EVOLAUDIT: A FRAMEWORK FOR MEASURING, INTERPRETING, AND IMPROVING SELF-EVOLVING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Self-evolving agents autonomously improve their capabilities by updating their components based on available feedback. They are typically evaluated by the final task performance. However, aggregate performance may hide performance drops on individual instances even when the overall score improves. Analyzing these performance declines provides fine-grained guidance for improvement. To fill this gap, we propose an audit framework (**EvolAudit**) that measures evolution, interprets performance changes, and provides guidance for improving self-evolving agents. During evolution, EvolAudit tracks instance-level changes between iterations, recording a gain when an instance fails in one iteration but succeeds in the next and a regression when the opposite occurs. To assess evolution quality, we define Evolution Efficiency as the proportion of gains among all gains and regressions. For evolution interpretation, we examine component updates at each iteration, and then relate them to instance-level gains/regressions to assess their contribution to performance changes. Building on this analysis, EvolAudit translates instance-level diagnostic insights into targeted component improvements to increase gains relative to regressions. We apply EvolAudit to harness evolution on terminal tasks and memory evolution on web tasks. Experimental results reveal frequent instance-level regressions throughout evolution and show that some updated components are rarely activated during execution. Guided by these findings, targeting previously untouched components improves final performance from 61.77% to 72.06%, showing that EvolAudit can effectively identify evolution failures and guide further improvement.

1 INTRODUCTION

Large language models enable agents to reason, use tools, and interact with external environments (Yao et al., 2023). These capabilities make agents useful for complex applications such as software engineering and web automation (Yang et al., 2024; Zhou et al., 2024). Such applications often involve changing tasks and environments, creating a need for agents that can adapt over time without frequent manual redesign. Self-evolving agents offer a promising direction by using task feedback to update components such as prompts, tools, memory, and agent workflows (Fang et al., 2025; Shinn et al., 2023; Zhao et al., 2024). Unlike static agents, they continue to modify these components across iterations, making the behavior and task performance change throughout evolution. Existing work usually evaluates self-evolution by comparing overall task performance across iterations (Zheng et al., 2026; Wang et al., 2026a). Such aggregate evaluation overlooks which individual instances regress during evolution and why these regressions occur.

Studying regressions can reveal capability losses during evolution, providing guidance for improvement. To fill this gap, we introduce EvolAudit, an instance-level evaluation framework for measuring and interpreting evolution, with the resulting analysis used to guide further improvement. EvolAudit records instance-level changes throughout evolution. It evaluates performance using task success rate and Evolution Efficiency, which captures the balance between gains and regressions. We develop Evolution Retention to measure stability by tracking the preservation of previously acquired capabilities. For interpretation assessment, EvolAudit links instance-level gains and regressions to component updates. It further checks whether the updates take effect during execution, helping ex-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

plain the observed performance changes. Based on these findings, we identify underexplored components and target them in later evolution iterations. We then re-evaluate the agent to test whether the targeted updates produce more gains than regressions.

We implement EvolAudit on two representative self-evolving settings: Agentic Harness Engineering (AHE) for harness evolution on Terminal-Bench 2.0 (Lin et al., 2026a) and ReasoningBank for memory evolution on WebArena Shopping (Ouyang et al., 2026). They represent two complementary modes of agent self-evolution: system-level and component-level adaptation. Across both settings, aggregate improvement is accompanied by frequent instance-level regressions, showing that final performance alone does not reflect the full evolution process. To understand these instance-level changes, we examine updates to system prompts, tool descriptions, tool implementations, middleware, skills, sub-agents, and long-term memory. This analysis reveals that harness updates concentrate on a small subset of these components, with some updated components remaining inactive during execution. Guided by these findings, targeting the previously untouched long-term memory component improves final evolution-set performance from 61.77% to 72.06%.

Our contributions are summarized as follows: (1) *Conceptually*, we highlight instance-level regression as an important aspect overlooked in existing self-evolution evaluation. (2) *Methodologically*, we propose EvolAudit to measure gains and regressions, interpret their links to component updates, and use the resulting findings to guide evolution improvement. (3) *Empirically*, we evaluate two self-evolving frameworks and reveal frequent regressions and limited component exploration during evolution. Guided by these findings, targeting a previously untouched component improves the final evolution-set success rate from 61.77% to 72.06%.

Our paper is organized as follows. Sec. 2 first reviews recent works on self-evolving methods and their evaluation.Building on this background, Secs. 3–4 present the self-evolving framework and the EvolAudit methodology. Sec. 5 then introduces the experimental results and analyzes instance-level evolution patterns. Sec. 6 summarizes the main findings and discusses the limitations of our study.

2 Related Work

Self-Evolving Methods. Self-evolving agents improve their behavior by learning from their own interaction experience. Agents use their execution feedback to update memory for knowledge retention, skills for procedural reuse, prompts for behavioral guidance. At the system level, they update harnesses for coordinating tools and execution. Early memory-evolving methods store task reflections (Shinn et al., 2023), while subsequent approaches distill reusable reasoning strategies (Ouyang et al., 2026) and evolve memory-management skills (Zhang et al., 2026a). Skill-evolving methods range from building executable skill libraries (Wang et al., 2023) to refining skills from interaction experience (Yang et al., 2026) and more recently to jointly evolving skills and agent policies through reinforcement learning (Xia et al., 2026). Prompt-evolving methods progress from mutation and crossover (Guo et al., 2024) to feedback-driven refinement of instructions and examples (Agarwal et al., 2025) and further toward trajectory-based reflection with Pareto-guided selection (Agrawal et al., 2026). Harness evolution enables joint updates across these components. They treat the agent scaffold as an editable system, using feedback to update components and keeping changes that improve performance (Lin et al., 2026b; Wei et al., 2026; Zhang et al., 2026b).

Evaluation of Self-Evolving Methods. Existing studies evaluate self-evolving agents mainly from long-term adaptation and experience reuse. Long-term evaluations examine whether agents can acquire, retain, and transfer skills across sequential tasks (Zheng et al., 2025). In addition to tracking performance and resource use over task streams (Jiang et al., 2026), some evaluations further assess adaptation to changing environments (Xu et al., 2026). Evaluations of experience reuse examine how accumulated knowledge supports procedural transfer across tasks (Gao et al., 2026). At the skill level, studies trace discovery, reuse, and refinement (Zhang et al., 2026c), while some researchers assess skill quality, execution trajectories, and task outcomes (Zhong et al., 2026). Memory-focused benchmarks complement this view by examining experience reuse within and across episodes (Wang et al., 2026b). Harness-level evaluations examine whether agents can construct and improve their own execution frameworks (Wu et al., 2026). Controlled studies further assess performance gains and generalization to held-out tasks under matched feedback (Wang et al., 2026a). Existing evaluations provide limited support for linking instance-level performance changes to component updates. EvolAudit tracks the instances across evolution iterations to measure gains,

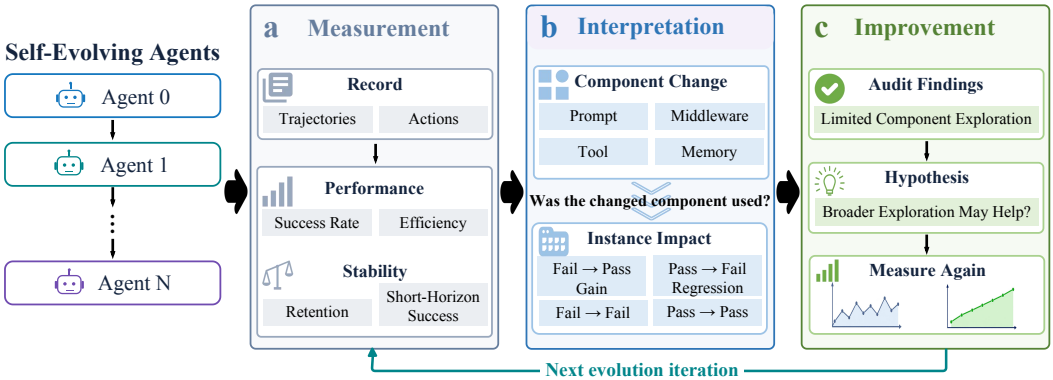

Figure 1: EvolAudit's evaluation workflow. (a) Measurement tracks performance, evolution efficiency, retention, and short-horizon continuation success. (b) Interpretation links component changes to instance-level outcomes. (c) Improvement uses diagnostic insights to guide targeted component evolution and evaluates its effects.

regressions, and retention. It connects these outcomes to records of component updates, supporting fine-grained diagnosis across different evolution methods.

