

Rethinking the Evaluation of Harness Evolution for Agents

Yike Wang^{*1,2} Huaisheng Zhu^{*3} Zhengyu Hu² Yige Yuan² Zhengyu Chen³
 Shakti Senthil² Hannaneh Hajishirzi² Yulia Tsvetkov² Pradeep Dasigi¹ Teng Xiao^{*1}
¹Allen Institute for AI ²University of Washington ³Independent

Abstract

We revisit the evaluation of automatic harness evolution for LLM agents. Existing harness evolution methods use unit test cases to search for harness configurations and then report final performance on the same public benchmark. This protocol raises two fundamental concerns. First, harness evolution is itself an iterative search procedure that repeatedly evaluates and revises candidate harnesses using task feedback. As in agentic test-time scaling, it should therefore be compared with simple task-level search baselines under matched feedback and inference budgets to determine whether its gains arise from improved harness design or from additional search alone. Second, because the search and the final evaluation share the same benchmark, the reported gains risk overfitting to that specific task set. To address these concerns, we conduct an extensive evaluation comparing harness evolution with simple test-time scaling and discovery baselines under comparable feedback and inference budgets, and also evaluate evolved harnesses on held-out tasks to assess whether the discovered improvements generalize. Experiments on Terminal-Bench 2.1 with GPT-5.4 and Claude Opus 4.6 show that automatic harness evolution does not consistently outperform simple test-time scaling methods and exhibits limited generalization. Our results raise important questions about the effectiveness of automatic harness evolution and highlight the need for fairer evaluation protocols and benchmarks for automatic harness design. Our code is available at <https://github.com/rethinking-harness-evolution>.

1 Introduction

Large language model (LLM) agents increasingly rely on external harnesses to interact with complex environments [Yang et al., 2024, Merrill et al., 2026]. A harness defines the prompts, tools, memory, verification routines, and control logic through which a model observes tasks and acts [Lopopolo, 2026]. Prior work shows that harness engineering can substantially affect agent performance even with a fixed model [Trivedy, 2026], yet it remains largely manual: developers inspect trajectories, diagnose failures, and revise the harness. This has motivated automatic harness evolution, in which agents improve external harnesses [Lee et al., 2026, Lin et al., 2026, Zhang et al., 2026]. These methods typically follow a search loop: analyze prior trajectories, scores, and failures; propose modifications; evaluate them on benchmark tasks; and repeat [Lee et al., 2026, Zhang et al., 2026].

However, the evaluation of harness evolution needs further consideration. In many studies [Lee et al., 2026, Lin et al., 2026, Zhang et al., 2026], harness search uses verifier feedback from benchmark tasks, and the final harness is then evaluated on the same public benchmark, such as Terminal-Bench [Merrill et al., 2026]. This setup makes it important to compare harness evolution against test-time scaling baselines: methods that spend additional computation on the evaluation tasks themselves, for example through parallel sampling, sequential refinement, or task-level revision

*equal contribution

using verifier feedback [Snell et al., 2024, Li et al., 2026, Novikov et al., 2025]. These baselines test whether harness evolution provides benefits beyond simply allocating more search budget to the evaluation tasks under comparable feedback and inference budgets. The existing evaluation protocol therefore leaves a fundamental question unresolved: *Does harness evolution yield generalizable improvements in harness design, or are its gains primarily due to repeated sampling?* Moreover, when the tasks used for search overlap with those used for evaluation [Lee et al., 2026], observed gains may reflect adaptation to task-specific patterns rather than improvements in harness design that are transferable to held-out tasks.

In this paper, we revisit the evaluation of automatic harness evolution for language model agents. We compare parallel sampling, sequential refinement, and harness evolution under a controlled budget protocol that specifies what feedback each method receives, how it allocates inference compute, and whether it modifies task trajectories or the harness itself. We evaluate these methods on Terminal-Bench 2.1 with multiple models and find that automatic harness evolution does not consistently outperform simple test-time scaling baselines. For instance, as summarized in Figure 1, when unit test cases are unavailable, harness evolution underperforms simple test-time scaling baselines on average across Claude Opus 4.6, GPT-5.4, and GPT-5.4 mini. As shown in our experiments 4.3, when the same task set is used for both harness evolution and final evaluation, reported gains can overstate true harness improvements, as they may reflect adaptation to the evaluation instances rather than generalizable harness design. This highlights the need for evaluation protocols that separate optimization feedback from final measurement and compare against test-time scaling baselines.

