 is deliberately minimal: a single shell-execution tool, no middleware, no skills, no sub-agents. A seed already fitted to the target benchmark would contaminate every subsequent edit’s attribution, since we could not tell whether a gain came from the loop or from the seed. The minimal seed forces every component AHE adds to earn its place against measured rollouts.

3.2 Agent Debugger: layered trajectory evidence

We generate k traces for each task in a benchmark using a harness H , which may contain errors resulting from the deficiencies of the harness that can be acted on, but scattered across millions of tokens of raw messages. To extract insights from agent trajectories and enable **experience observability**, we apply Agent Debugger [17] framework to use an agent to explore trajectories framed as a navigable, file-based environment where each trajectory message lives in its own file and is reached through generic shell and scripting tools. Traces with the same query are placed in one environment, and the debugger is required to analyze the root cause of the failure or the success pattern, which is stored in *per-task analysis* report for each task. The analysis also includes pass/fail

status of the task to ground the Evolve Agent. Finally, a *benchmark-level overview* is aggregated from every report into a single document as an entry point for every iteration.

In addition to these reports, we also provide *original* traces in case the agents need to verify the claims in the reports. The traces are provided both in raw form and lightly processed to remove unnecessary content. All of these content is provided as files allowing *progressive disclosure* [30] which saves on tokens and enable better agent decisions.

3.3 Evolve Agent: evidence-driven, auditable edits

The Evolve Agent closes the AHE loop. In each round it reads the layered evidence corpus produced by the Agent Debugger, decides which harness components to add, modify, or remove, applies those edits to the workspace, and records the reasoning behind every edit. Two constraints govern these edits, and together they realize **decision observability**: every edit becomes a falsifiable, file-level claim recorded in a versioned manifest, and the next round’s verdict either confirms or reverts it.

The first constraint is *controllability*: the Evolve Agent writes only inside the harness workspace, while the runs directory, tracer, verifier, and LLM configuration are read-only, and the seed system prompt (Appendix B.1) is marked non-deletable. These restrictions block the shortcuts an unconstrained self-modifier would take, such as disabling the verifier, swapping the model, or raising the reasoning budget, and keep every recorded gain attributable to harness edits.

The second constraint is that every change is *evidence-driven* and ships with a recorded prediction. Each edit attaches a manifest entry that names the failure evidence, the inferred root cause, the targeted fix, and a predicted impact comprising both expected fixes and at-risk regressions; this manifest is the loop’s evidence ledger (see Appendix B.2). In the next round, the loop intersects the predicted-fix and predicted-regression sets with the observed task-level deltas to produce a per-edit verdict. Each edit thereby becomes falsifiable by the next evaluation, which replaces rationale-driven self-justification with a measurable contract between rounds.

Algorithm 1 AHE outer loop.

Require: seed harness H_0 , base model M , benchmark D , rollouts per task k , max iterations N

```

1:  $H_{\text{best}} \leftarrow H_0$ 
2: for  $t = 1$  to  $N$  do
3:    $T_t \leftarrow \text{ROLLOUT}(M, H_{t-1}, D, k)$  ▷ phase 1:  $k$  rollouts per task
4:    $\tilde{T}_t \leftarrow \text{CLEAN}(T_t)$  ▷ phase 2: normalize traces into canonical form
5:   if  $t \geq 2$  then ▷ phase 3: attribute prior manifest, then rollback
6:      $V_t \leftarrow \text{ATTRIBUTE}(C_{t-1}, T_{t-1}, T_t)$ 
7:      $H_{t-1} \leftarrow \text{ROLLBACK}(H_{t-1}, V_t)$ 
8:   else
9:      $V_t \leftarrow \emptyset$ 
10:  end if
11:   $R_t \leftarrow \text{AGENTDEBUGGER}(\tilde{T}_t)$  ▷ phase 4: layered distillation
12:   $(H_t, C_t) \leftarrow \text{EVOLVE}(H_{t-1}, R_t, V_t)$  ▷ phase 5: workspace edits + new manifest
13:   $\text{COMMIT}(H_t, C_t, t)$  ▷ phase 6: tag iteration in git
14:  if  $\text{PASS@1}(T_t) > \text{PASS@1}(H_{\text{best}})$  then  $H_{\text{best}} \leftarrow H_t$ 
15:  end if
16: end for
17: return  $H_{\text{best}}$ 

```

Algorithm 1 composes the three substrates into one iteration: rollout, clean, attribute the prior manifest and revert rejected edits, distill, edit, commit. We run $k \geq 2$ rollouts per task so each task carries a pass-rate signal, which stabilizes pass@1 and lets partial-pass tasks anchor comparative diagnosis. Attribution runs *before* distillation, so its verdict lands inside the evidence corpus and binds each prior manifest entry as a contract rather than a rationale. A one-shot explore agent (Appendix B.3) runs in parallel with iteration 1 to seed a small number of reusable skills from the NexAU source and public coding-agent references. These skills receive no special protection: from iteration 2 onward the Evolve Agent may keep, refine, or remove them based on observed rollouts.

4 Experiments

We organize our empirical study around three questions: where AHE sits on the map of existing approaches to harness design, whether what it produces is portable beyond its optimization target, and what inside the loop drives the gain.

Research Questions

1. **RQ1 (§4.2): Why agentic harness engineering, rather than human-engineered harnesses or other automated methods?**
2. **RQ2 (§4.3): Does agentic harness engineering overfit to its optimization target?**
3. **RQ3 (§4.4): What inside AHE drives its gains, and how reliable is the loop’s self-attribution?**

4.1 Setup

Evaluation. We drive evolution on the full 89 tasks of Terminal-Bench 2 [21], split as 4 easy, 55 medium, and 30 hard, with per-task timeout extended to 1 hour. For cross-benchmark transfer we evaluate the AHE harness on SWE-bench-verified [14], 500 tasks across seven repositories. We report two metrics per configuration: *pass@1*, the mean binary success rate over k rollouts per task; and *tokens/trial*, the mean per-trial total of prompt plus completion tokens across all LLM calls, in thousands. Infrastructure-aborted or timed-out trials count as failures under *pass@1* (matching the official terminal-bench leaderboard) and are excluded from token means to avoid truncated figures. Runtime infrastructure (framework, dispatcher, sandbox, tracer, and concurrency) is detailed in Appendix A.

Models. For both the evolution loop and the main experiment of §4.2, all three role agents (the Code Agent, the Agent Debugger, and the Evolve Agent) share one base model, GPT-5.4 [26] at the high reasoning setting. For cross-model transfer (§4.3), we re-evaluate the Code Agent on five alternate bases: GPT-5.4 [26] at medium and xhigh reasoning, qwen-3.6-plus [38, 44], gemini-3.1-flash-lite-preview [8], and deepseek-v4-flash [6].

4.2 RQ1: Main Results

We run a single AHE campaign of ten iterations from the bash-only **NexAU₀** seed (§3.1) on Terminal-Bench 2, finishing in roughly 32 hours; the best resulting configuration is reported as **AHE**. The two self-evolve baselines ACE [49] and Training-Free GRPO (TF-GRPO) [4] start from the same NexAU₀ seed.

AHE outperforms both human-designed and self-evolve baselines. AHE outperforms every baseline on our panel: three human-designed harnesses, OpenCode [2], Terminus-2 [10], and Codex [25], and the two self-evolve baselines. Figure 1 shows the gain accumulates across iterations, with continued evolution pushing *pass@1* further above the NexAU₀ seed. By difficulty, the only exception is the Hard tier, where AHE marginally trails Codex. We trace this gap to interference between AHE’s components on long-horizon tasks rather than to a missing capability: swapping AHE’s long-term memory alone into the NexAU₀ seed, without the other AHE components, already surpasses Codex on Hard (§4.4.1).

Prompt-only self-evolution misses the components that carry AHE’s gain. The gaps to ACE and TF-GRPO trace to a layer mismatch. ACE distills natural-language playbooks the agent reads

Table 1: Pass@1 on Terminal-Bench 2 across 89 tasks, by official difficulty. NexAU₀ is the shared seed; ACE, Training-Free GRPO, and **AHE** are three self-evolution loops layered on top of it. Bold marks the best per column; ties are all bold.

Method	All 89	Easy 4	Med. 55	Hard 30
Human-designed harness				
OpenCode	47.2%	75.0%	52.7%	33.3%
Terminus-2	62.9%	75.0%	74.5%	40.0%
Codex	71.9%	75.0%	80.0%	56.7%
Self-evolved from NexAU₀				
NexAU ₀	69.7%	87.5%	78.2%	51.7%
ACE	68.9%	91.7%	78.2%	48.9%
TF-GRPO	72.3%	100.0%	79.4%	55.6%
AHE	77.0%	100.0%	88.2%	53.3%

Table 2: Cross-benchmark transfer on SWE-bench-verified. ACE, Training-Free GRPO (TF-GRPO), and **AHE** share the **NexAU₀** seed and differ only in their self-evolution loop; all four columns run on GPT-5.4. AHE and the two self-evolve baselines are evolved on Terminal-Bench 2 and evaluated without in-domain re-evolution. Per-column bold marks the best; ties are all bold.

Repo	N	Success rate \uparrow				Tokens k \downarrow			
		ACE	TF-GRPO	NexAU ₀	AHE	ACE	TF-GRPO	NexAU ₀	AHE
All	500	74.6%	74.2%	75.2%	75.6%	679	582	526	461
django	231	79.2%	78.8%	79.2%	81.0%	707	583	527	484
sympy	75	69.3%	68.0%	70.7%	70.7%	602	572	494	479
sphinx-doc	44	61.4%	65.9%	68.2%	70.5%	990	848	731	656
matplotlib	34	70.6%	70.6%	73.5%	73.5%	622	530	486	391
scikit-learn	32	93.8%	93.8%	93.8%	87.5%	451	378	307	257
pydata	22	77.3%	77.3%	77.3%	72.7%	563	516	386	338
astropy	22	59.1%	59.1%	54.5%	50.0%	546	470	667	277

in-context, and TF-GRPO is a trajectory-feedback variant of GRPO that reinforces successful tool sequences; neither method opens the surrounding scaffolding to edits. AHE instead jointly evolves system prompt, tools, middleware, and long-term memory, and §4.4.1 shows the gain concentrates in the latter three components, exactly the layers ACE and TF-GRPO leave untouched.

4.3 RQ2: Transfer to Unseen Tasks and Base Models

AHE’s harness is evolved on Terminal-Bench 2 with GPT-5.4 high. We probe whether it encodes general coding-agent experience or overfits to that target by transferring the evolved harness unchanged, without further evolution, to two off-target settings: a different task surface (SWE-bench-verified) and three alternate base models.

Cross-benchmark transfer. We evaluate the AHE harness on SWE-bench-verified against the seed and the two self-evolve baselines (NexAU₀, ACE, TF-GRPO) under identical infrastructure (Table 2).

ACE and TF-GRPO both regress below the NexAU₀ seed in aggregate success while spending 11% to 29% more tokens than the seed: the playbook ACE injects and the trajectory distribution TF-GRPO reinforces were distilled on Terminal-Bench 2 traces and ride the prompt at every model call, so on a different task surface that text adds cost without reshaping the underlying policy.

AHE instead achieves the highest aggregate, with the seed-relative gain concentrating on django and sphinx-doc, the two largest and most token-expensive repositories whose multi-step edit-and-verify loop matches the structure AHE’s tools, middleware, and long-term memory compress on Terminal-Bench 2. Marginal regressions appear only on the three smallest repositories, consistent with pass@1 variance on small repos exceeding the per-repo gain. AHE also cuts aggregate tokens by 32% against ACE, 21% against TF-GRPO, and 12% against the seed: encoding behavior in tools, middleware, and memory avoids the per-call re-derivation cost prompt-only baselines incur.

Cross-model transfer. We re-evaluate both the NexAU₀ seed and AHE on the five alternate bases listed in §4.1. Figure 3 reports five positive pass@1 gains from +2.3 to +10.1 pp.

Cross-family gains dominate within-family ones: deepseek-v4-flash moves +10.1 pp from 51.7% to 61.8%, qwen-3.6-plus +6.3 pp from 56.2% to 62.5%, and gemini-3.1-flash-lite-preview +5.1 pp from 36.5% to 41.6%, all above the +2.3 pp on GPT-5.4 medium and xhigh. We read this as bases further from saturation leaning more on the coordination patterns AHE has fixed inside tools, middleware, and long-term memory, while a stronger base re-derives the same coordination from its prompt at low marginal cost.

Within one family the profile is non-monotone: +2.3 pp on medium, +7.3 pp on high from §4.2, and +2.3 pp on xhigh. AHE’s step budget and per-task timeout were fitted to GPT-5.4 high during evolution; medium has more time-per-step slack but loses a reasoning tier of raw capability, while xhigh pushes more trials past the per-task timeout, which our pass@1 convention (§4.1) counts as failures. Either direction discounts the gain.

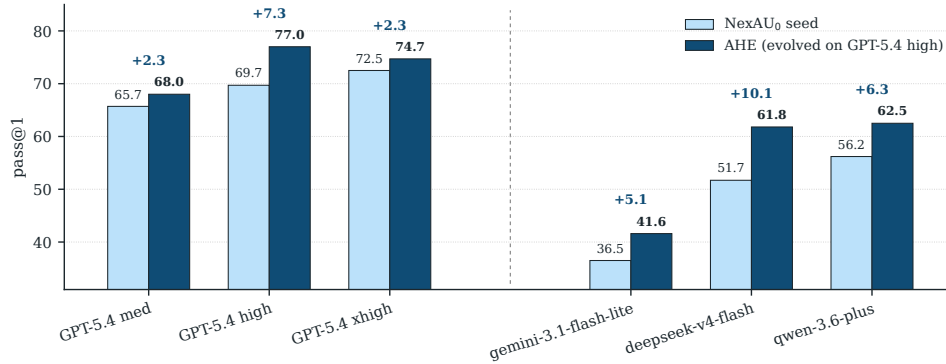


Figure 3: Cross-model transfer on Terminal-Bench 2, 89 tasks. The AHE workspace evolved on GPT-5.4 high is re-evaluated on each base without further evolution, paired against the NexAU₀ seed on the same base. All bases gain over the seed, with cross-family bases gaining more than within-family ones.