3 Problem Formulation

3.1 Component Definitions

We represent a self-evolving framework as a backbone language model equipped with an evolving harness. The harness contains seven editable component types. System prompts define the agent's objectives and behavioral instructions. Tool descriptions specify interfaces and intended usage; tool implementations provide the executable code. Middleware manages requests, observations, and tool interactions. Skills encode reusable procedures as instructions, workflows, or code. Sub-agent configurations specify roles, instructions, and available tools. Long-term memory retains knowledge across task executions.

At evolution step t, the agent executes tasks with the current harness $\mathcal{H}_t$ and collects trajectories and feedback in $\mathcal{D}_t$. The evolution operator $\mathcal{E}$ uses this experience to update one or more harness components:

$$\mathcal{H}_{t+1} = \mathcal{E}\big(\mathcal{H}_t, \mathcal{D}_t\big). \tag{1}$$

An update can affect one or multiple components, while components outside its scope remain unchanged. Holistic harness evolution permits updates across all seven components, whereas component-specific frameworks restrict evolution to a designated subset. The backbone model parameters remain fixed throughout the evolution process considered in this work.

3.2 Instance-Level Evolution

Let $\mathcal{Q} = \{x_1, \ldots, x_N\}$ denote a fixed evaluation set that is evaluated at every evaluation point. Given the agent $\mathcal{A}^{(k)}$ at evolution step k, the performance of instance x_i is defined as

$$p_i^{(k)} = \mathrm{Eval}\Big(\mathcal{A}^{(k)}, x_i\Big), \tag{2}$$

where $\mathrm{Eval}(\cdot)$ is a task-specific evaluator. The resulting score can be binary, such as task success, or continuous, such as execution reward, exact-match score, or partial-credit score.

We characterize instance-level evolution using both consecutive and cumulative performance changes:

$$\Delta_i^{(k)} = p_i^{(k)} - p_i^{(k-1)}, \qquad \Delta_{i,\mathrm{cum}}^{(k)} = p_i^{(k)} - p_i^{(0)}. \tag{3}$$

The consecutive change records the difference between two adjacent evolution iterations, whereas the cumulative change compares an evolution iteration with the initial agent. Given a tolerance threshold τ, an instance is categorized as improved when $\Delta_i^{(k)} > \tau$, regressed when $\Delta_i^{(k)} < -\tau$, and unchanged otherwise. For the reported binary outcomes we use $\tau = 0$, yielding fail-to-pass, pass-to-fail, pass-to-pass, and fail-to-fail transitions.

4 METHODOLOGY

4.1 OVERVIEW

EvolAudit takes an initial agent $\mathcal{A}^{(0)}$, an evolution operator Φ, an ordered experience stream partitioned into blocks $\{\mathcal{B}_k\}_{k=1}^{K}$, an evaluation set $\mathcal{Q}$, and a task-specific evaluator. It returns the evolution iteration–instance performance matrix $\mathbf{P}$, adjacent changes $\mathbf{\Delta}$, and diagnostic records $\mathcal{G}$ containing trajectories, component snapshots, updates, and usage evidence:

$$(\mathcal{A}^{(0)}, \Phi, \{\mathcal{B}_k\}_{k=1}^{K}, \mathcal{Q}, \mathrm{Eval}) \longrightarrow (\mathbf{P}, \mathbf{\Delta}, \mathcal{G}). \tag{4}$$

We first measure evolution quality by evaluating the same instances at each recorded evaluation point (Figure 1(a)). Success rate and Evolution Efficiency measure performance, and Evolution Retention tracks the preservation of earlier capabilities. Observed clade success describes short-horizon continuation quality. Additional evaluation points capture individual updates within a block. Each comparison specifies the task set, evaluation interval, and sampling budget.

Interpretation connects instance-level gains, regressions, and unchanged outcomes to component changes (Figure 1(b)). Activation and retrieval records show which components reach each instance, and execution traces reveal how they are used. These records help identify possible explanations for observed performance changes.

The resulting diagnostic findings guide improvement (Figure 1(c)). Specifically, limited component exploration motivates targeted evolution of underexplored components.

4.2 INSTANCE-LEVEL MEASUREMENT

For binary instance outcomes, G_k counts failure-to-success transitions and R_k counts success-to-failure transitions between evolution iterations $k-1$ and k. Evolution Efficiency (EE) measures how often outcome changes are beneficial. To moderate extreme estimates based on few transitions, we use symmetric pseudo-count smoothing:

$$\widetilde{\mathrm{E}}_k = \frac{G_k + \sigma/2}{G_k + R_k + \sigma}, \qquad \sigma = 1. \tag{5}$$

This adds equal pseudo-counts to gains and regressions, shrinking the estimate toward 0.5 when few changes are observed. Values above 0.5 indicate more gains than regressions, while values below 0.5 indicate the opposite. When both counts are zero, the score equals 0.5 solely by the smoothing convention; we explicitly flag these evolution iterations as having no observed changes. We report success rates, G_k, and R_k alongside smoothed EE to distinguish the direction of evolution from the extent of change. For pooled EE, we first aggregate gains and regressions across the relevant comparisons and then apply the same smoothing. We also report pooled EE separately for within-block, full-block, and adjacent evaluation point comparisons.

Inspired by backward transfer (Lopez-Paz & Ranzato, 2017), we define Evolution Retention (ER) on a fixed instance set without requiring distinct sequential learning-task groups. Let $\mathcal{S}_{<k} = \{i : \exists j < k,\ p_i^{(j)} = 1\}$ contain successful instances at any earlier recorded evolution iteration. Then

$$\mathrm{R}_k = \frac{\sum_{i \in \mathcal{S}_{<k}} p_i^{(k)}}{|\mathcal{S}_{<k}|}, \qquad |\mathcal{S}_{<k}| > 0. \tag{6}$$

Once an instance succeeds, it enters the reference set at the next evaluation point. An instance counts as retained if it succeeds at the current point, even after an earlier failure. Tracking all evaluation points shows whether success is maintained. We also examine how performance develops over subsequent updates. Observed clade success captures this short-horizon continuation quality by

combining results from a parent agent and its evaluated descendants. For a parent evolution iteration a, let $\mathcal{C}_H^{\text{obs}}(a)$ contain a and its next H evaluated descendants under the chosen evaluation point schedule. We define observed clade success as

$$\mathrm{C}_H(a) = \frac{\sum_{b \in \mathcal{C}_H^{\text{obs}}(a)} s_b}{\sum_{b \in \mathcal{C}_H^{\text{obs}}(a)} n_b}, \tag{7}$$

where s_b and n_b denote the numbers of successful and total evaluations at evolution iteration b, respectively. We use $H = 2$, covering three evolution iterations in each window.

4.3 Evolution Interpretability

We define evolution interpretability as the extent to which evaluation explains how changes to evolving components affect individual instances. We control for sampling variability through matched evaluation settings and repeated trials, excluding transitions that cannot be reliably distinguished from sampling fluctuations. Our analysis therefore focuses on reproducible changes associated with component evolution and proceeds from two complementary perspectives.

(1) Component-level analysis. We examine which components are updated, what changes are introduced, and how the updated components are used during execution. For harness evolution, we investigate both individual updates and interactions among components. For dedicated memory-evolution frameworks, we inspect memory construction, retrieval, and utilization: whether relevant knowledge is recorded, whether suitable entries are retrieved, and whether the agent applies them appropriately. Where feasible, we compare the original and updated versions of a component while holding other components and evaluation conditions fixed. These comparisons help attribute performance changes to specific updates. ***(2) Instance-level transitions.*** After accounting for sampling variability, we analyze fail-to-pass, pass-to-fail, pass-to-pass, and fail-to-fail transitions across evolution iterations. For each instance, we connect its transition to the components accessed and applied during execution. In memory evolution, for example, we examine whether a fail-to-pass transition depends on retrieving and correctly applying an evolved memory, using memory removal or substitution tests to strengthen attribution. Similarly, pass-to-fail transitions can reveal harmful interference from misleading or incorrectly applied knowledge. Unchanged outcomes help identify updates that are unused, redundant, or insufficient. Together, these analyses explain where component evolution helps, where it harms, and where it has little effect.

4.4 Evolution Improvement

To guide targeted evolution, we track component changes at each evolution step and summarize their update frequencies. The recorded update rationales help explain the agent's component choices. This analysis identifies components that receive little attention and helps explain why updates concentrate on particular components.

Based on this analysis, we select an underexplored component $c^\star$ and direct further evolution toward it:

$$\mathcal{A}_{\text{target}}^{(k)} = \Phi_{c^\star}\left(\mathcal{A}_{\text{target}}^{(k-1)}, \mathcal{B}_k\right), \tag{8}$$

where $\mathcal{B}_k$ is the evolution block at step k. The mechanism $\Phi_{c^\star}$ is instructed to focus its updates on the selected component.