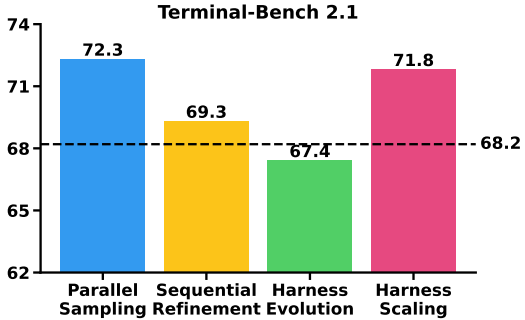


Figure 1: Average pass@1 without access to unit test feedback, averaged across Claude Opus 4.6, GPT-5.4, and GPT-5.4 mini. The dashed line indicates the performance of the initial harness. Harness evolution algorithms fail to outperform simple test-time scaling baselines.

2 Related Work

Test-time Scaling Test-time scaling refers to improving a language model’s outputs by allocating additional computation at inference, without modifying the model weights [Snell et al., 2024]. Existing work falls into two broad categories: parallel sampling [Wang et al., 2022, Brown et al., 2024] and sequential refinement [Madaan et al., 2023]. Parallel sampling draws multiple candidate answers independently and aggregates them by voting or verifier reranking. Sequential refinement generates answers iteratively, conditioning each attempt on prior ones to revise earlier reasoning. In this work, we argue that test-time scaling methods serve as strong and meaningful baselines for harness evolution algorithms.

Automatic Harness Evolution A growing body of work improves agent performance without retraining the underlying model by optimizing the external scaffold around the model. Early efforts target a single editable surface: prompt and instruction optimizers tune prompts, demonstrations, or language-model programs from execution feedback [Khatab et al., 2023, Opsahl-Ong et al., 2024, Yuksekgonul et al., 2025, Agrawal et al., 2025], while in-context methods accumulate experience as natural language artifacts that the model can reuse at inference time [Zhao et al., 2024, Zhang et al., 2025]. These approaches show that agent experience can be externalized outside model weights, but they typically optimize one component at a time. A more recent line treats the full harness as the object of search. Meta-Harness [Lee et al., 2026] searches over harness code using prior source code, scores, and execution traces. Agentic Harness Engineering [Lin et al., 2026] evolves prompts, tools, middleware, skills, and long-term memory through an observability-driven loop. AEVO [Zhang et al., 2026] views agentic evolution itself as an interactive environment and edits the procedure or agent context that controls future search. Our work revisits the evaluation of automatic harness evolution. We compare harness evolution against simple test-time scaling baselines under matched feedback and inference budgets, and evaluate whether evolved harnesses transfer to held-out tasks from the same benchmark.