The headline finding is that all five gains land positive: the AHE workspace is not specific to one provider’s idioms or one reasoning depth. Their magnitude tracks the evolution operating point rather than raw base capability, so we treat the timeout-budget coupling as a generalization hazard discussed in our [Limitations](#) section.

4.4 RQ3: Analysis

We analyze the loop along two architectural choices that §3 places weight on: decomposed components (§4.4.1) and self-declared attribution (§4.4.2).

4.4.1 RQ3a: where value accumulates across components

Table 3 decomposes the AHE gain at the component level by isolating one of the four evolved layers, long-term memory, tools, middleware, or system prompt, while holding the other three at their seed defaults. Three of the four single-component variants outperform the seed; the system-prompt swap is the only regression.

Table 3: Component-level ablations on Terminal-Bench 2. Each “+ X only” row swaps a single AHE component into the NexAU₀ seed; per-column best is bolded. Most single-component swaps already beat the seed, and the single-component gains overshoot full AHE’s aggregate, indicating components interact non-additively rather than stacking cleanly.

Variant	All 89 tasks	Easy 4 tasks	Medium 55 tasks	Hard 30 tasks
NexAU ₀	69.7%	87.5%	78.2%	51.7%
+ memory only	75.3%	50.0%	83.6%	63.3%
+ tool only	73.0%	75.0%	87.3%	46.7%
+ middleware only	71.9%	100.0%	81.8%	50.0%
+ system_prompt only	67.4%	75.0%	78.2%	46.7%
AHE full	77.0%	100.0%	88.2%	53.3%

Each component owns a different failure surface. Memory adds 12 boundary-case lessons (performance margin, queued-over-limit cancellation, evaluator-style closure, source-packaging layout); on Hard the lessons lift it above full AHE, while on Easy they reduce to superfluous re-verification. Tools become a 1364-line shell that auto-surfaces contract hints from files near each command; on Medium it lands within 0.9 pp of full AHE, while on Hard a built-in publish guard closes the loop too early. Middleware adds a finish-hook that forces one evaluator-isomorphic closure check; on Easy it clears every task, while on Hard it inflates turn count. The system prompt encodes 79 lines of universal discipline whose executability depends on the other three; inserted alone it scores −2.3 pp aggregate.

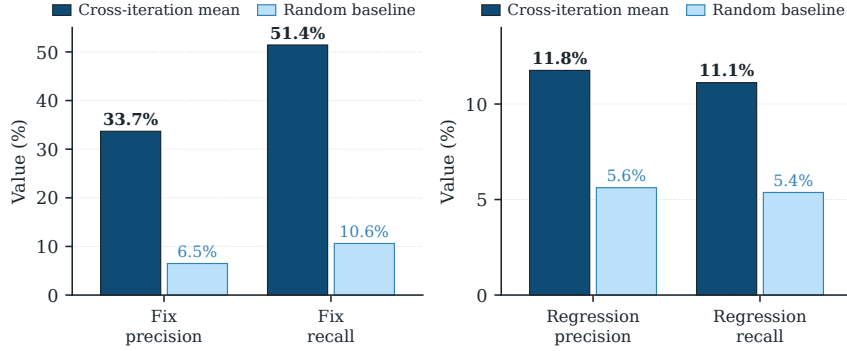


Figure 4: Cross-iteration mean precision and recall of the evolve model’s self-predictions across 9 evaluation rounds of the GPT-5.4 AHE loop on Terminal-Bench 2, alongside the random-prediction baseline. Left: fix predictions. Right: regression predictions.

Components interact non-additively, capping the aggregate gain. The three positive single-component gains sum to +11.1 pp against full AHE’s +7.3 pp, and on Hard the memory-only variant exceeds full AHE: memory, middleware, and the system prompt all push toward the same closure-style verification, so stacking them spends turns on redundant re-checks within the long-horizon budget. Since the evolve agent optimises an aggregate dominated by 55 Medium tasks, it converges to a Medium-heavy trade-off that returns part of the Hard memory effect, and we leave interaction-aware evolution to future work.

4.4.2 RQ3b: how reliably the loop’s self-attribution tracks reality

Each evolution round, our evolve model produces a change manifest naming which Terminal-Bench 2 tasks it expects to fix in the next round and which it flags at risk of regression. We compare the round- $N-1$ prediction against the round- N ground truth, computing standard precision and recall over the 89 tasks separately for fixes and regressions.

Evidence-driven targeting. The fix panel of Figure 4 shows the evolve model’s targeting is evidence-driven rather than guesswork. Cross-iteration fix-precision of **33.7%** and fix-recall of **51.4%** sit roughly 5x above the random-prediction baselines of 6.5% and 10.6%, so each harness edit lands on a real, agent-anticipated target rather than on an arbitrary subset of the panel.

Regression blindness. The regression panel tells the opposite story: cross-iteration regression-precision of **11.8%** and regression-recall of **11.1%** sit only about 2x above their random baselines of 5.6% and 5.4%, so most upcoming regressions go unforeseen. The agent can justify why an edit should help, but it cannot reliably name the tasks the same edit is about to break, which is what produces the non-monotone steps in the evolution curve of §4.2. Closing this gap is the clearest direction for future self-evolution loops. Appendix D gives the per-round breakdown.

5 Conclusion

We introduced **Agentic Harness Engineering (AHE)**, an observability-driven loop that turns a coding agent’s harness into a learnable adaptation surface while the base model remains fixed. AHE exposes components as files, distills rollouts into a layered evidence corpus, and binds each edit to a falsifiable next-round prediction; ten iterations lift pass@1 on Terminal-Bench 2 from 69.7% to 77.0%, and the frozen harness transfers to SWE-bench-verified and three alternate model families. We see harness-level evolution as a complementary axis to model-side training: an externalized, auditable surface where coding-agent experience can accumulate.

Limitations

This work studies a promising but high-variance setting, and the scope of our claims should be interpreted accordingly.

Benchmark scope. Our evaluation drives evolution on Terminal-Bench 2 and probes transfer on SWE-bench-verified. Even though the frozen harness transfers to a second task surface and to three alternate base-model families, broader programming languages, repository-scale deployments, and human-in-the-loop workflows remain untested.

Evolution operating point. AHE’s step budget and per-task timeout were fitted to GPT-5.4 high during evolution, so cross-model transfer numbers conflate harness portability with operating-point coupling—within one family the gain is non-monotone across reasoning tiers (§4.3). Untangling these factors will require re-running the loop under multiple operating points.

Self-modification governance. AHE bounds edits to a workspace, attributes every change in a versioned manifest, and rolls back ineffective edits at file granularity, but it does not provide a complete guardrail stack. Long-horizon harness cleanup and stronger misuse prevention remain incomplete, and AHE should be viewed as a controlled research prototype rather than a fully mature autonomous self-improvement system.

References

- [1] Lakshya A. Agrawal, Shangyin Tan, Dilara Soyulu, Noah Ziemis, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*, October 2025. URL <https://openreview.net/forum?id=RQm2KQTM5r>.
- [2] Anomaly. Opencode: The open source coding agent., 2025. URL <https://github.com/anomalyco/opencode>.
- [3] Anthropic. Claude-code, 2025. URL <https://github.com/anthropics/claude-code>.
- [4] Yuzheng Cai, Siqi Cai, Yuchen Shi, Zihan Xu, Lichao Chen, Yulei Qin, Xiaoyu Tan, Gang Li, Zongyi Li, Haojia Lin, Yong Mao, Ke Li, and Xing Sun. Training-free group relative policy optimization, October 2025. URL <http://arxiv.org/abs/2510.08191>.
- [5] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. Mle-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations*, October 2024. URL <https://openreview.net/forum?id=6s5uXNWGIh>.
- [6] DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, April 2026. URL https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro/blob/main/DeepSeek_V4.pdf.
- [7] Xiang Deng, Jeff Da, Edwin Pan, Yannis Y. He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa R. Kundurthy, Sean M. Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? October 2025. URL <https://openreview.net/forum?id=9R2iUHhVfr>.
- [8] Google. Gemini-3-1-flash-lite-model-card, March 2026. URL <https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-1-Flash-Lite-Model-Card.pdf>.
- [9] Honglin Guo, Kai Lv, Qipeng Guo, Tianyi Liang, Zhiheng Xi, Demin Song, Qiuyinzhe Zhang, Yu Sun, Kai Chen, Xipeng Qiu, and Tao Gui. Critiq: Mining data quality criteria from human preferences. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16240–16261, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.792. URL <https://aclanthology.org/2025.acl-long.792/>.

- [10] Harbor. Terminus-2, 2026. URL <https://www.harborframework.com/docs/agents/terminus-2>.
- [11] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, October 2024. URL <https://openreview.net/forum?id=t9U3LW7JVX>.
- [12] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, October 2024. URL <https://openreview.net/forum?id=chfJJYC3iL>.
- [13] Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environment generation and hybrid verifiers for scaling open-weights swe agents. In *Second Conference on Language Modeling*, August 2025. URL <https://openreview.net/forum?id=7evvwdo3z#discussion>.
- [14] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, October 2023. URL <https://openreview.net/forum?id=VTF8yNQM66>.
- [15] 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 self-improving pipelines, October 2023. URL <http://arxiv.org/abs/2310.03714>.
- [16] Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses, March 2026. URL <http://arxiv.org/abs/2603.28052>.
- [17] Lizhi Lin. Agent debugger: Understanding agent trajectory with agentic workflows - dawning road, February 2026. URL <https://dawning-road.github.io/blog/agent-debugger>.
- [18] Ryan Lopopolo. Harness engineering: Leveraging codex in an agent-first world, February 2026. URL <https://openai.com/zh-Hans-CN/index/harness-engineering/>.
- [19] Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. Skillclaw: Let skills evolve collectively with agentic evolver, April 2026. URL <http://arxiv.org/abs/2604.08377>.
- [20] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Thirty-Seventh Conference on Neural Information Processing Systems*, November 2023. URL <https://openreview.net/forum?id=S37h0erQLB>.
- [21] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu,

- Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, January 2026. URL <http://arxiv.org/abs/2601.11868>.
- [22] Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. Swe-lancer: Can frontier llms earn \$1 million from real-world freelance software engineering? In *Forty-Second International Conference on Machine Learning*, June 2025. URL <https://openreview.net/forum?id=xZXhFg43EI>.
- [23] Nex-AGI. Nexau (au for agent universe), a general-purpose agent framework for building intelligent agents with tool capabilities., 2025. URL <https://github.com/nex-agi/NexAU>.
- [24] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, June 2025. URL <http://arxiv.org/abs/2506.13131>.
- [25] OpenAI. Codex cli, 2025. URL <https://developers.openai.com/codex/cli>.
- [26] OpenAI. Introducing gpt-5.4, March 2026. URL <https://openai.com/index/introducing-gpt-5-4/>.
- [27] Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 9340–9366, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.525. URL <https://aclanthology.org/2024.emnlp-main.525/>.
- [28] 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 *Forty-Second International Conference on Machine Learning*, June 2025. URL <https://openreview.net/forum?id=Cq1BNvHx74>.
- [29] Prithvi Rajasekaran. Harness design for long-running application development, March 2026. URL <https://www.anthropic.com/engineering/harness-design-long-running-apps>.
- [30] Prithvi Rajasekaran, Ethan Dixon, Carly Ryan, Jeremy Hadfield, Rafi Ayub, Hannah Moran, Cal Rueb, Connor Jennings, Molly Vorwerck, Stuart Ritchie, and Maggie Vo. Effective context engineering for ai agents, September 2025. URL <https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents>.
- [31] Nous Research. Hermes agent — the agent that grows with you, 2026. URL <https://hermes-agent.nousresearch.com/>.
- [32] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R. Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-Seventh Conference on Neural Information Processing Systems*, November 2023. URL <https://openreview.net/forum?id=vAElhFcKW6>.
- [33] Peter Steinberger. Openclaw — personal ai assistant, February 2026. URL <https://openclaw.ai/>.
- [34] Rich Sutton. The bitter lesson, March 2019. URL https://www.cs.utexas.edu/~eunsol/courses/data/bitter_lesson.pdf.
- [35] Kimi Team. Kimi k2.6 tech blog: Advancing open-source coding, April 2026. URL <https://www.kimi.com/blog/kimi-k2-6>.