5 Experiments

5.1 Experimental Setup

Datasets. *Terminal-Bench 2.0* contains 89 terminal tasks drawn from real-world workflows, each with a task-specific environment and automated verification (Merrill et al., 2026). We use all 89 tasks in the full-set run and 68 evolution tasks with 17 held-out tasks in the split run. *WebArena Shopping* is the e-commerce subset of WebArena, covering multi-step browser tasks such as product search and order lookup (Zhou et al., 2024). We use 144 training tasks and 36 held-out tasks.

Table 1: Endpoint performance and transition statistics. $\bar{G}$ and $\bar{R}$ average counts over nine adjacent evolution iterations for AHE and 24 for ReasoningBank. EE uses pooled counts with smoothing. AHE reports rollout success; ReasoningBank reports pass@1.

Method	Split	Initial (%)	Last (%)	$\bar{G}$	$\bar{R}$	EE (%)
AHE	Full-set	57.58	62.92	6.56	6.22	51.29
AHE	Evolution	53.31	61.77	4.22	3.33	55.80
AHE	Held-out	41.18	44.12	0.89	1.00	47.22
ReasoningBank	Held-out	66.67	72.22	1.17	1.08	51.82

Settings. *For harness evolution,* we evaluate the Agentic Harness Engineering (AHE) framework (Lin et al., 2026b) with Qwen3.7-Plus as the backbone model. AHE runs for ten evolution rounds, evaluating the current evolution iteration before each update with two rollouts per task. To reduce the impact of sampling variability, we repeat the full ten-round experiment multiple times and report the mean results. Evolution-set results therefore cover evolution iterations I0–I9. After evolution, we evaluate frozen evolution iterations on the held-out tasks. The full-set run follows the original AHE across all rounds.

For task-level transition analysis, success requires two successful rollouts on the evolution set. Further details appear in Appendix A.1. *For memory evolution,* we evaluate the ReasoningBank framework (Ouyang et al., 2026) with Gemini 3.1 Pro Preview as the backbone model. During training, parallel MaTTS generates five candidate trajectories per task, which ReasoningBank uses to evolve memory. We process tasks in blocks of 12 and evaluate the agent immediately before and after each block's final update. Including the initial evaluation, this yields 25 evaluation points from I0 to I144. Held-out evaluation uses one rollout per task and reports official pass@1, with the memory bank and embedding cache frozen. Further details appear in Appendix A.2.

5.2 RQ1: How Well Do Self-Evolving Frameworks Evolve?

Overall evolution performance. Table 1 summarizes endpoint performance and instance-level changes. Full-set AHE improves its mean rollout success rate from 57.58% to 62.92%. However, the first run records 59 gains and 56 regressions across nine adjacent evaluation intervals, yielding a pooled EE of 51.29%. ReasoningBank with Gemini shows a similar contrast. Held-out success increases from 66.67% at I_0 to 72.22% at I_{144}. The endpoint comparison contains three gains and one regression, giving an EE of 70.00% (paired exact McNemar test, $p = 0.625$). Across all 24 adjacent evaluation points, however, 28 gains and 26 regressions yield a pooled EE of 51.82%. Endpoint comparisons therefore provide an incomplete account of the evolution process, as they reveal little about when individual instances regress.

Evolution dynamics and retention. Figures 2 and 3 examine intermediate performance and the preservation of historical successes. For split AHE, mean evolution-set success increases from 53.31% to 61.77%, while mean held-out success increases more modestly, from 41.18% to 44.12%. The held-out endpoint changes also differ across runs: the first run declines by 11.76 percentage points, whereas the second improves by 17.65 points. Therefore, improved performance on the evolution set does not necessarily translate into better held-out performance. Between I_5 and I_6, evolution-set success rises from 59.93% to 61.03%, while held-out success falls from 48.53% to 47.06%. Figure 2(b) shows substantial variation in EE across updates. On the evolution set, gains outnumber regressions in six of nine updates, but EE falls to 25.00% between I_6 and I_7. In the following update, evolution-set EE rebounds to 68.18%, while held-out EE drops to 12.50%. Thus, even an update with substantially more gains than regressions on the evolution set can coincide with deteriorating held-out performance. Historical successes are not fully retained either: at I_9, mean any-success ER is 80.84% on the evolution set, 71.80% on the held-out set, and 80.44% in the separate full-set experiments. For ReasoningBank, EE for within-block final updates and full-block comparisons is 52.08% and 53.23%, respectively. Figure 3(b) shows that gains during memory evolution can be short-lived.Across the 24 evaluation intervals, gains outnumber regressions in nine. Three intervals contain no outcome changes. At I_{144}, ReasoningBank retains 26 of the 32 instances solved at earlier evaluation points, giving an ER of 81.25%. Although the overall score improves,

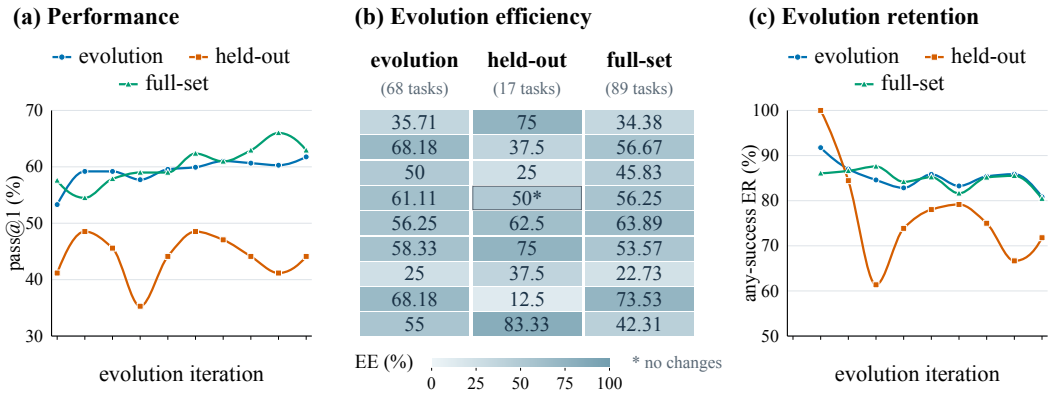

Figure 2: (a) Rollout success rates across evaluation settings. (b) Smoothed Evolution Efficiency (EE) between adjacent evolution iterations. (c) Evolution Retention (ER), measuring the proportion of previously solved instances that remain successful.

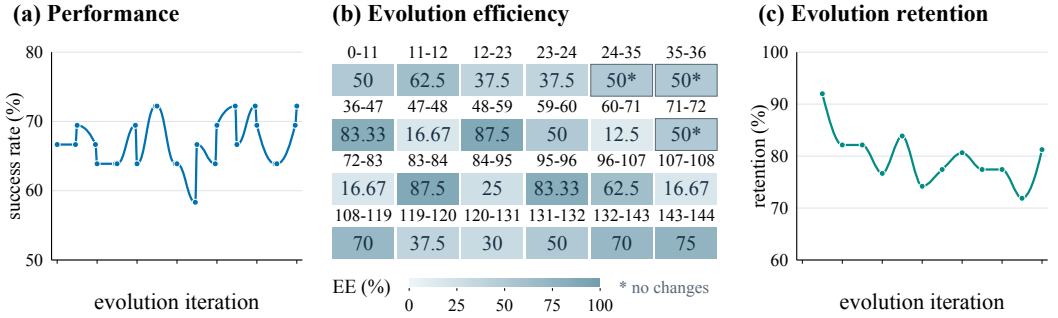

Figure 3: Evaluation of ReasoningBank with Gemini. (a) Pass@1 at 25 evaluation points. (b) Smoothed Evolution Efficiency (EE) between adjacent evaluation points. (c) Evolution Retention (ER), measuring the preservation of previously successful instances.

six instances solved at earlier evaluation points fail in the final evaluation. Across these experiments, overall performance gains coexist with failures on previously solved instances.

Observed continuation quality. Table 2 compares immediate performance with observed clade success over three evaluation points. In full-set AHE, I_5 has higher mean immediate success than I_6 (62.36% versus 60.96%), but lower clade success (62.08% versus 63.30%). For ReasoningBank, I_{60} and I_{84} have different immediate success rates, yet both yield 67.59% over their respective three-point windows. Based on these results, immediate performance therefore does not fully characterize the quality of an observed continuation.

Table 2: Three-point clade success (%).

Parent	Current	Next 1	Next 2	Clade
AHE/full, I_5	62.36	60.96	62.92	62.08
AHE/full, I_6	60.96	62.92	66.01	63.30
ReasoningBank, I_{60}	72.22	63.89	66.67	67.59
ReasoningBank, I_{84}	66.67	69.44	66.67	67.59

Full accounting appears in Appendix F. Together, these results show that assessing evolution quality requires tracking performance changes, stability, retention, and transfer throughout the process, alongside the final score.