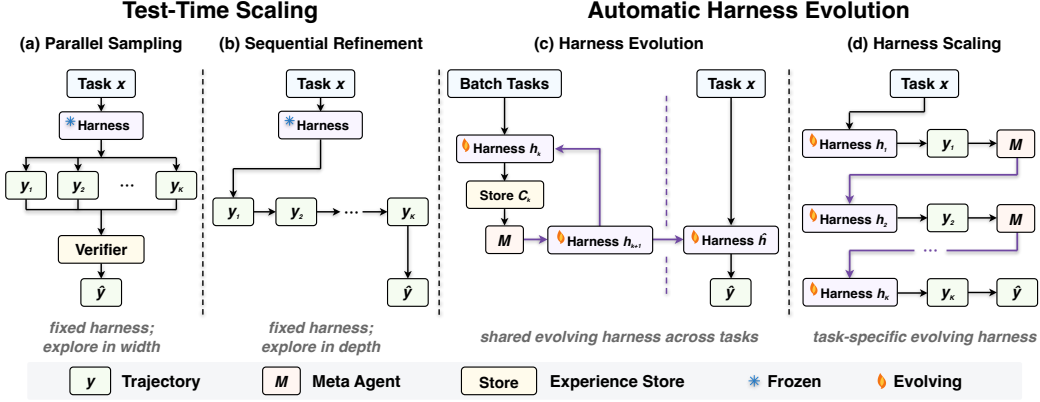


Figure 2: **Test-time scaling and automatic harness evolution algorithms.** We compare four approaches under a unified budget, emphasizing what feedback the agent receives, how it spends its budget, and whether it modifies the trajectories or the harness.

3 Rethinking the Evaluation of Harness Evolution

In this section, we formalize the evaluation of harness evolution under a unified budget view. The goal is to make explicit whether performance gains reflect reusable harness improvement or test-time discovery over the evaluated tasks. Given a fixed agent policy, a task distribution, and a compute budget, each method is specified by what it updates and what feedback it observes. We consider four methods: parallel sampling, sequential refinement, harness evolution, and harness scaling. They spend the budget on independent trajectories, trajectory revisions, shared harness updates across tasks, and task-specific harness updates, respectively. We first introduce the common notation, and then describe each method in Sections 3.2–3.5.

3.1 Preliminary

Let π_θ denote the agent policy and let x denote a task drawn from a task distribution $\mathcal{X}$. Executing the agent on x using the harness h yields a trajectory y , the full sequence of states, actions, and observations, which we write as $y \sim \pi_\theta(\cdot \mid x; h)$. When a unit test case g is available, we define an outcome $R(y, g) \in \{0, 1\}$ that returns 1 exactly when the trajectory solves the task. Every method below consumes a compute budget K and returns a final trajectory $\hat{y}$ for each task $x \sim \mathcal{X}$.

We also define a summarization map Φ , realized by the same underlying model as the policy. The map Φ summarizes an experience store into a form that a downstream agent can consume, such as recurring failures, redundant attempts, or cost, so that subsequent refinement or editing steps can diagnose why earlier attempts underperformed and avoid repeating edits.

3.2 Parallel Sampling

Parallel sampling (Figure 2 (a)) is a representative test-time scaling method that generates multiple trajectories in parallel [Wang et al., 2022, Brown et al., 2024]. Specifically, we draw K trajectories independently from the policy using a fixed harness h ,

$$y_1, \dots, y_K \sim \pi_\theta(\cdot \mid x; h),$$

such that exploration is spread across K mutually independent attempts. Increasing K expands the reachable action space, thereby improving the likelihood that the agent explores at least one trajectory that contains a correct solution. We consider two settings. First, when unit test cases are not available, selection instead relies on model self-selection, where the model is asked to choose the best candidate among the sampled solutions. Let J be a self judge realized by the agent itself, and we return the highest scoring candidate,

$$\hat{y} = \arg \max_{k \leq K} J(y_k).$$

Second, we assume access to unit test cases, which provide verifiable signals that can be used directly to select the final response. In this case, we set $\hat{y}$ to any trajectory accepted by the outcome,

$$\hat{y} \in \{y_k : R(y_k, g) = 1\}.$$

3.3 Sequential Refinement

In contrast to parallel sampling, sequential refinement [Madaan et al., 2023] (Figure 2 (b)) allocates the budget to depth, extending the interaction horizon so that each trajectory builds on the last. In this process, the model leverages prior reasoning experience and explores alternative solution paths to guide refinement in subsequent turns. To realize this process, we sample the first trajectory directly based on the prompt,

$$y_1 \sim \pi_\theta(\cdot \mid x; h),$$

and for $k = 2, \dots, K$, we condition the next trajectory on the previous one, which lets the agent reflect on and revise its prior attempt, using the same fixed harness h . When no unit test case is available, the agent reads the previous trajectory alone,

$$y_k \sim \pi_\theta(\cdot \mid x, \Phi(y_{k-1}); h),$$

and we return the final trajectory in the sequence, $\hat{y} = y_K$.