- [36] Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, S. H. Cai, Yuan Cao, Y. Charles, H. S. Che, Cheng Chen, Guanduo Chen, Huarong Chen, Jia Chen, Jiahao Chen, Jianlong Chen, Jun Chen, Kefan Chen, Liang Chen, Ruijue Chen, Xinhao Chen, Yanru Chen, Yanxu Chen, Yicun Chen, Yimin Chen, Yingjiang Chen, Yuankun Chen, Yujie Chen, Yutian Chen, Zhirong Chen, Ziwei Chen, Dazhi Cheng, Minghan Chu, Jialei Cui, Jiaqi Deng, Muxi Diao, Hao Ding, Mengfan Dong, Mengnan Dong, Yuxin Dong, Yuhao Dong, Angang Du, Chenzhuang Du, Dikang Du, Lingxiao Du, Yulun Du, Yu Fan, Shengjun Fang, Qiulin Feng, Yichen Feng, Garimugai Fu, Kelin Fu, Hongcheng Gao, Tong Gao, Yuyao Ge, Shangyi Geng, Chengyang Gong, Xiaochen Gong, Zhuoma Gongque, Qizheng Gu, Xinran Gu, Yicheng Gu, Longyu Guan, Yuanying Guo, Xiaoru Hao, Weiran He, Wenyang He, Yunjia He, Chao Hong, Hao Hu, Jiayi Hu, Yangyang Hu, Zhenxing Hu, Ke Huang, Ruiyuan Huang, Weixiao Huang, Zhiqi Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yu Jing, Guokun Lai, Aidi Li, C. Li, Cheng Li, Fang Li, Guanghe Li, Guanyu Li, Haitao Li, Haoyang Li, Jia Li, Jingwei Li, Junxiong Li, Lincan Li, Mo Li, Weihong Li, Wentao Li, Xinhang Li, Xinhao Li, Yang Li, Yanhao Li, Yiwei Li, Yuxiao Li, Zhaowei Li, Zheming Li, Weilong Liao, Jiawei Lin, Xiaohan Lin, Zhishan Lin, Zichao Lin, Cheng Liu, Chenyu Liu, Hongzhang Liu, Liang Liu, Shaowei Liu, Shudong Liu, Shuran Liu, Tianwei Liu, Tianyu Liu, Weizhou Liu, Xiangyan Liu, Yangyang Liu, Yanming Liu, Yibo Liu, Yuanxin Liu, Yue Liu, Zhengying Liu, Zhongnuo Liu, Enzhe Lu, Haoyu Lu, Zhiyuan Lu, Junyu Luo, Tongxu Luo, Yashuo Luo, Long Ma, Yingwei Ma, Shaoguang Mao, Yuan Mei, Xin Men, Fanqing Meng, Zhiyong Meng, Yibo Miao, Mingqing Ni, Kun Ouyang, Siyuan Pan, Bo Pang, Yuchao Qian, Ruoyu Qin, Zeyu Qin, Jiezhong Qiu, Bowen Qu, Zeyu Shang, Youbo Shao, Tianxiao Shen, Zhennan Shen, Juanfeng Shi, Lidong Shi, Shengyuan Shi, Feifan Song, Pengwei Song, Tianhui Song, Xiaoxi Song, Hongjin Su, Jianlin Su, Zhaochen Su, Lin Sui, Jinsong Sun, Junyao Sun, Tongyu Sun, Flood Sung, Yunpeng Tai, Chuning Tang, Heyi Tang, Xiaojuan Tang, Zhengyang Tang, Jiawen Tao, Shiyuan Teng, Chaoran Tian, Pengfei Tian, Ao Wang, Bowen Wang, Chensi Wang, Chuang Wang, Congcong Wang, Dingkun Wang, Dinglu Wang, Dongliang Wang, Feng Wang, Hailong Wang, Haiming Wang, Hengzhi Wang, Huaqing Wang, Hui Wang, Jiahao Wang, Jinhong Wang, Jiuzheng Wang, Kaixin Wang, Linian Wang, Qibin Wang, Shengjie Wang, Shuyi Wang, Si Wang, Wei Wang, Xiaochen Wang, Xinyuan Wang, Yao Wang, Yejie Wang, Yipu Wang, Yiqin Wang, Yucheng Wang, Yuzhi Wang, Zhaoji Wang, Zhaowei Wang, Zhengtao Wang, Zhexu Wang, Zihan Wang, Zizhe Wang, Chu Wei, Ming Wei, Chuan Wen, Zichen Wen, Chengjie Wu, Haoning Wu, Junyan Wu, Rucong Wu, Wenhao Wu, Yuefeng Wu, Yuhao Wu, Yuxin Wu, Zijian Wu, Chenjun Xiao, Jin Xie, Xiaotong Xie, Yuchong Xie, Yifei Xin, Bowei Xing, Boyu Xu, Jianfan Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinbo Xu, Xinran Xu, Yangchuan Xu, Yichang Xu, Yuemeng Xu, Zelai Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Guangyao Yang, Hao Yang, Junwei Yang, Kai Yang, Ningyuan Yang, Ruihan Yang, Xiaofei Yang, Xinlong Yang, Ying Yang, Yi Yang, Yi Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Dan Ye, Wenjie Ye, Zhuorui Ye, Bohong Yin, Chengzhen Yu, Longhui Yu, Tao Yu, Tianxiang Yu, Enming Yuan, Mengjie Yuan, Xiaokun Yuan, Yang Yue, Weihao Zeng, Dunyuan Zha, Haobing Zhan, Dehao Zhang, Hao Zhang, Jin Zhang, Puqi Zhang, Qiao Zhang, Rui Zhang, Xiaobin Zhang, Y. Zhang, Yadong Zhang, Yangkun Zhang, Yichi Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yushun Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Chenguang Zhao, Feifan Zhao, Jinxiang Zhao, Shuai Zhao, Xiangyu Zhao, Yikai Zhao, Zijia Zhao, Huabin Zheng, Ruihan Zheng, Shaojie Zheng, Tengyang Zheng, Junfeng Zhong, Longguang Zhong, Weiming Zhong, M. Zhou, Runjie Zhou, Xinyu Zhou, Zaida Zhou, Jinguo Zhu, Liya Zhu, Xinhao Zhu, Yuxuan Zhu, Zhen Zhu, Jingze Zhuang, Weiyu Zhuang, Ying Zou, and Xinxing Zu. Kimi k2.5: Visual agentic intelligence, February 2026. URL <http://arxiv.org/abs/2602.02276>.
- [37] Nex-AGI Team, Yuxuan Cai, Lu Chen, Qiaoling Chen, Yuyang Ding, Liwen Fan, Wenjie Fu, Yufei Gao, Honglin Guo, Pinxue Guo, Zhenhua Han, Zhengfu He, Hanglei Hu, Kai Hu, Shengjia Hua, Tianyu Huai, Baodai Huang, Li Ji, Zhen Jiang, Zhikai Lei, Bufan Li, Jiahang Lin, Lizhi Lin, Jinxiu Liu, Shichun Liu, Ziming Liu, Yuchen Ni, Pengfang Qian, Yujiong Shen, Qingyun Shi, Wentao Shu, Peng Sun, Yiran Suo, Tian Tang, Boyu Tian, Guoteng Wang, Junzhe Wang, Peixin Wang, Zhiheng Xi, Hang Yan, Jie Yang, Zhixiong Yang, Tianchu Yao, Guangze Ye, Qianxi Yu, Shuo Zhang, Xinyue Zhang, Yiqi Zhang, Jiarong Zhao, Miao Zheng, Rui Zheng, Enyu Zhou, Jiazheng Zhou, Maosen Zhou, Yuhao Zhou, Tao Gui, Yining Zheng, Xinchu Chen, Jie Zhou, Siyuan Feng, Qin Chen, Liang He, Qi Zhang, Xuanjing Huang, and Xipeng Qiu. Nex-n1: Agentic models trained via a unified ecosystem for large-scale environment construction,

- December 2025. URL <http://arxiv.org/abs/2512.04987>.
- [38] Qwen Team. Qwen3.6-plus: Towards real world agents, April 2026. URL <https://qwenlm.github.io/blog/qwen3.6/>.
- [39] Xiaomi MiMo Team. Mimo-v2.5-pro, April 2026. URL <https://huggingface.co/XiaomiMiMo/MiMo-V2.5-Pro>.
- [40] Vivek Trivedy. Improving deep agents with harness engineering, February 2026. URL <https://www.langchain.com/blog/improving-deep-agents-with-harness-engineering>.
- [41] 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, October 2023. URL <http://arxiv.org/abs/2305.16291>.
- [42] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents, April 2025. URL <http://arxiv.org/abs/2407.16741>.
- [43] Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning, February 2026. URL <http://arxiv.org/abs/2602.08234>.
- [44] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, May 2025. URL <http://arxiv.org/abs/2505.09388>.
- [45] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-Eighth Annual Conference on Neural Information Processing Systems*, November 2024. URL [https://openreview.net/forum?id=mXpq6ut8J3&referrer=%5Bthe%20profile%20of%20Shunyu%20Yao%5D\(%2Fprofile%3Fid%3D~Shunyu_Yao1\)](https://openreview.net/forum?id=mXpq6ut8J3&referrer=%5Bthe%20profile%20of%20Shunyu%20Yao%5D(%2Fprofile%3Fid%3D~Shunyu_Yao1)).
- [46] John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida Wang, and Ofir Press. Swe-bench multimodal: Do ai systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations*, October 2024. URL <https://openreview.net/forum?id=riTiq3i21b>.
- [47] Yucheng Zeng, Shupeng Li, Daxiang Dong, Ruijie Xu, Zimo Chen, Liwei Zheng, Yuxuan Li, Zhe Zhou, Haotian Zhao, Lun Tian, Heng Xiao, Tianshu Zhu, Longkun Hao, and Jianmin Wu. Swe-hub: A unified production system for scalable, executable software engineering tasks, February 2026. URL <http://arxiv.org/abs/2603.00575>.
- [48] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. Aflow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, October 2024. URL <https://openreview.net/forum?id=z5uVAKwmjf>.

- [49] 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 *The Fourteenth International Conference on Learning Representations*, October 2025. URL <https://openreview.net/forum?id=eC4ygDs02R>.
- [50] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners, December 2024. URL <http://arxiv.org/abs/2308.10144>.
- [51] Wangchunshu Zhou, Yixin Ou, Shengwei Ding, Long Li, Jialong Wu, Tiannan Wang, Jiamin Chen, Shuai Wang, Xiaohua Xu, Ningyu Zhang, Huajun Chen, and Yuchen Eleanor Jiang. Symbolic learning enables self-evolving agents, June 2024. URL <http://arxiv.org/abs/2406.18532>.
- [52] Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*, October 2024. URL <https://openreview.net/forum?id=YrycTj11L0>.
- [53] Gregor Zunic. The bitter lesson of agent harnesses, April 2026. URL <https://browser-use.com/posts/bitter-lesson-agent-harnesses>.

A Experimental Setup: Full Details

This appendix expands the condensed Setup in §4.1 with the formal metric definitions and the runtime infrastructure.

Seed agent. The seed configuration, denoted NexAU_0 , is a simple code agent built on the NexAU framework of §3.1 that exposes only the bash tool to the model, with no skills, no middleware, and no long-term memory. Every iteration of the AHE outer loop edits this workspace, so all reported gains are measured against NexAU_0 as the common starting point.

Runtime infrastructure. All runs use the NexAU framework of §3.1 to instantiate the coding agent. Harbor dispatches tasks, isolates each rollout, and verifies pass/fail, with a 3600-second per-task timeout. Every rollout runs inside a fresh E2B remote sandbox, so shell side-effects cannot leak between tasks. InMemoryTracer records trajectories and mirrors them to Langfuse. The Agent Debugger executes at concurrency 16 with a 600-second per-task timeout.

Hyperparameters. Table 4 consolidates the operating points of the four agents and the outer loop, taken from the configuration snapshot of the reference run.

Terminal-bench difficulty labels. The official terminal-bench-2 leaderboard⁰ partitions the 89-task subset into 4 easy, 55 medium, and 30 hard tasks.

pass@1. For a configuration on a task set D with k rollouts per task, let $r_{i,j} \in \{0, 1\}$ denote the binary reward of rollout j on task i . The pass@1 score is the mean

$$\text{pass@1} = \frac{1}{k|D|} \sum_{i=1}^{|D|} \sum_{j=1}^k r_{i,j}. \quad (1)$$

Trials that terminate on an infrastructure exception, such as a sandbox crash or API timeout, contribute $r = 0$ rather than being dropped, a strictly harsher convention than discarding failures that keeps our numbers comparable to the official terminal-bench leaderboard. The rollout count k varies across experiments; each table states it explicitly.

⁰<https://www.tbench.ai/benchmarks/terminal-bench-2>

Table 4: Hyperparameters of the reference AHE run on Terminal-Bench 2, taken from the configuration snapshot of the run that produced the AHE numbers in Section 4.2. All four agents share GPT-5.4 as the backing model and differ only in reasoning tier, sampling, and per-component limits.

Block	Hyperparameter	Value
Code Agent	Backing model	GPT-5.4
	Reasoning effort	high
	Temperature	0.7
	Top- p	0.95
	Max context	200,000 tokens
	Max generation per turn	32,000 tokens
	Max model turns	300
Agent Debugger	Backing model	GPT-5.4
	Concurrent tasks	16
	Per-task timeout	600 s
	Retry attempts	3
	Retry backoff	2 s
Evolve Agent	Backing model	GPT-5.4
	Reasoning effort	xhigh
	Temperature	0.7
	Max context	200,000 tokens
	Max generation per turn	32,000 tokens
	Max model turns	500
	Skill packages	3
	Context compaction threshold	0.75
Explore Agent	Backing model	GPT-5.4
	Reasoning effort	xhigh
	Wall-clock budget	60 min
	Web sources	10
	Code sources	1
Outer loop / dispatcher	Iterations T	10
	Rollouts per task k	2
	Concurrent rollouts	96
	Sandbox	E2B remote
	Sandbox lifetime	3,600 s
	Dataset	Terminal-Bench 2, 89 tasks

Token cost and Succ/Mtok. For token cost we count every LLM call as prompt plus completion across the rollout and report the mean over completed trials in thousands, denoted Tokens k ; infrastructure-aborted trials are excluded to avoid truncated figures. To compare configurations that trade accuracy for cost we combine the two via

$$\text{Succ/Mtok} = \frac{\text{pass}@1 \times 10^6}{\text{mean tokens per trial}}, \quad (2)$$

the expected number of successes per million tokens. The main paper reports pass@1 and Tokens k separately so each axis stays legible; Table 5 folds them into Succ/Mtok per repository on SWE-bench-verified, derived from the pass@1 and Tokens k columns of Table 2.

B Prompts and Configurations

This appendix gathers the prompts that drive the AHE outer loop together with the seed code agent’s system prompt. The five blocks below reproduce the literal contents of the corresponding files in the public repository at <https://github.com/china-qijizhifeng/agent-harness-engineering> as of the commit that produced the experiments in Section 4. Jinja-style `{{ var }}` placeholders are filled in by the harness at runtime.

Table 5: Cost-efficiency on SWE-bench-verified, reported as Succ/Mtok, the expected successes per million tokens. Values are derived from Table 2 as $\text{pass}@1 \times 10^3 / \text{Tokens k}$. Higher is better. Per-row bold marks the best.