5.3 RQ2: What Patterns Emerge During Evolution?

Component updates do not ensure exposure. Across both frameworks, updated components are not necessarily encountered during execution. In both systems, some evolved components provide no direct benefit during evaluation because they remain unused: middleware stays inactive, and

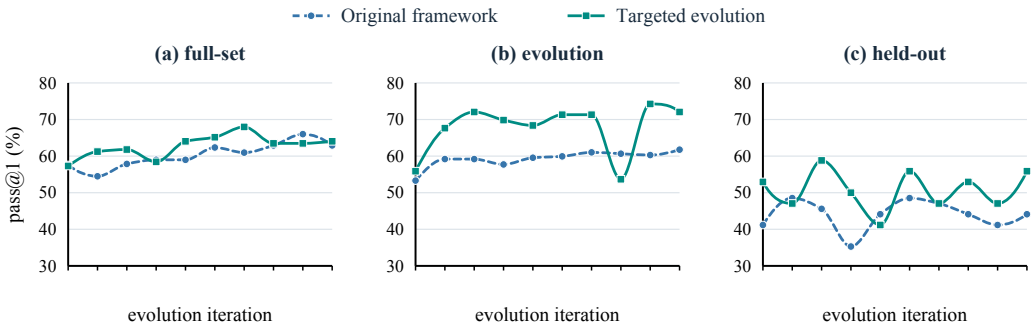

Figure 4: Comparison of the original framework and targeted component evolution on (a) the full-set run, (b) the split evolution set, and (c) the frozen test set.

stored memories go unretrieved. In split AHE, the first middleware update never activates across its 136 evolution-set rollouts. In ReasoningBank, the final bank contains 144 task-level entries, but only 17 are retrieved across the 36 held-out instances.

Guidance exposure does not ensure adherence. Across 684 adjacent instance pairs in the I11–I144 ReasoningBank audit: 40 of 41 outcome flips occur with unchanged memory IDs and injected text. Conversely, only one of 71 retrieval changes coincides with an outcome flip (Table 3). Even with the same retrieved memory, agents can take different actions and produce different outcomes. Execution traces show that agents sometimes deviate from the guidance they receive. Table 4 shows that system prompts are updated most frequently, followed by tool descriptions and middleware, with all three associated with both gains and regressions. The associated gains involve handling previously missed inputs or test cases, completing cancellation, and improving dependency scheduling. Regressions include changes to protected data, exact-output mismatches, and poor performance despite promising local test results. Instance traces provide more specific explanations for these changes. In `polyglot-c-py`, cleanup guidance aligns with successful removal of temporary compilation files from the delivery directory. However, in `sqlite-with-gcov`, temporary-directory builds and cleanup leave the verifier unable to locate the required coverage files. The strengthened cleanup instruction conflicts with this task's artifact requirements. In ReasoningBank, WebArena instance 240 requires finding the most expensive product in the competitive swimwear category. Between I36 and I47, retrieval changes from page-aggregation guidance to hierarchical category-navigation guidance. At I47, the agent follows the category path, sorts the products, and succeeds. At I48, it receives the same memory but switches to keyword search and fails. The two cases share an exposure–adherence gap, with different execution failures: unnecessary cleanup in AHE and departure from a relevant navigation procedure in ReasoningBank. Detailed cases appear in Appendix D.

Table 3: Retrieval–outcome correspondence.

Memory	Same	Flipped	Total
Unchanged	573	40	613
Changed	70	1	71
Total	643	41	684

5.4 RQ3: Can the Identified Patterns Improve Evolution?

Imbalanced Component Exploration. The component audit reveals uneven exploration of the available update space.As shown in Figure 5, although AHE permits changes to seven component types, both runs modify only system prompts, tool descriptions, and middleware, leaving four component types unchanged. The recorded update rationales reveal a conservative preference for modifying familiar components. Evolving agents favor simpler revisions and avoid adding complexity to reduce perceived regression risk. As a result, potentially useful changes remain unexplored, even with a broad range of editable components.

Targeted Component Evolution. Based on these findings, we hypothesize that evolving harder-to-modify components alongside frequently updated ones could improve performance. We investigate targeted evolution of long-term memory, an underexplored component in the

analyzed runs. The objective is not to update all seven components equally, but to explore potentially useful changes overlooked by the original update-selection process. Figure 4 compares the targeted variant with the original AHE framework. At I_9, targeted evolution reaches 72.06% on the evolution set, compared with 61.77% for AHE. Their improvements over their respective initial scores are 16.18 and 8.46 percentage points. Targeted evolution nevertheless remains non-monotonic, including a temporary decline to 53.68% at I_7. This contrast demonstrates that higher overall performance can coexist with substantial intermediate regressions. In the separate full-set evaluation, targeted evolution reaches 64.04%, compared with 62.92% for AHE. On the held-out set, targeted evolution reaches 55.88% at I_9, exceeding AHE's mean score of 44.12% by 11.76 percentage points. Targeted component evolution also achieves higher pooled EE. On the evolution set, pooled EE increases from 55.80% for AHE to 58.13%. On the held-out set, it rises from 47.22% to 54.17%, shifting the balance from more regressions to more gains. These results show that targeted evolution improves the proportion of beneficial changes across the evaluated trajectory. Overall, these results suggest that diagnosing component-selection patterns can improve both task performance and evolution efficiency.

Table 4: Component updates and associated transitions.

Component	Updates	Gains	Regressions
System prompt	9	9	7
Tool description	5	5	3
Middleware	2	3	3

5.5 Discussion

Across harness and memory evolution, a recurring pattern is that component updates do not consistently translate into useful behavior. In AHE, this appears as inactive middleware and improvements on evolution tasks that do not transfer to held-out tasks. In memory evolution, outcomes can fluctuate despite unchanged retrieved content. Evaluation should track which components change, when they are accessed, and how agents use them during execution. We recommend checking component activation before accepting updates and verifying whether retrieved memories are relevant and used during execution. Targeted component evolution provides a practical application of these insights. In our runs, it achieves stronger adaptation and higher final held-out performance than the original framework.

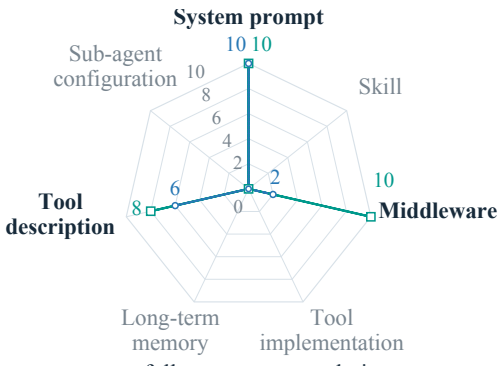

Figure 5: Component update frequencies over ten evolution rounds.

6 Conclusion

We introduced EvolAudit, an instance-level protocol for evaluating self-evolving agents. It combines performance, evolution efficiency, evolution retention, and observed continuation quality. Together, these measurements connect aggregate performance to the behavior of individual instances throughout evolution. Our experiments on harness and memory evolution reveal recurring regressions and gaps between component updates and their effective use. Harness updates concentrate on a small subset of components, motivating targeted evolution of underexplored components. In our recorded runs, targeted evolution achieves stronger adaptation and higher final held-out performance than the original framework. These findings provide initial evidence for the practical value of component-level diagnosis. They also highlight the importance of component selection when designing evolution mechanisms. The study has several limitations. Our findings are based on a limited range of experimental settings. Broader evaluation is needed to assess their generalizability. Future work should extend the evaluation to more frameworks and tasks.

REFERENCES

Eshaan Agarwal, Raghav Magazine, Joykirat Singh, Vivek Dani, Tanuja Ganu, and Akshay Nambi. PromptWizard: Optimizing prompts via task-aware, feedback-driven self-evolution. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 19974–20003. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.1025. URL https://aclanthology.org/2025.findings-acl.1025/.

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2507.19457.

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, Zhaochun Ren, Nikos Aletras, Xi Wang, Han Zhou, and Zaiqiao Meng. A comprehensive survey of self-evolving AI agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv preprint arXiv:2508.07407*, 2025.

Xingze Gao, Chuanrui Hu, Hongda Chen, Pengfei Yao, Zhao Wang, Yi Bai, Zhengwei Wu, Yunyun Han, Xiaofeng Cong, Jie Gui, Yafeng Deng, and Teng Li. EvoAgentBench: Benchmarking agent self-evolution via ability transfer. *arXiv preprint arXiv:2607.05202*, 2026. URL https://arxiv.org/abs/2607.05202.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=ZG3RaNIsO8.

Sihang Jiang, Lipeng Ma, Zhonghua Hong, Keyi Wang, Zhiyu Lu, et al. SEA-Eval: A benchmark for evaluating self-evolving agents beyond episodic assessment. *arXiv preprint arXiv:2604.08988*, 2026. URL https://arxiv.org/abs/2604.08988.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Xuanjing Huang, Hang Yan, Zhenhua Han, and Tao Gui. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026a. URL https://arxiv.org/abs/2604.25850.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic Harness Engineering: Observability-driven automatic evolution of coding-agent harnesses, 2026b. URL https://arxiv.org/abs/2604.25850.