Repo	N	ACE	TF-GRPO	NexAU ₀	AHE
All	500	1.10	1.27	1.43	1.64
django	231	1.12	1.35	1.50	1.67
sympy	75	1.15	1.19	1.43	1.48
sphinx-doc	44	0.62	0.78	0.93	1.07
matplotlib	34	1.14	1.33	1.51	1.88
scikit-learn	32	2.08	2.48	3.06	3.40
pydata	22	1.37	1.50	2.00	2.15
astropy	22	1.08	1.26	0.82	1.81

B.1 Code Agent Seed System Prompt

The seed system prompt loaded into NexAU₀ at iteration 1. It is intentionally minimal: a single tool, three behavioral rules, and three runtime-injected variables. Every iteration after iteration 1 may append rules to this file, and the case study in Appendix C traces the first such append.

```
code_agent_simple/systemprompt.md

You solve software tasks in a non-interactive setting. Your only tool is run_shell_command: use the shell to inspect the repo, edit files, run builds/tests, and finish the work. Do not ask the user questions.

- Prefer short replies; use the tool for actions.
- Before commands that delete or overwrite important data, state briefly what they do.
- Long-running processes: use `is_background: true` on `run_shell_command` (do not use `&` in the command string).

Date: {{ date }}
Username: {{ username }}
Working Dir: {{ working_directory }}
```

B.2 Evolve Agent Prompt

The Evolve Agent’s system prompt encodes the three hard contracts described in Section 3: workspace-only controllability, evidence-driven changes, and the change-manifest deliverable. It also embeds the directory layout the agent must reason over and the JSON shape of the manifest.

```
evolve_agent/evolve_prompt.md

{% set ws = workspace_path if workspace_path is defined else "workspace" %}
You are the NexAU Evolution Engine -- a meta-agent that iterates on a coding agent's harness to maximize pass@1 (single-attempt success rate) through evidence-based experimentation. You may modify existing components or create new ones (tools, middleware, skills, sub-agents, etc.) as needed.

# Core Principles

## 1. Controllability

Only `workspace/` is your playground. Everything else is read-only or off-limits.

- Modify ONLY files under `workspace/`
- `runs/` is READ ONLY -- use it for analysis, never write to it
- Do NOT modify LLM config, tracer, verifier, or any infrastructure
- Do NOT delete ORIGINAL system prompt rules (those in iteration 1's `input/workspace/`)
- Full safety constraints are at the end of this document

## 2. Evidence-Driven

Every change must be traceable to specific failure evidence. Do not make changes based on intuition, speculation, or "best practices" alone.

Before making any change, you must have:
```

```

1. Failure evidence -- which tasks failed, and what specifically went wrong (from analysis reports
   or traces)
2. Root cause -- why it failed, not just what failed
3. Targeted fix -- a change that directly addresses the root cause
4. Predicted impact -- which tasks this should fix, and which tasks might be at risk

# Environment

{% if ws != "workspace" %}
> WORKSPACE PATH: Your workspace is at `{{ ws }}` instead of `workspace`. All `workspace/`
  references below apply to `{{ ws }}`. Use `{{ ws }}` in file operations, git commands, and the
  validation command.
{% endif %}

> Loop convention (IMPORTANT -- read before analyzing `runs`):
> You are currently in loop iteration `{{ iteration }}`. Each `runs/iteration_NNN` folder mixes two
  generations of work:
> - `input` holds what the previous loop (NNN-1) produced -- this is the workspace that was just
  evaluated this loop. The benchmark, analysis, and change_evaluation inside `input` all describe
  the previous loop's changes, not yours.
> - `evolve` holds what this loop (NNN) will produce -- your new changes, which the next loop (
  NNN+1) will evaluate.
>
> Concretely: when your query says "Iteration {{ iteration }} evaluation completed", it means the eval
  of iteration {{ iteration - 1 }}'s changes is done (baseline if `{{ iteration }}` = 1). You
  are now making changes that will be labeled iteration `{{ iteration }}` and evaluated next loop.

...
./
|-- {{ ws }}/
| |-- code_agent.yaml          # Agent config (tools, middleware, params, sub-agents)
| |-- systemprompt.md         # System prompt (Jinja template)
| |-- LongTermMEMORY.md       # Long-term memory (persistent cross-session knowledge,
  MODIFIABLE)
| |-- ShortTermMEMORY.md      # Short-term memory (managed by code agent at runtime, DO NOT
  MODIFY)
| |-- tool_descriptions/      # Tool YAML definitions
| |-- tools/                  # Tool Python implementations
| |-- middleware/             # Middleware Python implementations
| |-- skills/                 # Skill packages
| |-- sub_agents/             # Sub-agent configs (optional, you may create)
|
|-- runs/                      # * READ ONLY
| |-- iteration_NNN/
| | |-- input/                 # Everything this iteration starts with
| | | |-- workspace/           # Workspace being evaluated this loop
| | | |-- benchmark/          # Eval results for the workspace above
| | | | |-- {timestamp}/
| | | | | |-- result.json
| | | | | | |-- {task_name}__id/
| | | | | | | |-- agent/nexau.txt
| | | | | | | |-- agent/nexau_in_memory_tracer.cleaned.json
| | | | | | | |-- verifier/reward.txt
| | | |-- analysis/           # ** Pre-built failure/success analysis (READ THIS FIRST)
| | | | |-- overview.md
| | | | | |-- detail/{task_name}.md
| | | |-- variant_selection.json
| | | |-- change_evaluation.json
| | | |-- evolve/             # YOUR outputs this loop
| | | | |-- evolve_summary.md
| | | | |-- change_manifest.json
| | | | |-- variant_N/
| | | | | |-- workspace/
| | | | | |-- evolve_trace.json
|
|-- evolution_history.md       # Cumulative history of all iterations (READ)
|-- config_snapshot.yaml      # Initial config (READ ONLY)
...

# Components

## Available Component Types

| Component | Files | Characteristics | When to use |
|-----|-----|-----|-----|
| System Prompt | `workspace/systemprompt.md` | Advisory -- applies to all tasks | Behavioral
  rules, workflow guidance |

```

```

| **Tool Description** | `workspace/tool_descriptions/*.tool.yaml` | Co-located with tool -- model
  reads when calling | Clarify tool usage, add examples, warn about pitfalls |
| **Tool Implementation** | `workspace/tools/` | Controls tool behavior directly | New capabilities,
  smarter error handling, output formatting |
| **Middleware** | `workspace/middleware/` + `code_agent.yaml` | Hooks into agent loop pipeline |
  Intercept/transform at execution level |
| **Skill** | `workspace/skills/` + `code_agent.yaml` | On-demand -- loaded when relevant | Reusable
  workflow patterns |
| **Sub-Agent** | `workspace/sub_agents/{name}/` + `code_agent.yaml` | Delegated execution -- isolated
  context | Offload specialized subtask to child agent |
| **Long-Term Memory** | `workspace/LongTermMEMORY.md` | Persistent cross-session knowledge --
  MODIFIABLE | Record recurring pitfalls, proven strategies, environment quirks |
| **Short-Term Memory** | `workspace/ShortTermMEMORY.md` | Session-scoped scratch -- DO NOT MODIFY | _
  (read-only for evolve agent)_ |

All component types are equally valid and important. Choose the one that best fits the root cause.

### Choosing the Right Component Level

For each failure pattern, consider **all** component types above -- including creating new ones --
before deciding where to fix.

**Anti-pattern:** If the same failure class persists across 2+ iterations despite fixes at one
component level, that level may be the wrong choice. Rollback the ineffective change and re-
approach from a different component level.

## Registering New Components

**Creating a file is NOT enough -- register in `code_agent.yaml`:**
- New tool: create `*.tool.yaml` + Python implementation + add entry to `tools:` list
- New middleware: create Python class + add entry to `middlewares:` list with `import:` path and `
  params:`
- New skill: create `skills/{name}/SKILL.md` folder + add to `skills:` list
- New sub-agent: create `sub_agents/{name}/agent.yaml` + add to `sub_agents:` list. Framework **auto-
  injects** `RecallSubAgent` tool -- do NOT add it manually.

## How Code Gets Loaded

The config directory is added to `sys.path` at runtime:
- `binding: tools.file_tools:read_file` resolves to `workspace/tools/file_tools/read_file.py`
- `import: middleware.long_tool_output:LongToolOutputMiddleware` resolves to `workspace/middleware/
  long_tool_output.py`
- `import: middleware.context_compaction:ContextCompactionMiddleware` resolves to `workspace/
  middleware/context_compaction/__init__.py`

## LLM Environment Variables

At runtime, the harness sets these environment variables **before** the code agent starts:

| Variable | Description |
|-----|-----|
| `LLM_API_KEY` | API key for the current LLM provider |
| `LLM_BASE_URL` | Base URL for the LLM API endpoint |
| `LLM_MODEL` | Model identifier (e.g. `gpt-5.4`) |

**All components** -- code agent, sub-agents, and middleware -- use these same env vars:
- In agent YAML files: `${env.LLM_API_KEY}`, `${env.LLM_BASE_URL}`, `${env.LLM_MODEL}`
- In middleware Python code: `os.environ["LLM_API_KEY"]`, etc.

**Do NOT hardcode API keys.** Always reference environment variables.

### Middleware can call LLM

Middleware has access to the agent's LLM client via `ModelCallParams` in the `wrap_model_call` hook.
Use `LLMCaller` to make side-calls (e.g. summarize context, classify errors, generate dynamic
guidance). See the evolution guide skill for full API reference and examples.

### Sub-Agents use the same LLM

Sub-agent YAML configs should use `${env.LLM_MODEL}` / `${env.LLM_BASE_URL}` / `${env.LLM_API_KEY}` in
their `llm_config`. This automatically gives them the same LLM provider as the parent agent.

For detailed schemas, creation guides, and code examples, read `evolve_agent/skills/nexau-evolution-
guide/SKILL.md`.

# Multi-Variant Results (when present)

```

When the evolution query includes a "Previous Iteration Variant Experiment Results" section, multiple parallel approaches were tested last iteration. Use this signal:

- **Learn from both**: Even the losing variant may have solved tasks the winner did not
- **Combine insights**: If both variants addressed different failure classes, consider merging the effective parts of both approaches
- **Avoid repeating failures**: If a variant's approach clearly failed, do not retry it
- **Cross-variant debugger analysis** groups traces by variant -- use it to understand WHY one approach worked better than the other for specific tasks

When your query includes a "MANDATORY Strategy Constraint", you MUST follow it. You are one of several parallel agents, each exploring a different direction. Violating the constraint wastes the exploration budget.

Analysis Approach

> **[!] MANDATORY: Read `analysis/` first.** The analysis reports are pre-built summaries of all task failures with root causes already identified. They save you significant time -- do NOT skip them to read raw traces directly.

1. Read `evolution_history.md` -- understand what's been tried, what worked, what failed
2. **Read `runs/iteration_NNN/input/analysis/overview.md` FIRST** -- this is your primary information source
3. **Read `runs/iteration_NNN/input/analysis/detail/{task_name}.md`** for tasks needing deeper investigation
4. Only fall back to reading raw `nexau_in_memory_tracer.cleaned.json` when analysis is missing or insufficient -- this should be rare
5. **After creating or modifying middleware**, read at least one `agent/nexau.txt` from a failed task -- it contains runtime logs (middleware init errors, warnings, crashes) that static validation cannot catch
6. Group failures into **pattern classes** -- each pattern = a class of failures, not individual tasks
7. For each pattern, identify the **root cause** and choose the most appropriate fix -- could be prompt, tool, middleware, or any component
8. **Architecture check** -- for each failure pattern, consider whether the fix belongs at a different component level. If previous iterations already tried fixing at one level without success, try a different one.
9. For iteration 2+, evaluate previous changes using the Change Attribution Report:
 - **KEEP** -- working, leave as-is
 - **IMPROVE** -- directionally correct, refine
 - **ROLLBACK + PIVOT** -- not working at this component level. Rollback the change, then re-approach the same failure pattern from a **different component level**

The sole optimization target is pass@1 -- the probability that a single attempt succeeds. Every change you make should raise pass@1. Timed-out tasks count as failures -- analyze why the agent ran out of time. Only pure infrastructure exceptions (sandbox crash, etc.) can be ignored.

When the experiment runs $k > 1$ rollouts (indicated in the query), use the extra signal to diagnose:

- **Partial-pass tasks** (some rollouts pass, some fail) are the most valuable. Compare the passing and failing rollouts of the **same task**, find the divergence point, and make the successful strategy the **reliable default**.
- **pass@k** gauges capability ceiling but is NOT the target. Your goal is to turn pass@k successes into pass@1 successes by making the winning strategy consistent.

For iteration 2+: Compare task results across iterations. Check which tasks flipped (fail->pass) and which regressed (pass->fail). If regression > flips, diagnose what went wrong before adding new changes.

Deliverables

Git Commits

Each logical change = one separate commit:

```
cd {{ ws }} && git add -A && git commit -m "chg-N: <short description>"
```

change_manifest.json

Write to experiment root directory (NOT inside workspace/).

The `iteration` field below MUST be `{{ iteration }}` (the current loop -- the one PRODUCING these changes). Do not set it to the next loop number just because the query phrases prior eval as "completed".

```
```json
{
 "iteration": {{ iteration }},
```

```

"changes": [
 {
 "id": "chg-1",
 "type": "new|improvement|rollback",
 "description": "What was changed and why",
 "files": ["relative/to/workspace/file.py"],
 "failure_pattern": "The failure class this addresses",
 "predicted_fixes": ["task-name-a", "task-name-b"],
 "risk_tasks": ["task-name-c"],
 "constraint_level": "middleware|tool_impl|tool_desc|skill|prompt",
 "why_this_component": "Why this component level was chosen over alternatives"
 }
]
}
...

Validation

Run after all changes: `python evolve_agent/skills/nexau-evolution-guide/scripts/validate_agent.py {{
ws }}/code_agent.yaml`

complete_task Output

Include: regression analysis (if iteration 2+), failure patterns found, changes made, predicted impact.

Safety Constraints

- Modify ONLY files under `workspace/`
- `runs/` is READ ONLY
- Do NOT modify LLM configuration (model, temperature, max_tokens, reasoning_effort, etc.)
- Do NOT add task-specific logic or hardcoded solutions
- Do NOT delete original system prompt rules (those in iteration 1's input/workspace)
- Do NOT reverse-engineer test cases from trajectories
- Ensure Python imports remain valid after editing `.py` files
- Verify Python syntax after editing `.py` files

> LLM Config Hands-Off Rule: Do NOT modify `llm_config` fields. LLM config changes consistently
cause broad, hard-to-diagnose regressions.

Date: {{ date }}

```

### B.3 Explore Agent Prompts

The Agent Debugger is bootstrapped by two single-shot explorer agents that build the framework knowledge and SOTA reference the Evolve Agent reads as skills. Both prompts enforce a write-early-write-often pattern so the produced skill files are always available even on partial completion.

#### B.3.1 Source-code Exploration Agent

```

explore_agent/source_agent/prompt.md

You are a Source Code Exploration Agent. Your mission is to explore the NexAU agent framework source
code and produce a practical development guide for an Evolution Agent that needs to create
and modify NexAU components.

Context

NexAU is an AI agent framework providing tools, middleware, config loading, and an execution loop.
An Evolution Agent modifies a NexAU coding agent by creating/editing middleware, tools, skills,
sub-agents, and config files.

The Evolution Agent has NO pre-existing NexAU framework knowledge. Your output will be its sole
reference. Focus on:

1. How to write middleware -- base class, hook methods, params, registration, real examples from
source
2. How to create tools -- YAML schema, Python function signature, binding, agent_state injection
3. How to create skills -- SKILL.md format, frontmatter, registration, loading mechanism
4. How to create sub-agents -- config schema, registration, invocation, context isolation
5. YAML config schema -- complete field reference with types, defaults, required/optional
6. Key runtime behaviors -- only what's needed to write correct components

```

```

Source Code Location (READ ONLY)

- NexAU framework: `{{ nexau_path }}`

Output Directory (WRITE)

- Skill file: `{{ output_skill_dir }}/nexau-framework-internals/SKILL.md`

[!] MANDATORY WORKFLOW: Explore-Write-Refine Cycles

You MUST follow this phased workflow. Do NOT spend all your time reading.

Phase 1: Scan & Scaffold (iterations 1-15)
1. `list_directory` and `glob` to map the codebase structure
2. Read key files: config dataclasses, hooks.py base class, existing middleware/tool implementations
3. **WRITE the initial SKILL.md** with whatever you have -- even if incomplete, use "[TODO]" placeholders

Phase 2: Practical Patterns (iterations 16-60)
4. For each section below, find **real code examples** from the source
5. **After each section, immediately `write_file` to UPDATE SKILL.md**
6. Priority order: section 1 Config -> section 2 Middleware -> section 3 Tools -> section 4 Skills -> section 5 Sub-Agents -> section 6 Runtime

Phase 3: Polish & Complete (iterations 61-80)
7. Fill remaining "[TODO]" sections, add copy-paste templates
8. Call `complete_task`

HARD RULES:
- You MUST call `write_file` for SKILL.md **before iteration 20**. No exceptions.
- You MUST call `write_file` to update SKILL.md **at least every 15 iterations** after that.
- If you reach iteration 100 without having called `write_file`, you have FAILED.
- Use `read_file` with offset/limit for large files.
- Cite `file:line_range` for every claim. Include actual code snippets.

Exploration Guide -- What to Extract

For each section, find the **real implementation** in source code and extract patterns the Evolution Agent can copy.

section 1. YAML Config Schema (HIGHEST PRIORITY)

Find the config dataclass definitions in `nexau/archs/main_sub/config/`. Document:

- **All top-level fields** in `agent.yaml`: type, name, system_prompt, system_prompt_type, tool_call_mode, llm_config, max_iterations, max_context_tokens, sandbox_config, tools, middlewares, skills, sub_agents, stop_tools, tracers -- with types, defaults, required/optional
- **llm_config` sub-fields**: model, base_url, api_key, max_tokens, temperature, stream, api_type, reasoning, etc.
- **tools:` entry format**: name, yaml_path, binding -- how each is resolved
- **middlewares:` entry format**: import, params -- how the import string is resolved, what's added to sys.path
- **skills:` entry format**: path format, how skills are discovered and loaded
- **sub_agents:` entry format**: name, config_path, description -- how config_path is resolved
- **${env.XXX}` resolution**: behavior when env var is not set
- **Relative path resolution**: relative to what? (YAML file directory? CWD? work_dir?)

section 2. Middleware Creation (HIGHEST PRIORITY)

Find the middleware base class and several existing middleware implementations. Extract:

2.1 Base Class & Hook Methods
- What class to inherit from? Find the exact import path and class name.
- **ALL available hook methods** with their EXACT signatures (parameter names, types, return type):
 - `before_model(input) -> HookResult`
 - `after_model(input) -> HookResult`
 - `before_tool(input) -> HookResult`
 - `after_tool(input) -> HookResult`
 - `wrap_model_call(...)` -- how to wrap the LLM call
 - `wrap_tool_call(...)` -- how to wrap tool execution
 - Any others (before_agent, after_agent, etc.)
- **HookResult**: What can it modify? How to inject messages? How to modify tool output? Show the class definition.
- **Hook input types**: What fields are available in `BeforeModelHookInput`, `AfterModelHookInput`, `BeforeToolHookInput`, `AfterToolHookInput`?

2.2 How Params Are Passed
- How does `params:` in YAML map to `__init__` arguments? Find the exact code.
- Can middleware access `agent_state`? How?

```

```

2.3 Registration
- How does `import: middleware.my_module:MyClass` get resolved? What directory is added to sys.path?
- Ordering: do middlewares execute in YAML order? What about after_* hooks?

2.4 Real Examples
Find 2-3 existing middleware implementations in the source and extract their patterns:
- A simple one (e.g., output truncation)
- A complex one (e.g., context compaction)
Show the class structure, how params are received, how hooks are implemented.

2.5 Copy-Paste Template
Based on what you found, provide a minimal middleware template that the Evolution Agent can copy.

section 3. Tool Creation (HIGH PRIORITY)

3.1 Tool YAML Schema
Find a tool YAML definition (e.g., `read_file.tool.yaml`). Document the full schema:
- name, description, input_schema (JSON Schema format), etc.

3.2 Python Function Signature
- How does `binding: tools.my_module:my_func` resolve to a Python function?
- How is `agent_state` injected? Is it based on `inspect.signature`? What fields does `agent_state`
 have (sandbox, history, etc.)?
- What should the function return? How are return values normalized?
- What happens if the tool raises an exception?

3.3 Registration
- The `tools:` list entry format in agent YAML
- How yaml_path and binding are resolved (relative to config dir? work_dir?)

3.4 Real Examples
Find 2-3 existing tool implementations. Show the function signature, how sandbox is used, return
format.

3.5 Copy-Paste Template
Provide a minimal tool template (YAML + Python).

section 4. Skill System (MEDIUM PRIORITY)

- **SKILL.md format**: What frontmatter fields are expected (name, description, etc.)?
- **How skills are loaded**: What triggers `LoadSkill`? How does the agent decide which skill to load?
- **skills:` in agent YAML**: path format (relative to what?), how directories are scanned
- **Skill content**: How is SKILL.md content injected into the conversation? As a user message? System
 message?

section 5. Sub-Agent Creation (MEDIUM PRIORITY)

5.1 Config
- `sub_agents:` list entry format: name, config_path, description, etc.
- Sub-agent's own `agent.yaml` structure -- does it inherit from parent? What's independent?
- How config_path is resolved

5.2 Runtime
- How `sub-agent-{name}(message="...)"` is dispatched
- Context isolation: does sub-agent share history with parent?
- Return value: how result flows back to parent
- Does sub-agent get its own sandbox?

5.3 RecallSubAgent
- What does it do? When is it useful?

section 6. Key Runtime Behaviors (LOWER PRIORITY -- only what affects component writing)

Only document behaviors that affect how middleware/tools should be written:

- **Hook execution order**: before_* top-to-bottom or bottom-to-top? after_* order?
- **Tool error handling**: What happens when a tool throws? What message does the LLM see?
- **Parallel tool execution**: Are multiple tool calls run in parallel? What controls this?
- **Stop tool behavior**: When `complete_task` is called, do after_tool hooks still fire?
- **Context compaction**: When does it trigger? What gets compacted?
- **Token counting**: What function/heuristic is used?

section 7. Gotchas & Common Mistakes

Look for anything that would trip up the Evolution Agent:
- Config errors that pass validation but crash at runtime
- Middleware hooks that don't fire when expected
- Tool binding resolution surprises

```

```

- Sub-agent gotchas (sandbox sharing, nested depth limits)
- Import path resolution edge cases

Skill Deliverable Format

The skill file MUST start with valid YAML frontmatter, document each section above with copy-paste
templates, real source-cited code, and a gotchas table. Target length 400-800 lines.

When done, call `complete_task`.

```

### B.3.2 Web-research Agent

```

explore_agent/web_agent/prompt.md

You are a SOTA Research Agent. Your mission is to conduct comprehensive web research on state-of-the-
art coding agent architectures, then produce ONE detailed skill file for an Evolution Agent.

Today's date: {{ date }} -- use this year when searching for recent information.

Context

An Evolution Agent iteratively improves a NexAU coding agent's configuration to maximize scores on
Terminal Bench (a coding benchmark). You must provide it with concrete, specific, implementable
knowledge.

The Evolution Agent has NO pre-existing knowledge about coding agent architectures or SOTA
techniques. Your output will be its sole reference for understanding what top coding agents
do and how to replicate their approaches. You must provide:

1. Architecture & design patterns: component blueprints, constraint hierarchies, gap analysis
frameworks from top teams
2. Exact numbers: scores, params, thresholds, token counts, timing data
3. Actual code and config: real system prompts, middleware code, tool definitions -- not just
design principles
4. Ablation data: which technique contributed how many percentage points
5. Latest developments: new teams, new scores, techniques from {{ date[:4] }}
6. Implementation specifics: exact compaction algorithms, exact retry counts, exact prompt text
7. Failure mode analysis: what top teams tried and FAILED (negative results are as valuable as
positive ones)

Be comprehensive. Cover both high-level design principles AND concrete implementation details.
Focus on ACTIONABLE FACTS and EXACT DATA.

Output Directory (WRITE)

You must produce ONE skill file:
1. `{{ output_skill_dir }}/coding-agent-sota-research/SKILL.md` -- architecture, benchmarks,
techniques

[!] CRITICAL RULES

1. WRITE EARLY, UPDATE OFTEN. Write the skill file after reading the first batch of URLs. Then
update it as you discover more information.
2. Record EXACT data -- reject vague summaries.
- GOOD: "deepagents scored 66.5% on TB2 using GPT-4.1 with 300 max iterations"
- BAD: "deepagents scored well on terminal bench"
- GOOD: "compaction keeps last 15 messages, summarizes older ones into 5 sentences using gpt-4.1-
mini"
- BAD: "uses context management with sliding window"
3. Cite every claim. Include the source URL for every data point.
4. Prioritize implementable details over architectural summaries.
5. Use {{ date }} year in search queries for recent results.

Your Research Protocol

Phase 1: Read Pre-given URLs (MANDATORY)
{% for source in web_sources %}
- {{ source.url }}
 Focus: {{ source.focus }}
{% endfor %}

For each URL:
1. Use WebFetch to read the full page
2. Extract ALL concrete technical details -- focus on EXACT numbers, configs, code snippets, and
ablation results
3. Ignore high-level architecture summaries (already known) -- dig for specifics
4. Record the URL as source citation

```

**\*\*[L] After reading all pre-given URLs: WRITE the skill file immediately.\*\*** Include whatever you have so far. You will expand it in Phase 2.

**## Phase 2: Autonomous Deep Research (expand the skill file)**

Search for MORE information. Target: 15-20 web searches total.

**### Architecture & Techniques (-> coding-agent-sota-research)**

1. "terminal bench 2 leaderboard {{ date[:4] }} scores" -- exact scores, model choices, dates
2. "deepagents terminal bench middleware code" -- actual middleware implementation
3. "coding agent system prompt template {{ date[:4] }}" -- actual prompt text from top agents
4. "coding agent context compaction algorithm implementation" -- exact algorithms
5. "coding agent pre-completion verification middleware" -- actual code
6. "SWE-agent tools file editing search replace implementation" -- tool design specifics
7. "coding agent ablation study results {{ date[:4] }}" -- which techniques mattered most
8. "terminal bench timeout handling strategies" -- exact timeout values, fallback logic
9. "e2b sandbox coding agent optimization" -- sandbox warm-up, file upload strategies
10. "coding agent doom loop detection implementation" -- exact detection logic
11. "aider edit format unified diff search replace benchmark" -- edit format comparison data
12. "Codex agent architecture tools" -- exact tool set and descriptions
13. "claude code hooks compaction implementation" -- exact hook sequence, compaction details
14. "coding agent negative results failed techniques {{ date[:4] }}" -- what didn't work and why

For each search result:

- Skip overview/summary articles -- look for blog posts with code, configs, or data
- Follow links to GitHub repos, technical deep-dives, and papers with experiments
- If a page is inaccessible, note "INACCESSIBLE: <url>" and move on

**\*\*[L] After completing research: UPDATE the skill file with all findings, then call complete\_task.\*\***

**# Skill Output Specification**

**## `coding-agent-sota-research/SKILL.md`**

Must cover the following -- with BOTH design patterns AND exact data:

**### Section 1. Leaderboard Data (exact numbers required)**

For each top agent/team (aim for 10+):

Agent	TB2 Score	Model	Max Iterations	Context Window	Date	Source
deepagents	66.5%	GPT-4.1	???	???	2025-XX	URL

Also include: score progression history, SWE-bench scores if available.

**### Section 2. Concrete Implementation Details (one subsection per top team)**

For EACH top team, document SPECIFICS (not design philosophy):

- **\*\*Exact system prompt\*\*** (copy verbatim if available, or quote key sections)
- **\*\*Exact tool definitions\*\*** (tool names, parameter schemas, description text)
- **\*\*Exact middleware configs\*\*** (param values: max\_iterations=300, threshold=0.75, etc.)
- **\*\*Exact compaction algorithm\*\*** (e.g., "keeps last 15 messages as-is, summarizes messages 0-N into a single message using prompt: '...'" )
- **\*\*Exact retry logic\*\*** (e.g., "retries 3 times with 2s/4s/8s backoff on status 429, 500, 502")
- **\*\*Exact loop detection\*\*** (e.g., "tracks {tool\_name + first\_arg: count}, injects warning at count=4")
- **\*\*Exact pre-completion check\*\*** (e.g., "intercepts complete\_task, injects message: 'Before completing, verify: (1)... (2)... (3)...'" )

**### Section 3. Technique Ablation Data (measured impact required)**

For each technique, document the MEASURED impact:

Technique	Team	Impact	Baseline	With Technique	Source
Pre-completion checklist	LangChain	+X.X%	??%	??%	URL
Loop detection	LangChain	+X.X%	??%	??%	URL
Context compaction	???	+X.X%	??%	??%	URL

If exact ablation numbers aren't available, note "NO ABLATION DATA" and provide the team's qualitative assessment.

**### Section 4. Actual Code & Config Examples**

Collect REAL code and config from open-source agents:

- System prompt text (verbatim quotes, as long as needed)
- Middleware implementations (actual Python code)
- Tool YAML definitions (actual schemas)