David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning. In *Advances in Neural Information Processing Systems*, volume 30, 2017. URL https://papers.nips.cc/paper_files/paper/2017/hash/f87522788a2be2d171666752f97ddebb-Abstract.html.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=jL7fwchScm.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

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, 2023. URL https://arxiv.org/abs/2305.16291.

Yike Wang, Huaisheng Zhu, Zhengyu Hu, Yige Yuan, Zhengyu Chen, Shakti Senthil, Hannaneh Hajishirzi, Yulia Tsvetkov, Pradeep Dasigi, and Teng Xiao. Rethinking the evaluation of harness evolution for agents. *arXiv preprint arXiv:2607.12227*, 2026a. URL https://arxiv.org/abs/2607.12227.

Yuyao Wang, Zhongjian Zhang, Mo Chi, Kaichi Yu, Yuhan Li, Miao Peng, Bing Tong, Chen Zhang, Yan Zhou, and Jia Li. EvoMemBench: Benchmarking agent memory from a self-evolving perspective. *arXiv preprint arXiv:2605.18421*, 2026b. URL https://arxiv.org/abs/2605.18421.

Tianxin Wei, Zhan Shi, Minhua Lin, Bing He, Zewen Liu, Yisi Sang, Yuanchen Bei, Xuying Ning, Jiaru Zou, Ting-Wei Li, Xiao Lin, Yanjun Zhao, Chi Wang, Benoit Dumoulin, Dakuo Wang, Jingrui He, and Hanqing Lu. Evo-Harness: Context-to-harness skill compilation for self-evolving agents. *arXiv preprint arXiv:2608.15071*, 2026. URL https://arxiv.org/abs/2608.15071.

Yuhao Wu, Jingyuan Zhang, Jiajun Shi, Xinping Lei, Qingshui Gu, Yuxuan Zhang, Zexuan Wang, Chen He, Chen Huang, Maojia Song, Zhiyuan Zeng, Shaowen Wang, Jinkai Liu, Yunfeng Shi, Jiaheng Liu, Shen Yan, Wenhao Huang, Ge Zhang, and Wenxuan Zhang. HarnessDev: Can LLMs create and evolve their own agent harness?, 2026. URL https://arxiv.org/abs/2609.01437.

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. *arXiv preprint arXiv:2602.08234*, 2026. URL https://arxiv.org/abs/2602.08234.

Jundong Xu, Qingchuan Li, Jiaying Wu, Yihuai Lan, Shuyue Stella Li, Huichi Zhou, Bowen Jiang, Lei Wang, Jun Wang, Anh Tuan Luu, et al. EvoArena: Tracking memory evolution for robust llm agents in dynamic environments. *arXiv preprint arXiv:2606.13681*, 2026. URL https://arxiv.org/abs/2606.13681.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Yutao Yang, Junsong Li, Qianjun Pan, Bihao Zhan, Yuxuan Cai, Lin Du, Jie Zhou, Kai Chen, Qin Chen, Xin Li, Bo Zhang, and Liang He. AutoSkill: Experience-driven lifelong learning via skill self-evolution. *arXiv preprint arXiv:2603.01145*, 2026. URL https://arxiv.org/abs/2603.01145.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03629.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. MemSkill: Learning and evolving memory skills for self-evolving agents. *arXiv preprint arXiv:2602.02474*, 2026a. URL https://arxiv.org/abs/2602.02474.

Yifan Zhang, Yutong Dai, Juntao Tan, Luyu Yang, Rishi Mullur, Thai Hoang, Zhiyuan Hu, James Zhu, Phil Mui, Silvio Savarese, Ran Xu, and Zeyuan Chen. DarwinX: Evolving agent harnesses through natural selection, 2026b. URL https://arxiv.org/abs/2608.07545.

Ziao Zhang, Kou Shi, Shiting Huang, Avery Nie, Yu Zeng, Yiming Zhao, Zhen Fang, Qishen Su, Haibo Qiu, Wei Yang, et al. SkillFlow: Benchmarking lifelong skill discovery and evolution for autonomous agents. *arXiv preprint arXiv:2604.17308*, 2026c. URL https://arxiv.org/abs/2604.17308.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936.

Congjie Zheng, Chuanyi Xue, Bin Liang, Jun Yang, and Changshui Zhang. SEAGym: An evaluation environment for self-evolving LLM agents. *arXiv preprint arXiv:2606.17546*, 2026.

Junhao Zheng, Xidi Cai, Qiuke Li, Duzhen Zhang, ZhongZhi Li, Yingying Zhang, Le Song, and Qianli Ma. Lifelongagentbench: Evaluating llm agents as lifelong learners. *arXiv preprint arXiv:2505.11942*, 2025. URL https://arxiv.org/abs/2505.11942.

Shanshan Zhong, Yi Lu, Jingjie Ning, Yibing Wan, Lihan Feng, Yuyi Ao, Leonardo F. R. Ribeiro, Markus Dreyer, Sean Ammirati, and Chenyan Xiong. SkillLearnBench: Benchmarking continual learning methods for agent skill generation on real-world tasks. *arXiv preprint arXiv:2604.20087*, 2026. URL https://arxiv.org/abs/2604.20087.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=rmiwIL98uQ.

A Additional Experimental Details

This appendix presents the experimental configurations and detailed evidence for component concentration, instance-level regressions, and capability retention in AHE and ReasoningBank. Harness evolution uses *evolution iteration*; memory evolution uses *evaluation point*. In both settings, I_0 denotes the initial agent. AHE success and retention summaries average two experiments; component records and transition counts use the first. ReasoningBank results cover the Gemini memory-evolution experiment.

A.1 Harness Evolution: Configuration and Schedule

Full-set and split AHE start from a minimal coding agent with a system prompt and one registered shell tool. Middleware, skills, sub-agents, and runtime long-term memory are initially empty. Evolution has access to all seven component types. File differences and runtime registration records identify accepted changes.

Table 5: AHE configuration and evaluation coverage per run.

Setting	Full-set AHE	Split AHE
Tasks	89 full-set tasks	68 evolution / 17 held-out tasks
Split seed	–	42
Agent / debugger / evolution model	Qwen3.7-Plus	Qwen3.7-Plus
Evolution updates	10	10
Rollouts per task	2	2
Rollout concurrency	20	20
Task / sandbox timeout	3,600 s	3,600 s
Full-set or evolution-set rollouts	1,780	1,360
Main-text evolution iterations	I_0–I_9	I_0–I_9

At update t, the agent executes the evolution tasks with the state at evolution iteration I_{t-1}. The debugger analyzes the resulting trajectories, and the evolution mechanism produces the state at I_t. Ten updates therefore provide ten pre-update measurements, from I_0 to I_9. The split-run initial state is recorded as NexAU0′.

Held-out evaluations use frozen agent states after the evolution sequence is complete, with their results stored separately from the feedback used for updates. The first split experiment includes an additional held-out evaluation at I_{10}. The main-text comparisons and two-run means use the common range I_0–I_9. Full-set AHE evaluates adaptation on the same 89 tasks that provide evolution feedback.

Each rollout uses a fresh E2B sandbox. Debugger concurrency is 16, with a 600-second task timeout and up to three retries. A local tracer stores raw and cleaned trajectories. The full-set configuration uses agent temperature 0.7, top-p 0.95, a 200,000-token context limit, a 32,000-token generation limit, and a maximum of 300 turns.

A.2 Memory Evolution: Construction and Evaluation

ReasoningBank uses parallel MaTTS with Gemini 3.1 Pro Preview for trajectory generation, selection, memory construction, and evaluation. The WebArena Shopping subset contains 180 tasks, divided by intent template into 144 training and 36 held-out instances. The two partitions contain 38 and 8 disjoint template groups, respectively. Every evaluation point uses the same 36 held-out instances.

Evaluation point I_x represents the agent after x training instances. Training proceeds in blocks of 12. Evaluation occurs before and after each block's final update: I_{11}/I_{12}, I_{23}/I_{24}, and so forth through I_{143}/I_{144}. Point I_0 reuses 36 memory-free baseline records from the same model, seed, and task split; the remaining 24 evaluations each execute the 36 held-out instances.

Task selection uses seeded SHA-256 ordering and an exact subset-sum procedure over intent templates. The BrowserGym/ReAct agent uses temperature 0.7 and a maximum of 30 steps. Five

Table 6: ReasoningBank configuration for Gemini memory evolution.