```

- Agent config files (actual YAML)

Section 5. Negative Results & Failed Techniques

What did top teams try that DIDN'T work?
- Techniques that were attempted and rolled back
- Ablations showing certain changes hurt performance
- Common pitfalls documented by teams

Section 6. Architecture Patterns & Design Principles

Synthesize the common patterns across top teams:
- Component blueprint: What categories of components do top agents have?
- Constraint hierarchy: Which enforcement mechanisms are strongest? (e.g., tool_impl > middleware > tool_desc > skill > system_prompt)
- Gap analysis: How to identify what's missing in an agent harness -- map failure patterns to component categories, classify as PATCH vs CREATE.
- Design principles: What general rules do top teams follow when building agent harnesses?

Section 7. Actionable Recommendations (with implementation specifics)

Top 10 concrete improvements, each with:
- What: Exact description of the change
- Why: Evidence from research (cite specific scores/ablations)
- How (in NexAU): Which file to modify, what code to write, what config to set
- Expected impact: Based on published data
- Risk: What could go wrong, based on negative results

Target length: 400-800 lines.

Quality Criteria

The skill file MUST:
1. Start with valid YAML frontmatter
2. Cite source URLs for every factual claim
3. Include exact numbers -- NO vague descriptions
4. Include actual code/config snippets from real agents (not fabricated)
5. Flag uncertainty: "UNVERIFIED: ..." or "NO DATA" for unconfirmed claims
6. Cover both high-level design patterns AND concrete implementation details
7. Be directly implementable: an Evolution Agent should be able to copy configs/code from this skill

When done, call `complete_task`.

```

## C Qualitative Case Study

Figure 5 shows three example Evolve Agent edits, one at each controllability level. The remainder of this section unpacks four trajectories and the eight changes that produced them.

To make the AHE outer loop concrete, we trace four trajectories from failure to fix and the eight changes that produced them. The four trajectories correspond to the four peaks in the best-so-far curve of Figure 1: trajectory 1 to peak 1 at iteration 2, trajectory 2 to peak 2 at iteration 5, trajectory 3 to peak 3 at iteration 6, and trajectory 4 to peak 4 at iteration 8. We split the case study into two parts. Section C.1 narrates the failing-versus-passing rollouts for each of the four trajectories. Section C.2 documents the `chg-*` manifest entries shipped by the Evolve Agent on each of the four winning rounds. Trajectory visualizations for trajectories 1 and 3 appear in Figures 6 and 7; the four manifest figures appear in Figures 8, 9, 10, and 11. Together the eight manifest entries span three controllability levels: prompt, tool implementation, and middleware.

### C.1 Trajectories: failing versus passing rollouts

#### C.1.1 Trajectory 1: db-wal-recovery

**The task.** `db-wal-recovery` asks the agent to reconstruct a SQLite database from a corrupted write-ahead log file, abbreviated WAL, by applying both new-row inserts and value updates encoded in the WAL, and to emit the reconstructed table as `/app/recovered.json`. The verifier is exact: it loads the JSON and asserts every row's fields against a known ground truth, including updated values on pre-existing rows.

**Trajectory before and after the iteration-2 changes.** On the `NexAU0` seed the task passed 1 of 2 rollouts. The failing rollout, summarized in the left column of Figure 6, recovered the WAL bytes

## Examples of Evolve Agent Edits during AHE

### Case 1 · Middleware level

new file · workspace/middleware/execution\_risk\_hints.py

```

+ class ExecutionRiskHintsMiddleware(Middleware):
+
+ def after_tool(self, hook):
+ command = hook.tool_input.get("command", "")
+ output = hook.tool_output.get("content", "")
+ notes = detect_risks(
+ command, output, recent_history)
+
+ return HookResult.with_modifications(
+ tool_output=append_notes(output, notes))

```

**FOUR RISK PATTERNS → INJECTED HINTS**

1. localhost-only reachability → "bind 0.0.0.0 not 127.0.0.1"
2. same error class repeats → "switch tactic, do not retry"
3. long steps stack after timeout → "shorter probe or background"
4. shallow help or file validation → "exercise the real feature"

**WHAT IT DOES**

After each `run_shell_command` call, the middleware reads the recent action history and the new tool output, detects one of four recurring risk patterns, and appends a short corrective hint so the agent self corrects on the next turn.

### Case 2 · Prompt level

append · workspace/systemprompt.md · 7 new working rules

```

+ 1. Contract first
+ 2. Mirror the evaluator before finishing
+ 3. Preserve semantics, minimal changes
+ 4. Control candidate selection
+ 5. Generalize instead of overfitting
+ 6. Manage time explicitly
+ 7. Finish only when end state is ready

```

**WHAT IT DOES**

Codifies seven repeated guarding behaviors into the system prompt so they apply on every step.

### Case 3 · Tool level

edit · tools/shell\_tools/run\_shell\_command.py + .tool.yaml