Setting	Configuration
Framework	ReasoningBank with parallel MaTTS
Backbone	Gemini 3.1 Pro Preview
Training candidates per task	5
Evaluation rollouts per instance	1
Seed	42
Training / held-out instances	144 / 36
Evaluation points	25, from I_0 to I_{144}
Evaluation records	900: 36 baseline records and 864 subsequent records
Final memory bank	144 task entries containing 346 structured items
Embedding / retrieval	3,072 dimensions; cosine top-1 retrieval

independent browser replicas generate the training candidates. A same-model selector at temperature 0 reads their trajectory text and returns a candidate index. Memory construction independently contrasts all five trajectories at temperature 1.0 and commits one schema-valid task entry.

Each entry contains one to five items with `Title`, `Description`, and `Content` fields. The embedding model is `gemini-embedding-001`; retrieval selects the most similar task-level entry by cosine similarity. The memory bank starts empty. The local seed fixes browser and numerical-library randomness.

A.3 Frozen Evaluation and Execution Records

Memory evaluation uses read-only bank and embedding-cache files, disabled update flags, and a training allowlist separate from the test IDs. All 24 memory-enabled evaluation points have matching bank and cache hashes before and after testing.

Diagnostic records link each instance's outcome and action–observation trajectory to retrieved entry IDs, injected text, component changes, and activation records. After I_{120}, Shopping execution moves to a KVM environment while retaining the task manifest, model settings, and saved memory state.

B Scoring and Temporal Aggregation

B.1 Outcome Definitions

AHE rollout success uses all scheduled executions as its denominator, counting timeouts and verifier or sandbox exceptions as unsuccessful. Strict task success requires both rollouts to pass. Tasks with two exceptions form a separate state; all other tasks are failures. Full-set and evolution-set EE count transitions between strict pass and fail states. Held-out EE and AHE retention use any-success, requiring at least one successful rollout. WebArena uses the official binary task reward, including answer and final-URL requirements.

B.2 Evaluation Intervals and Efficiency

Memory evaluation separates final-update effects from full-block changes. Within-block comparisons pair I_{11}/I_{12}, I_{23}/I_{24}, and the final update in each subsequent block. Full-block comparisons pair successive 12-task boundaries, beginning with I_0/I_{12}. Adjacent-point comparisons follow $I_0, I_{11}, I_{12}, I_{23}, I_{24}, \ldots, I_{144}$, alternating 11-instance and single-instance intervals. Each schedule is pooled separately.

EE adds 0.5 to gains and regressions (Equation 5). Pooled EE applies this smoothing once to the summed counts. The value $50^{*}\%$ identifies zero observed changes. Counts represent transition events, including repeated changes on the same instance.

C Additional Harness Results and Component Analysis

C.1 Evolution-Iteration Results

Mean evolution-set success rises from 53.31% to 61.77%, and full-set success rises from 57.58% to 62.92% (Table 7). Means average the recorded run-level percentages at each iteration. ER first constructs each run's historical success set, then averages the resulting ratios.

Table 7: AHE success rate (SR) and any-success retention (ER): two-run means, in percent.

Iteration	Evo. SR	Held-out SR	Full SR	Evo. ER	Held-out ER	Full ER
I_0	53.31	41.18	57.58	–	–	–
I_1	59.20	48.53	54.49	91.75	100.00	86.06
I_2	59.19	45.59	57.87	87.02	84.45	86.61
I_3	57.72	35.29	58.99	84.60	61.37	87.59
I_4	59.56	44.12	58.99	82.84	73.87	84.17
I_5	59.93	48.53	62.36	85.84	78.03	85.20
I_6	61.03	47.06	60.96	83.26	79.17	81.62
I_7	60.66	44.12	62.92	85.39	75.00	85.16
I_8	60.29	41.18	66.01	85.88	66.67	85.51
I_9	61.77	44.12	62.92	80.84	71.80	80.44

The first-run analysis links component changes and instance transitions to 1,780 full-set, 1,360 evolution-set, and 374 held-out rollouts (Table 8), plus 20 activation checks. Held-out measurements extend through I_{10}.

Table 8: First-run rollout results. P/F/E denotes pass/fail/exception counts; any/2 counts tasks passed at least once.

Iteration	Full-set P/F/E	Evolution P/F/E	Held-out P/F/E	Held-out any/2
I_0	108/56/14	72/32/32	17/10/7	9/17
I_1	97/57/24	73/30/33	18/8/8	10/17
I_2	106/60/12	76/31/29	13/11/10	9/17
I_3	103/56/19	74/32/30	12/10/12	8/17
I_4	103/64/11	75/30/31	16/11/7	8/17
I_5	115/59/4	75/26/35	16/10/8	9/17
I_6	114/58/6	76/27/33	16/8/10	10/17
I_7	110/62/6	74/32/30	15/11/8	9/17
I_8	121/51/6	80/22/34	11/15/8	6/17
I_9	112/61/5	76/21/39	13/9/12	8/17
I_{10}	–	–	13/9/12	7/17

First-run transitions reveal substantial gains and regressions throughout evolution (Table 9). Over I_0–I_9, full-set AHE has 59 gains and 56 regressions (51.29% pooled EE), the evolution set has 38 gains and 30 regressions (55.80%), and the held-out set has eight gains and nine regressions (47.22%). Including the supplementary held-out transition to I_{10} gives eight gains and ten regressions (44.74%). These counts expose the instance-level turnover hidden by endpoint scores.

C.2 Component Update Frequencies

Accepted updates concentrate on three of the seven editable component types: system prompts, tool descriptions, and middleware.

Accepted changes include additions, edits, deletions, and rollbacks. Full-set AHE adds middleware in update 1 and removes it in update 2; later changes focus on prompt rules and tool descriptions. Split AHE changes prompts and middleware in every update, and tool descriptions in every update except 6 and 8. Description and input-schema edits are distinct from tool-implementation edits. Runtime memory evolution requires a registered memory component and an accepted change to it.

Table 9: First-run gains/regressions and smoothed EE (percent).

Transition	Full-set: strict	Evolution: strict	Held-out: any/2
$I_0 \to I_1$	5/10; 34.38	2/4; 35.71	1/0; 75.00
$I_1 \to I_2$	8/6; 56.67	7/3; 68.18	1/2; 37.50
$I_2 \to I_3$	5/6; 45.83	4/4; 50.00	0/1; 25.00
$I_3 \to I_4$	4/3; 56.25	5/3; 61.11	0/0; 50.00*
$I_4 \to I_5$	11/6; 63.89	4/3; 56.25	2/1; 62.50
$I_5 \to I_6$	7/6; 53.57	3/2; 58.33	1/0; 75.00
$I_6 \to I_7$	2/8; 22.73	1/4; 25.00	1/2; 37.50
$I_7 \to I_8$	12/4; 73.53	7/3; 68.18	0/3; 12.50
$I_8 \to I_9$	5/7; 42.31	5/4; 55.00	2/0; 83.33
$I_9 \to I_{10}$	–	–	0/1; 25.00
Pooled I_0–I_9	59/56; 51.29	38/30; 55.80	8/9; 47.22
Pooled I_0–I_{10}	–	–	8/10; 44.74

Table 10: Accepted component changes over ten updates in the first AHE experiment.

Component	Full-set run	Split run
System prompt	10/10	10/10
Tool description	6/10	8/10
Tool implementation	0/10	0/10
Middleware	2/10	10/10
Skill	0/10	0/10
Sub-agent configuration	0/10	0/10
Long-term memory	0/10	0/10

C.3 Component Activation During Execution

An accepted middleware update remains inactive across all 136 evolution-set rollouts at I_1. This first split-run update adds iteration-aware reminders with thresholds of 15 and 25, together with prompt and tool-description changes. Synthetic checks activate the hook under its trigger conditions. Activation tracing thus distinguishes implemented changes from components actually exercised by benchmark tasks.

Later updates lower the thresholds, add loop detectors, and introduce command-history and apparent-success checks. Between I_7 and I_8, the evolution set records seven strict gains and three strict regressions. The held-out analysis records three regressions and zero gains, involving `adaptive-rejection-sampler`, `query-optimize`, and `rstan-to-pystan`. The same iteration combines evolution-set gains with held-out regressions.

C.4 Rollbacks and Predicted Repairs

In the full-set run, the first update adds workflow rules and middleware, followed by five strict gains and ten regressions. The second update rolls back both components, followed by eight gains and six regressions. At the eighth update, simplifying accumulated prompt and tool-description rules accompanies 12 gains and four regressions. These transitions characterize each joint update's realized instance-level outcomes.

Manifest predictions and realized repairs provide complementary diagnostics. The predicted-repair hit rate measures how many listed repairs appear in the next evolution-set evaluation. Full-set records contain 27 observed repairs among 81 task–action predictions (33.33%); split-run records contain 25 among 132 (18.94%). A task listed under multiple actions contributes one prediction per action. At update 8, zero of two predicted repairs appear, alongside the 12-gain/four-regression transition. At update 9, four of five predicted repairs appear, alongside five gains and seven regressions.

D Additional Memory Results and Instance-Level Cases

D.1 Complete Evaluation-Point Results

The complete Gemini trajectory contains 603 official successes across 900 instance–evaluation-point observations, covering 25 evaluation points through I_{144} (Table 11).

Table 11: Held-out performance and resource use at all 25 Gemini evaluation points.

Evaluation point	Pass/36	Pass@1 (%)	Mean steps	Tokens/instance
I_0	24	66.67	7.03	64,190.5
I_{11}	24	66.67	6.83	64,841.1
I_{12}	25	69.44	7.39	69,157.8
I_{23}	24	66.67	7.78	69,200.8
I_{24}	23	63.89	7.69	68,795.0
I_{35}	23	63.89	7.36	62,136.2
I_{36}	23	63.89	8.06	75,232.4
I_{47}	25	69.44	6.14	61,126.9
I_{48}	23	63.89	7.69	76,196.0
I_{59}	26	72.22	7.25	73,582.3
I_{60}	26	72.22	5.89	53,791.8
I_{71}	23	63.89	7.06	66,114.2
I_{72}	23	63.89	6.89	62,814.5
I_{83}	21	58.33	7.33	70,097.6
I_{84}	24	66.67	5.75	54,962.8
I_{95}	23	63.89	6.31	57,113.5
I_{96}	25	69.44	6.50	58,470.0
I_{107}	26	72.22	6.25	58,959.3
I_{108}	24	66.67	7.06	65,082.3
I_{119}	26	72.22	5.14	45,183.4
I_{120}	25	69.44	6.28	56,642.7
I_{131}	23	63.89	6.33	65,772.5
I_{132}	23	63.89	8.00	72,936.2
I_{143}	25	69.44	6.25	55,950.1
I_{144}	26	72.22	5.67	49,371.9

The 12 within-block final updates contain 12 gains and 11 regressions, giving 52.08% pooled EE. The 12 full-block comparisons contain 16 gains and 14 regressions, giving 53.23%. Across all 24 adjacent evaluation intervals, 28 gains and 26 regressions give 51.82% pooled EE. Table 12 lists the within-block and full-block comparisons.

Higher endpoint success coexists with repeated intermediate regressions. At I_{144}, 26 of 36 instances succeed (72.22%), compared with 24 of 36 (66.67%) at I_0. The endpoint comparison contains three gains (240, 301, and 585), one regression (238), 23 successes at both endpoints, and nine failures at both endpoints. Its smoothed EE is 70.00% (exact paired McNemar test: $p = 0.625$). The complete block I_{132}–I_{144} contains three gains and zero regressions; its final update I_{143}–I_{144} contains one gain and zero regressions.

D.2 Training-Time Selection and Memory Construction

The reward-blind selector captures 66 of 77 available successful opportunities (85.71%). Across 144 tasks and 720 candidates, selected trajectories achieve 45.83% success, compared with 44.72% expected from uniform selection. Table 13 reports candidate success, selection outcomes, and oracle pass@5. The numbers of tasks with zero through five successful candidates are 67, 7, 5, 7, 6, and 52, respectively.

The final bank contains 144 task entries and 346 structured items. The numbers of entries containing one, two, three, and four items are 26, 57, 38, and 23. The embedding cache stores one vector per

Table 12: Memory-evolution gains/regressions and smoothed EE (percent) for final-update and full-block comparisons.

Block	Within-block final update	Full-block comparison
1	2/1; 62.50	2/1; 62.50
2	1/2; 37.50	0/2; 16.67
3	0/0; 50.00*	0/0; 50.00*
4	0/2; 16.67	0/0; 50.00*
5	1/1; 50.00	3/0; 87.50
6	0/0; 50.00*	0/3; 12.50
7	3/0; 87.50	1/0; 75.00
8	2/0; 83.33	1/0; 75.00
9	0/2; 16.67	1/2; 37.50
10	1/2; 37.50	3/2; 58.33
11	1/1; 50.00	2/4; 35.71
12	1/0; 75.00	3/0; 87.50
Pooled	12/11; 52.08	16/14; 53.23

Table 13: Gemini MaTTS candidate, selection, and oracle outcomes.

Measure	Count	Rate (%)
Successful candidates	322/720	44.72
Successful selected trajectories	66/144	45.83
Oracle-solvable training tasks	77/144	53.47
Captured oracle opportunities	66/77	85.71
All five candidates fail	67/144	46.53
All five candidates succeed	52/144	36.11

task entry. At I_{144}, the 36 held-out instances retrieve 17 distinct training entries. Entries 233, 338, and 431 each serve five instances.

D.3 Training and Evaluation Resources

Candidate generation dominates the recorded training workload, accounting for 98.43% of 63,971,140 tokens. Selection and memory construction account for 0.76% and 0.81%. The 145 construction calls produce 144 committed entries. Stage time totals 50.77 hours across calls and parallel workers, including 49.52 actor-worker hours; summed task wall time is 12.85 hours. Token totals sum returned usage values, with unreported usage logged separately.

Table 14: Training resource breakdown.

Stage	Calls	Tokens	Share (%)	Stage hours
Five-way actor	8,037	62,965,241	98.43	49.52
Trajectory selector	144	485,478	0.76	0.44
Memory curation	145	520,421	0.81	0.81
Total	8,326	63,971,140	100.00	50.77

Held-out evaluation uses 2,310,859 recorded tokens at I_0 and 1,777,389 at I_{144}. Interaction steps total 253 and 204, respectively. Summed evaluation task wall times are 1.47 and 1.55 hours.

D.4 Template-Level Outcomes

An unchanged template score hides offsetting instance-level changes. Category-constrained selection (template 138) retains one success out of five, but that success shifts from instance 238 to 240.

Table 15: Gemini endpoint outcomes grouped by held-out intent template.

Template	n	I_0	I_{144}	Task IDs
138	5	1	1	238, 239, 240, 241, 242
180	5	3	4	298, 299, 300, 301, 302
189	5	5	5	511, 512, 513, 514, 515
193	1	1	1	96
194	5	2	3	585, 586, 587, 588, 589
197	5	3	3	47, 48, 49, 50, 51
208	5	5	5	324, 325, 326, 327, 328
214	5	4	4	188, 189, 190, 191, 192

Order-state lookup (180) and product reviews (194) each gain one successful instance. Each held-out template contains five instances, except template 193, which contains one.

D.5 Stable Subsets and Outcome Changes

All 54 adjacent outcome changes concentrate on 13 instances. The other 23 instances remain stable: 19 always succeed and four (50, 51, 241, and 242) always fail. Instances 301 and 585 each change outcome nine times, jointly contributing one-third of these events. Their histories include repeated loss and recovery before the final success. All five wishlist tasks (511–515) succeed at every evaluation point.

Table 16: The seven Gemini instances with the most outcome changes across 25 evaluation points.

Instance	Successes/25	Outcome changes	Task family
301	8	9	Order-state lookup
585	7	9	Product reviews
191	4	6	Order-state amount
587	4	6	Product reviews
238	3	5	Category-constrained selection
588	23	4	Product reviews
240	3	3	Category-constrained selection

D.6 Retrieval and Execution in Category Selection

Instance 240 asks for the most expensive item in a specified category. Between evaluation points I_{36} and I_{47}, retrieval changes from memory 143, covering page aggregation, to memory 262, covering hierarchical category navigation. At I_{47}, the agent follows the nested category path and succeeds. At I_{48}, memory 262 is again injected, while the execution follows an unsuccessful keyword-search route. Retrieval and effective memory use are distinct stages: the same injected entry accompanies different execution paths and outcomes.

D.7 Instance Outcomes Under Identical Retrieved Memory

Five held-out regressions occur with identical retrieved IDs and injected text. Between I_{60} and I_{83}, success falls from 26/36 to 21/36 as instances 301, 585, 587, 327, and 588 regress. Their trajectories include continued searching after a negative order lookup, a review submitted for the wrong product, an unconfirmed sorting direction, and repeated navigation up to the step limit. Instance-level analysis locates these execution differences while the retrieved content remains fixed.

E Additional Evolution Retention Results

ER exposes losses in previously demonstrated capabilities alongside new successes. Gemini's historical reference includes every earlier evaluation point. Instance records reconstruct the 36-by-

25 outcome matrix and ER in Table 17. At I_{96} and I_{108}, retention is 25/31 (80.65%) and 24/31 (77.42%), respectively. Instance 585 succeeds at the former point and fails at the latter.