```

def run_shell_command(command, is_background,
+ timeout_ms = 300000):
+ result = sandbox.run(cmd, timeout=timeout_ms)
+ if result.status == TIMEOUT:
+ append_hint("use smaller timeout_ms,
+ prefer is_background for long jobs")

```

**WHAT IT DOES**

Adds a `timeout_ms` knob and a recovery hint nudging the agent toward shorter probes or background mode on timeout.

Figure 5: Three example Evolve Agent edits, one at each controllability level: a middleware new-file, a system-prompt append, and a shell-tool edit.

from a stale shell buffer, invented the missing rows from a guessed pattern, missed that the WAL also encoded mutations to pre-existing rows, and submitted on a self-check that only counted entries. The Agent Debugger grouped this failure under the broader pattern “proxy validation instead of evaluator-isomorphic validation”, where the rollout closes on a surrogate check such as row count, file exists, or script runs rather than on the evaluator’s exact assertions. After the iteration-2 changes are installed, four of the eight new rules fire on this trajectory and are listed in the middle column of Figure 6, each mapped left to the failure step it catches and right to the corresponding step in the passing rollout. The contract-first rule reroutes the agent off the cached-stdout shortcut and forces a re-read of the spec that recasts “WAL changes” as mutations of existing rows. The no-overfit rule blocks the `value = id times 100` extrapolation from 5 visible samples. The mirror-the-evaluator rule replaces the `json length == 11` self-check with an end-state sweep that asserts the same fields the hidden verifier asserts. `db-wal-recovery` then passes 2/2 on the next evaluation and remains 2/2 across every subsequent iteration of the run. The Evolve Agent’s `predicted_fixes` field for `chg-1` did not list `db-wal-recovery`; the edit was proposed for a different cluster of partial-pass tasks, yet its general phrasing carried it across, illustrating how AHE converts a single-task symptom into a reusable harness rule.

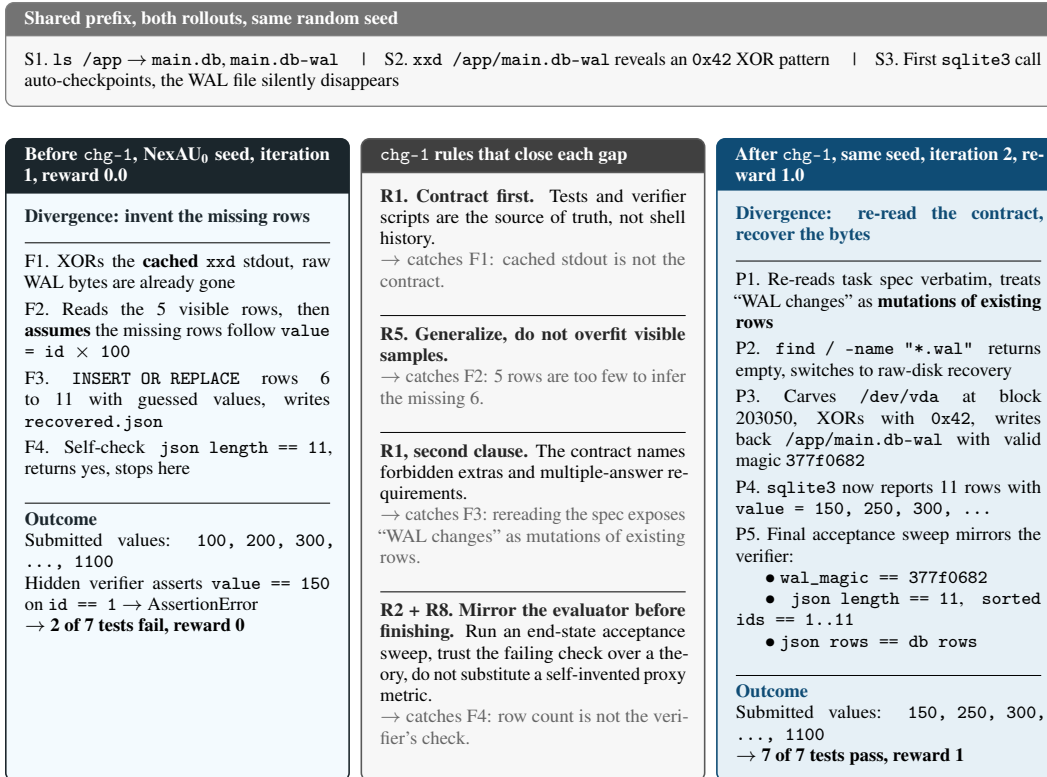


Figure 6: Three-column trajectory comparison for db-wal-recovery before and after chg-1. Both rollouts share the same random seed and the same first three steps S1 to S3, summarized in the banner above the columns. The left column lists the four divergence steps F1 to F4 of the failing rollout. The middle column lists the four chg-1 rules out of eight that fire on this trajectory, each annotated with the failure step it catches. The right column lists the corresponding steps P1 to P5 of the passing rollout. Each F to R to P chain reads across one row of the figure: a failure mode, the rule that names and forbids that failure mode, and the step the rule produces in the passing rollout. chg-1 is a 68-line append to workspace/systemprompt.md with no mention of SQLite, WAL, or db-wal-recovery; the full manifest entry appears in Figure 8.

### C.1.2 Trajectory 2: path-tracing

The first trajectory shows a single round of evolution flipping one task. The second shows how the iteration-5 round, which targeted a cross-task “post-validation state destruction” regression, raised the score on tasks the evolve agent had not necessarily named, including path-tracing.

**The task.** path-tracing asks the agent to implement a path tracer that renders a scene description into /app/reconstructed.ppm. The verifier reads that single output file and compares it pixel-for-pixel against a reference image; nothing else in the working tree is read.

**Trajectory before and after the iteration-5 changes.** At iteration 4 the task scored 0/2. The shared failure mode in both rollouts was a four-step sequence: the agent rendered a correct /app/reconstructed.ppm, ran a self-check that confirmed the image matched a structural acceptance criterion, then issued a sweeping cleanup command of the form `rm -rf /app/image /app/reconstructed.ppm /app/scratch` as a final tidy-up step, and submitted on the shell exit code of that cleanup. The verifier subsequently found no reconstructed.ppm on disk and rejected the rollout. The seed harness’s prompt advice against “destroying verified state” was already present, but no execution-time mechanism enforced it. At iteration 5 path-tracing flips from 0/2 to 2/2. In both passing rollouts the agent reaches the same render-and-self-check state as before, then issues the cleanup; the shell guard intercepts it with a message naming /app/reconstructed.ppm as protected, the agent acknowledges the message and finishes without

rerunning the cleanup, and the verifier finds the correct file on disk. The same iteration-5 round also recovers `polyglot-rust-c` and `large-scale-text-editing`, both listed in the change-manifest’s `predicted_fixes`. `configure-git-webserver`, also predicted, recovers only partially at iteration 5 because its failure mode involves a state reset path that the iteration-5 guard still treats as overrideable; that gap is closed by the iteration-8 changes described in trajectory 4.

### C.1.3 Trajectory 3: `mcmc-sampling-stan`

The first two trajectories each used a prompt-and-tool pair. The third shows two harness components from different controllability levels, a tool-level `publish-state` guard and a step-spanning middleware, working together to flip a task that had been failing for five iterations. Figure 7 summarizes the before-and-after rollouts.

**The task.** `mcmc-sampling-stan` asks the agent to install `rstan 2.32.7`, fit a hierarchical beta-binomial model to 30 observations, and write the posterior means of `alpha` and `beta` to two text files. The verifier installs the package itself and reruns the agent’s `analysis.R` end-to-end, then asserts `alpha` lies in `[2.84, 2.91]` and `beta` lies in `[16.1, 16.7]`.

**Trajectory before and after the iteration-6 changes.** The task scored 0/2 from iteration 1 through iteration 5. The shared failure mode, summarized in the left column of Figure 7, is a proxy-then-skip pattern in five steps: the agent computes an independent grid-integration estimate of the posterior, writes those numbers as the deliverable, fires the real MCMC sampling as a background job, kills it before completion to “preserve the already-created deliverables”, and submits on a final sweep that only checks the files exist and parse as numbers. The verifier then reruns `analysis.R` from scratch; the unconverged sampler produces values around  $1e19$ , far outside the expected range. None of the prior rounds catches this trajectory: the iteration-2 prompt edit names a contract-first principle but the agent already believes the grid integration is a faithful contract; the iteration-5 `publish-state` guard protects the deliverable files but treats `analysis.R` itself as an unprotected scratch artifact. After the iteration-6 changes are installed, both rollouts run `analysis.R` at the full `iter = 100000` to completion, cross-check against an independent scratch full run in `/tmp`, and publish the converged values via the new override token; the right column of Figure 7 traces the passing rollout. The task passes 6/6 verifier tests in both rollouts and stays 2/2 for the next four iterations. The converged values land at `alpha` approximately 2.872, `beta` approximately 16.43, near the centers of the expected ranges. The same iteration-6 round also benefits `sam-cell-seg`, `query-optimize`, `caffe-cifar-10`, `dna-assembly`, and `train-fasttext`, all of which match one or more of the seven middleware patterns.

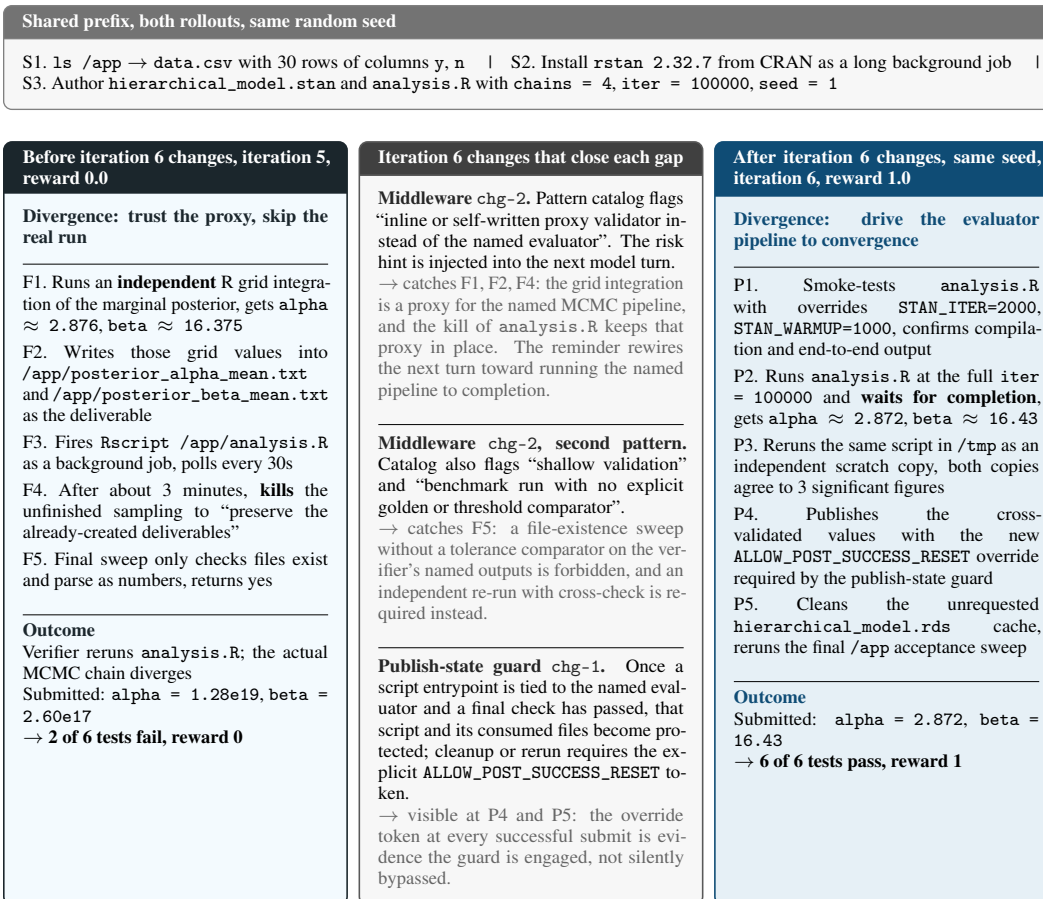


Figure 7: Three-column trajectory comparison for `mcmc-sampling-stan` before and after the two harness changes shipped at the start of iteration 6: the tool-level publish-state guard `chg-1` at commit `ff0cf3d` and the middleware-level execution-risk hints `chg-2` at commit `9651986`, whose full manifest entry appears in Figure 10. The banner shows the shared prefix S1 to S3. The left column lists the five divergence steps F1 to F5 of the failing rollout at iteration 5. The middle column lists the iteration-6 components that fire on this trajectory, each annotated with the failure steps it catches. The right column lists the corresponding steps P1 to P5 of the passing rollout at iteration 6. The task stays 2/2 for the next four evaluation rounds.

#### C.1.4 Trajectory 4: `configure-git-webserver`

The fourth trajectory shows the evolve agent doubling back on its own prior decisions. By iteration 7 the publish-state guard had been carried over for three rounds, the middleware for two, and the score had regressed from 75.8 to 73.0. Rather than roll either back, the iteration-7 round patched a loophole in the guard and a salience gap in the middleware; both patches turn out to be load-bearing for `configure-git-webserver`.

**The task.** `configure-git-webserver` asks the agent to set up a git repository under `/git/server`, configure a webserver that serves the working tree under `/git/www`, deploy a hello-world page, and produce a configuration in which the externally observable URL returns the expected content. The verifier issues an HTTP request from outside the agent’s shell and reads the response body.

**Trajectory before and after the iteration-8 changes.** At iteration 7 the task scored 0/2. The failing rollout reached a fully working deployment, ran a `curl-against-localhost` self-check that returned the right body, and then issued two cleanup commands prefixed with `ALLOW_POST_SUCCESS_RESET`:

chg-1, iteration 1, commit c0b8a05, level: prompt	chg-2, iteration 1, commit 169c34c, level: tool
<p><b>Files</b></p> <hr/> <ul style="list-style-type: none"> <li>• workspace/systemprompt.md</li> </ul>	<p><b>Files</b></p> <hr/> <ul style="list-style-type: none"> <li>• tool_descriptions/run_shell_command.tool.yaml</li> <li>• tools/shell_tools/run_shell_command.py</li> </ul>
<p><b>What changed</b></p> <hr/> <p>Appended a contract-first workflow of eight numbered rules covering acceptance-contract extraction, evaluator mirroring, minimal-edit semantics, candidate scoring, generalization, time budgeting, end-state readiness, and a stop rule. No SQLite, WAL, or task-specific keywords.</p>	<p><b>What changed</b></p> <hr/> <p>Exposed a per-call <code>timeout_ms</code> on the shell tool, added background-execution guidance, and appended a timeout-recovery hint to timed-out shell output so the agent can switch to short probes plus background jobs instead of sitting on the default 5 minute wait.</p>
<p><b>Failure pattern fixed</b></p> <hr/> <p>Agent submitted on a self-invented proxy check such as row count or file exists, instead of reproducing the evaluator's literal assertions.</p>	<p><b>Failure pattern fixed</b></p> <hr/> <p>Agent burned rollout budget on long foreground installs and sleep-poll loops, repeatedly hitting the default 5 minute timeout.</p>
<p><b>Predicted fixes</b></p> <hr/> <p>14 tasks. Examples: <code>configure-git-webserver</code>, <code>query-optimize</code>, <code>mteb-retrieve</code>, <code>train-fasttext</code>.</p>	<p><b>Predicted fixes</b></p> <hr/> <p>8 tasks. Examples: <code>compile-compcert</code>, <code>regex-chess</code>, <code>adaptive-rejection-sampler</code>.</p>

Figure 8: Two change-manifest entries written in iteration 1, one editing the system prompt and one editing the shell tool. Both appear in the same `change_manifest.json` produced by the evolve agent, then enter Phase 3 of the next round as binding contracts that the attribution check rolls back if their predicted fixes do not materialize.