At I_{120}, 24 of 31 historically successful instances succeed, giving ER of 77.42%. Instance 239 first succeeds at this point and enters the historical reference set at the next evaluation. At I_{144}, 26 of 32 historically successful instances succeed, giving ER of 81.25%. The six other instances are 191, 238, 239, 302, 587, and 589. The endpoint therefore combines higher aggregate success with the loss of six previously successful instances.

Table 17: Gemini instance-level transitions and historical success retention.

Evaluation point	Gains	Regressions	Historical	Retained	ER (%)
I_0	–	–	0	0	–
I_{11}	1	1	24	23	95.83
I_{12}	2	1	25	23	92.00
I_{23}	1	2	27	23	85.19
I_{24}	1	2	28	23	82.14
I_{35}	0	0	28	23	82.14
I_{36}	0	0	28	23	82.14
I_{47}	2	0	28	23	82.14
I_{48}	0	2	30	23	76.67
I_{59}	3	0	30	25	83.33
I_{60}	1	1	31	26	83.87
I_{71}	0	3	31	23	74.19
I_{72}	0	0	31	23	74.19
I_{83}	0	2	31	21	67.74
I_{84}	3	0	31	24	77.42
I_{95}	0	1	31	23	74.19
I_{96}	2	0	31	25	80.65
I_{107}	2	1	31	26	83.87
I_{108}	0	2	31	24	77.42
I_{119}	3	1	31	26	83.87
I_{120}	1	2	31	24	77.42
I_{131}	1	3	32	23	71.88
I_{132}	1	1	32	23	71.88
I_{143}	3	1	32	25	78.12
I_{144}	1	0	32	26	81.25

For AHE, the two-run mean any-success ER at I_9 is 80.84% on the evolution set, 71.80% on the held-out set, and 80.44% for the full-set setting. The first run's supplementary held-out evaluation at I_{10} retains seven of 12 historically successful tasks (58.33%). Figure 6 traces retention throughout memory and held-out harness evolution.

F Observed-Clade Accounting

Observed-clade success measures realized performance along the evolution sequence. Equation 7 pools successful and total evaluations over an agent state and its evaluated successors. Table 18 reports the complete first-run AHE histories and the Gemini history, including its baseline.

Three-point windows separate immediate success from continuation performance. AHE uses three consecutive evolution iterations and two runs, pooling six sets of 178 rollouts per window. ReasoningBank uses three successive 12-task block boundaries, each with 36 held-out observations. Table 19 gives all overlapping windows through I_9 for AHE and I_{144} for ReasoningBank.

AHE's mean immediate success is higher at I_5 than I_6 (62.36% versus 60.96%), while its three-point clade score is lower (62.08% versus 63.30%). Immediate scores and observed continuation quality therefore rank these two agent states differently.

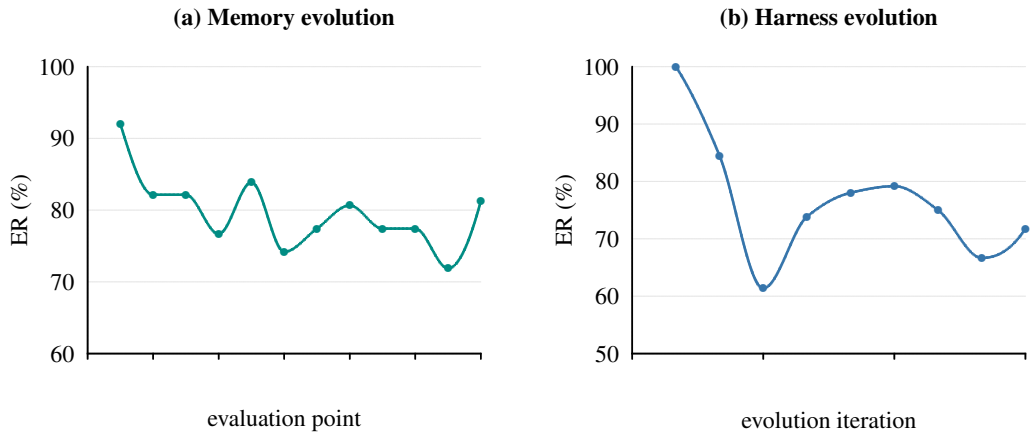

Figure 6: Evolution Retention. (a) Gemini pass@1 retention at block boundaries. (b) AHE held-out any-success retention, averaged over two runs.

Table 18: Whole-history success. AHE rows use the first run and rollout-level outcomes; ReasoningBank uses official pass@1.

Evaluation history	Points	Successes / evaluations	Score (%)
AHE full-set	10	1089/1780	61.18
AHE evolution	10	751/1360	55.22
AHE held-out	11	160/374	42.78
ReasoningBank held-out	25	603/900	67.00

Table 19: Complete three-point continuation windows: immediate success and clade success (percent).

Condition and range	Parent	Next 1	Next 2	Count	Clade
AHE, I_0–I_2	57.58	54.49	57.87	605/1068	56.65
AHE, I_1–I_3	54.49	57.87	58.99	610/1068	57.12
AHE, I_2–I_4	57.87	58.99	58.99	626/1068	58.61
AHE, I_3–I_5	58.99	58.99	62.36	642/1068	60.11
AHE, I_4–I_6	58.99	62.36	60.96	649/1068	60.77
AHE, I_5–I_7	62.36	60.96	62.92	663/1068	62.08
AHE, I_6–I_8	60.96	62.92	66.01	676/1068	63.30
AHE, I_7–I_9	62.92	66.01	62.92	683/1068	63.95
Gemini, I_0–I_{24}	66.67	69.44	63.89	72/108	66.67
Gemini, I_{12}–I_{36}	69.44	63.89	63.89	71/108	65.74
Gemini, I_{24}–I_{48}	63.89	63.89	63.89	69/108	63.89
Gemini, I_{36}–I_{60}	63.89	63.89	72.22	72/108	66.67
Gemini, I_{48}–I_{72}	63.89	72.22	63.89	72/108	66.67
Gemini, I_{60}–I_{84}	72.22	63.89	66.67	73/108	67.59
Gemini, I_{72}–I_{96}	63.89	66.67	69.44	72/108	66.67
Gemini, I_{84}–I_{108}	66.67	69.44	66.67	73/108	67.59
Gemini, I_{96}–I_{120}	69.44	66.67	69.44	74/108	68.52
Gemini, I_{108}–I_{132}	66.67	69.44	63.89	72/108	66.67
Gemini, I_{120}–I_{144}	69.44	63.89	72.22	74/108	68.52

G TASK PARTITION INDEX

Tables 20 and 21 list the disjoint 68-task evolution and 17-task held-out partitions, covering 85 tasks. The full-set AHE setting uses a separate 89-task collection. Shopping's 36 held-out IDs are 47–51, 96, 188–192, 238–242, 298–302, 324–328, 511–515, and 585–589.

Table 20: The 68 tasks in the fixed AHE evolution set.

Task identifier	Task identifier
bn-fit-modify	break-filter-js-from-html
build-cython-ext	build-pmars
chess-best-move	circuit-fibsqrt
code-from-image	configure-git-webserver
constraints-scheduling	count-dataset-tokens
crack-7z-hash	custom-memory-heap-crash
db-wal-recovery	distribution-search
dna-assembly	dna-insert
extract-elf	extract-moves-from-video
feal-differential-cryptanalysis	feal-linear-cryptanalysis
filter-js-from-html	financial-document-processor
fix-git	fix-ocaml-gc
git-leak-recovery	hf-model-inference
install-windows-3.11	kv-store-grpc
large-scale-text-editing	largest-eigenval
llm-inference-batching-scheduler	log-summary-date-ranges
mailman	make-doom-for-mips
mcmc-sampling-stan	merge-diff-arc-agi-task
model-extraction-relu-logits	modernize-scientific-stack
mteb-leaderboard	mteb-retrieve
nginx-request-logging	overfull-hbox
path-tracing	path-tracing-reverse
polyglot-c-py	polyglot-rust-c
portfolio-optimization	protein-assembly
prove-plus-comm	pypi-server
pytorch-model-cli	pytorch-model-recovery
qemu-alpine-ssh	qemu-startup
regex-chess	regex-log
reshard-c4-data	sam-cell-seg
sanitize-git-repo	sparql-university
sqlite-db-truncate	sqlite-with-gcov
torch-tensor-parallelism	tune-mjcf
video-processing	vulnerable-secret
winning-avg-corewars	write-compressor

Table 21: The 17 tasks in the fixed AHE held-out set.

Task identifier	Task identifier
adaptive-rejection-sampler	build-pov-ray
caffe-cifar-10	cancel-async-tasks
cobol-modernization	fix-code-vulnerability
gcode-to-text	gpt2-codegolf
headless-terminal	make-mips-interpreter
openssl-selfsigned-cert	password-recovery
query-optimize	raman-fitting
rstan-to-pystan	schemelike-metacircular-eval
torch-pipeline-parallelism	