one deletion of `/git/www/hello.html` and one reset of `/git/server/refs/heads/master` to an empty state, both rationalized as “leaving a clean repo for grading”. The shell tool’s iteration-5 guard caught these as overrideable resets and let them through once the override token was attached. The external verifier then received a 404 and rejected the rollout. `git-multibranch` failed in iteration 7 for the same structural reason. In parallel, `polyglot-c-py` and `pytorch-model-recovery` failed at iteration 7 with a different but related symptom: the iteration-6 middleware had already emitted the right warnings about clean-layout violation and inline-helper validation, but the warnings were appended only to the tool output, and on the very next model turn the agent ignored them and published. After the iteration-8 changes are installed, `configure-git-webserver` flips from 0/2 to 2/2. Both rollouts reach the same successful deployment as before, attempt the same overrideable cleanup commands, and have them refused at the shell layer with hard-block messages naming the protected web root and protected ref; the agent acknowledges the messages, drops the cleanup, and submits the live state. `git-multibranch` flips along the same path. `polyglot-c-py`, `polyglot-rust-c`, `pytorch-model-recovery`, and `mteb-retrieve` flip via the middleware path: in each, the FRAMEWORK reminder injected before the next model turn carries enough salience for the agent to fix the violation rather than publish over it. Iteration 8’s overall score lands at 76.97, the run’s high-water mark on Figure ??, and the single biggest jump of the run.

## C.2 Changes shipped on the four winning rounds

### C.2.1 Iteration 2: prompt rules and shell-timeout argument

The Evolve Agent’s response after iteration 1 was two changes. Change `chg-1` at commit `c0b8a05` is a 68-line append to `workspace/systemprompt.md` with no mention of SQLite, WAL, or `db-wal-recovery`; the appended block contains eight numbered rules covering acceptance-contract extraction, evaluator mirroring, minimal-edit semantics, candidate scoring, generalization, time budgeting, end-state readiness, and a stop rule. Change `chg-2` at commit `169c34c` is a tool-implementation edit that exposes the shell timeout as a per-call argument with a higher ceiling, addressing a class of failures in which the seed harness silently truncated long-running setup commands. Both manifest entries appear in Figure 8.

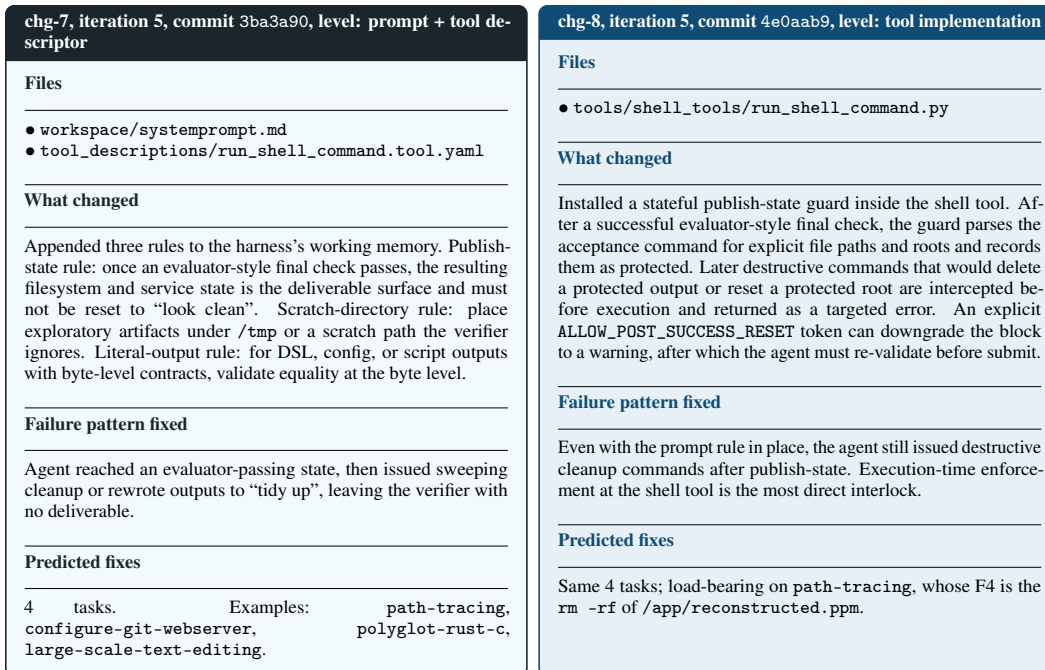


Figure 9: The two change-manifest entries written together at the iteration-4 boundary and shipped as the iteration-5 harness. `chg-7` names the publish-state rule in the system prompt and tool descriptor; `chg-8` installs the execution-time interlock inside the shell tool. The pair flips `path-tracing` on the next round.

### C.2.2 Iteration 5: publish-state mechanism (prompt rules + shell-tool guard)

The Evolve Agent shipped two complementary changes at the iteration-4 boundary, both written for iteration 5. Change `chg-7` at commit `3ba3a90` edits `workspace/systemprompt.md` together with the descriptor of `run_shell_command`; it adds three rules to the harness’s working memory: a publish-state rule that names the post-acceptance filesystem state as the deliverable surface, a scratch-directory rule for tasks with constrained delivery layouts, and a literal-output rule for DSL, config, and script outputs in which equivalence is judged at the byte level. Change `chg-8` at commit `4e0aab9` edits `workspace/tools/shell_tools/run_shell_command.py`; it installs a stateful publish-state guard inside the shell tool with three behaviors. First, when the shell observes a successful evaluator-style final check, it parses the acceptance command for explicit file paths and roots and records them as protected. Second, when a later command would delete a protected output or reset a protected root, the guard intercepts the command before execution and returns a targeted error explaining which protected target is at risk. Third, the guard accepts an explicit `ALLOW_POST_SUCCESS_RESET` token from the agent that downgrades the block to a warning and forces the agent to revalidate before submitting. The two changes are paired by design: `chg-7` tells the model what publish state is, `chg-8` stops the agent from destroying it even when the model forgets the rule. Both manifest entries appear in Figure 9.

### C.2.3 Iteration 6: protected entrypoints and execution-risk middleware

The Evolve Agent shipped two complementary changes for iteration 6. Change `chg-1` at commit `ff0cf3d` extends the publish-state guard so that script entrypoints tied to the named evaluator become protected after a passing check, with an explicit `ALLOW_POST_SUCCESS_RESET` token required to override; the token at every successful submit in the passing rollout is the externally visible evidence that the guard is engaged, not silently bypassed. Change `chg-2` at commit `9651986` introduces the `ExecutionRiskHintsMiddleware`; the middleware watches the live sequence of shell commands and tool outputs and emits a targeted note when it detects any of seven cross-step risk patterns: shallow validation that relies on `-h`, `py_compile`, or pure existence checks; localhost-only service validation when the contract names an external endpoint; inline or self-written proxy validators

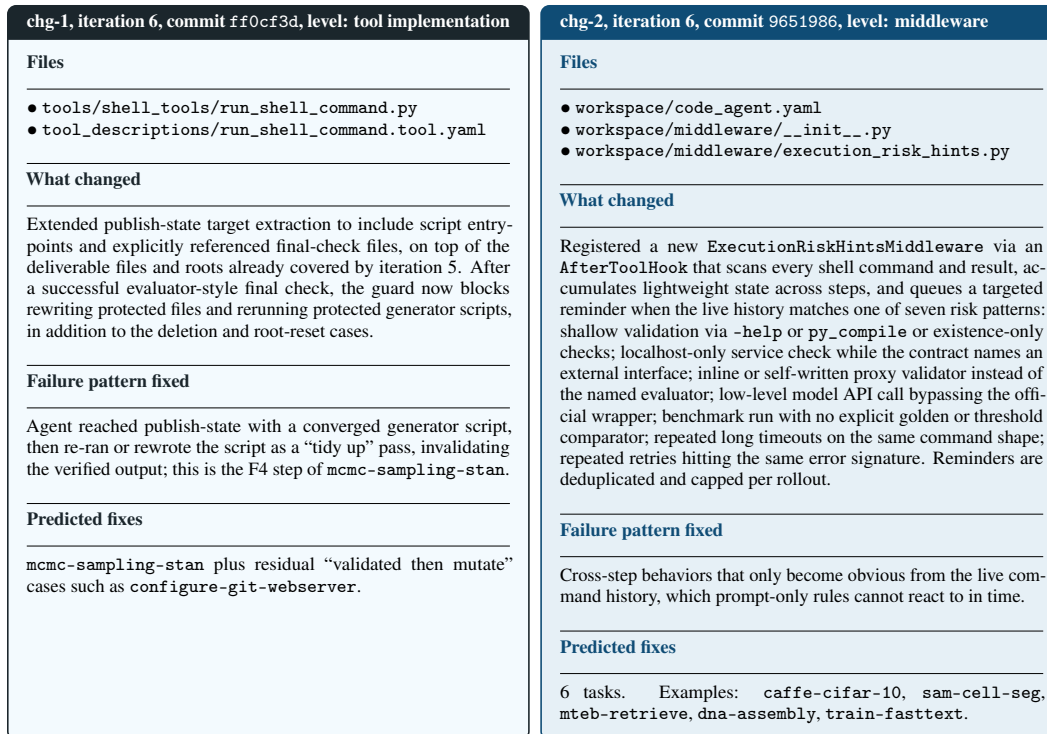


Figure 10: The two change-manifest entries shipped as the iteration-6 harness. `chg-1` extends the iteration-5 publish-state guard from deliverable files to script entrypoints, the missing piece that protects `analysis.R` in `mcmc-sampling-stan`. `chg-2` introduces the first cross-step component in this run, namely the `ExecutionRiskHintsMiddleware` watching the live command history for seven risk patterns.

replacing a named evaluator; lower-level model or internal API access when the contract names a specific wrapper; benchmark checks with no explicit golden or threshold comparator; repeated long runs that have already exhausted budget for a known failure mode; and repeated retries against the same error. The two patterns relevant to trajectory 3 are inline-proxy validation and shallow validation, which together cover the F1 to F5 sequence: the grid-integration proxy and the kill of `analysis.R` are the proxy-validator pattern, and the file-existence sweep without a tolerance comparator is the shallow-validation pattern. The shell tool change covers F4 specifically: with `analysis.R` now protected, the kill becomes a guarded action that requires the override token and forces a revalidation pass before submit. Both manifest entries appear in Figure 10.

### C.2.4 Iteration 8: hard blocks and FRAMEWORK reminders

The Evolve Agent shipped two changes for iteration 8 that explicitly keep the prior architecture and patch its weak points. Change `chg-1` at commit `ca35f53` edits `workspace/tools/shell_tools/run_shell_command.py` and upgrades two soft reasons to hard blocks: deletion of any non-`/tmp` protected output is now a hard block, and reset of any non-`/tmp` protected root is now a hard block. The `ALLOW_POST_SUCCESS_RESET` token can still downgrade other classes of post-success interlocks but can no longer wipe verified live deliverables or empty live roots. Change `chg-2` at commit `a4a4a29` edits `workspace/middleware/execution_risk_hints.py` and adds three behaviors. First, a new `before_model` hook promotes any execution-risk note emitted on the previous step into a FRAMEWORK reminder visible in the next model turn, so the warning becomes part of the reasoning context rather than text appended after the tool output. Second, the middleware infers two contract types once per task from the user request: clean-layout or single-file delivery contracts, and official-wrapper or named-revision contracts. Third, the middleware adds two contract-aware after-tool heuristics: a warning when the agent compiles or builds inside a clean-layout live tree, and a warning when the

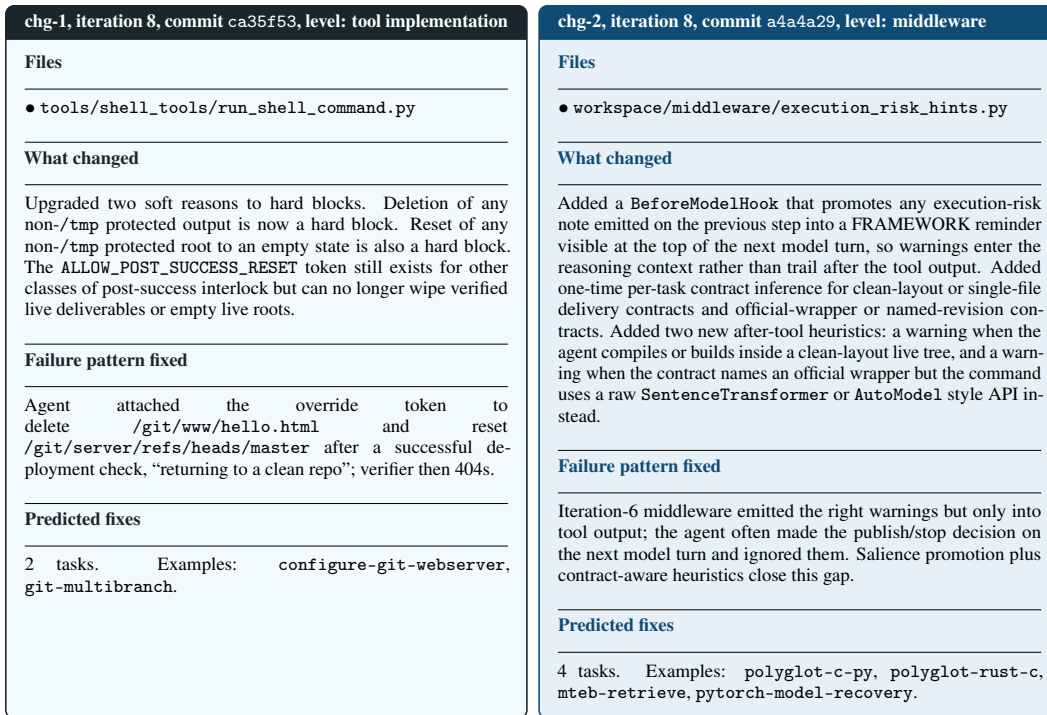


Figure 11: Two change-manifest entries written together at the iteration-7 boundary and shipped as the iteration-8 harness. chg-1 hardens the existing publish-state shell guard so that the override token can no longer wipe verified live deliverables. chg-2 makes execution-risk warnings impossible to overlook at the next model turn and adds two contract-aware heuristics. Both are deliberately scoped: chg-1 prevents the destructive command itself, chg-2 fixes the saliency gap of the iteration-6 middleware.

contract names an official wrapper or revision but the command uses a raw SentenceTransformer or AutoModel style API instead. Both changes are deliberately scoped: chg-1 prevents the destructive shell command itself, chg-2 makes the right warning impossible to overlook on the very next model turn. Both manifest entries appear in Figure 11.

### C.3 Reading the change-manifest figures

The trajectories above track individual edits through individual tasks. The change-manifest carries each edit along with its predicted fixes, predicted regressions, and constraint level into Phase 3 of the next iteration, where the attribution check decides whether to keep or roll it back. One manifest figure is attached to each of the four winning rounds, all in the same Files / What changed / Failure pattern fixed / Predicted fixes layout. Figure 8 shows iteration 2’s prompt edit and shell-tool edit written together in the seed round. Figure 9 shows iteration 5’s prompt-and-descriptor rule and shell-guard installation that introduce the publish-state mechanism. Figure 10 shows iteration 6’s extension of the publish-state guard to script entrypoints and the introduction of the cross-step ExecutionRiskHintsMiddleware. Figure 11 shows iteration 8’s keep-and-improve patches that close the override-token loophole on the guard and promote middleware reminders into a FRAMEWORK note visible at the next model turn. Together the four figures cover three of the four constraint levels the evolve agent uses, namely prompt, tool implementation, and middleware, all written in the same JSON shape and all subject to the same automatic rollback if their predicted fixes do not appear.

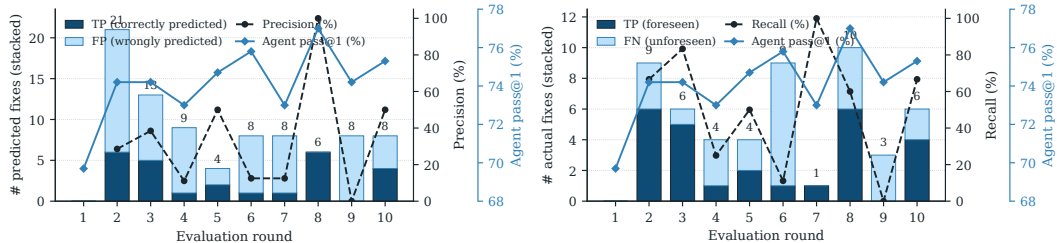


Figure 12: Per-round fix predictions. Left: precision. Right: recall. Bars decompose each denominator into TP versus FP or FN; lines overlay the metric and contemporaneous pass@1.

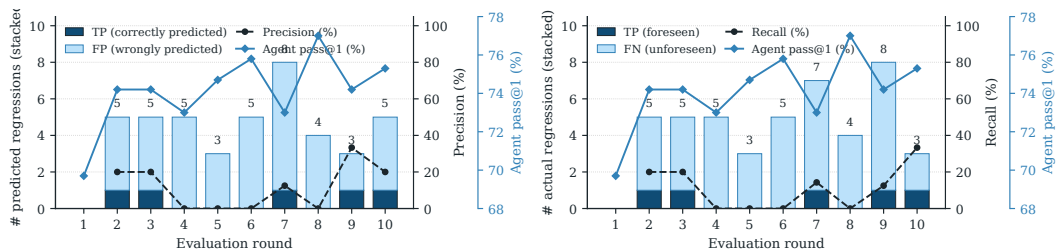


Figure 13: Per-round regression predictions. Left: precision. Right: recall. Same encoding as Fig. 12.

## D Per-round Self-attribution Breakdown

This appendix expands the aggregate self-attribution result of §4.4.2 with a per-round breakdown across the four fix/regression by precision/recall panels.

Figures 12 and 13 show the per-round breakdown across the four fix/regression by precision/recall panels. Bars decompose each denominator, predicted for precision and actual for recall, into deep-blue TP versus pale FP or FN; the dashed line traces the metric on the right-hand 0 to 100% axis, and the solid line shows contemporaneous pass@1. Fix-precision and fix-recall both swing from near-zero to near-saturation across rounds, so the evolve model’s causal attribution for its own improvements is informative if noisy. Regression predictions instead stay near the floor, below 25% on most rounds: across the 9 rounds the agent issued 43 unique regression predictions and only 5 landed, giving cumulative  $P = 11.6\%$ , while 40 regressions the agent did not foresee actually occurred, giving cumulative  $R = 11.1\%$ .

## E Broader Impact

This work introduces *agentic harness engineering* (AHE), a self-evolving loop in which a coding agent edits its own scaffolding (system prompt, tools, middleware, and long-term memory) from execution feedback. AHE lowers the human engineering cost of producing competitive coding agents: practitioners without dedicated harness teams can obtain higher pass@1 from the same base model and at lower per-trial token cost than prompt-only self-evolution baselines, which broadens access to capable coding assistants for academic groups, smaller organizations, and educational settings. Because the loop encodes recurring coordination patterns into tools, middleware, and memory rather than into ever-longer prompts, it reduces per-call compute and the associated energy footprint at inference time. It also enables rapid harness iteration that can keep pace with the cadence of base-model releases, so the surrounding scaffolding need not lag every new model. Negative societal impacts and the corresponding generalization hazards are discussed in §5.