When a unit test case is available, we expose both the trajectory and its outcome, so that the agent can refine against the signal for the next turn’s response,

$$y_k \sim \pi_\theta(\cdot \mid x, \Phi(y_{k-1}, R(y_{k-1}, g)); h).$$

In this case, we return any trajectory accepted by the outcome,

$$\hat{y} \in \{y_k : R(y_k, g) = 1\}.$$

3.4 Harness Evolution

Harness evolution methods [Lee et al., 2026, Lin et al., 2026, Zhang et al., 2026] (Figure 2 (c)) retain the iterative loop of sequential refinement but allocate budget to the harness surrounding the policy rather than to trajectories alone, and optimize across a task distribution rather than fitting to one task. We draw a batch of tasks $\{x^{(i)}\}_{i=1}^n \sim \mathcal{X}$ and initialize the experience store with them, $\mathcal{C}_0 = \{x^{(i)}\}_{i=1}^n$. At round k we sample m rollouts per task under the current harness,

$$y_k^{(i,j)} \sim \pi_\theta(\cdot \mid x^{(i)}; h_k), \quad i = 1, \dots, n, \quad j = 1, \dots, m.$$

Each round then contributes one piece of evidence

$$e_k = (h_k, \{y_k^{(i,j)}\}_{i,j}), \quad \mathcal{C}_k = \mathcal{C}_{k-1} \cup \{e_k\},$$

recording the harness and the rollouts it produced across the batch. When unit test cases are available for each task, the evidence also carries the outcome of every rollout, so that

$$e_k = (h_k, \{y_k^{(i,j)}\}_{i,j}, \{R(y_k^{(i,j)}, g^{(i)})\}_{i,j}).$$

A meta agent $\mathcal{M}$ drives the search. It reads an observation $\Phi(\mathcal{C}_{k-1})$ of the accumulated store and emits the next harness,

$$h_k = \mathcal{M}(\Phi(\mathcal{C}_{k-1})), \quad k = 2, \dots, K,$$

starting from a seed harness h_1 . The search optimizes the harness toward expected success on the task distribution, or on a held-out distribution,

$$h^* = \arg \max_h \mathbb{E}_{x \sim \mathcal{X}} [R(y, g)], \quad y \sim \pi_\theta(\cdot \mid x; h).$$

When no unit test cases are available, we take the most evolved harness, $\hat{h} = h_K$; when unit test cases are available, we select the harness in the store with the highest aggregate outcome, estimated on the task batch or a held-out set,

$$\hat{h} = \arg \max_{k \leq K} \bar{R}(h_k), \quad \bar{R}(h_k) = \frac{1}{nm} \sum_{i=1}^n \sum_{j=1}^m R(y_k^{(i,j)}, g^{(i)}).$$

We then sample the final trajectory for the target task x under this harness, $\hat{y} \sim \pi_\theta(\cdot \mid x; \hat{h})$.

We instantiate harness evolution with AHE [Lin et al., 2026], with its *explore agent* disabled so that improvements come from evolving the harness on feedback rather than from retrieving benchmark-specific harnesses externally. See Appendix A for details.

3.5 Harness Scaling

To further investigate whether harness improvements arise from reusable design or from additional test-time computation, we introduce *harness scaling* (Figure 2 (d)), a harness-level analogue of test-time scaling. Whereas harness evolution aims to improve a harness across a set of tasks and produce a reusable design, harness scaling adapts the harness for a single evaluation instance by allocating additional inference budget to harness revision. In this sense, harness scaling can be viewed as instance-guided harness adaptation, in contrast to dataset-guided harness evolution. From an initial harness h_1 , we sample for each task

$$y_1 \sim \pi_\theta(\cdot \mid x; h_1).$$

After observing each trajectory, the agent updates the harness through a meta agent $\mathcal{M}$ instantiated by the same underlying model, and then resamples under the updated harness. When no unit test case is available, the update conditions on the task, the previous harness, and the previous trajectory,

$$h_k = \mathcal{M}(x, \Phi(h_{k-1}, y_{k-1})), \quad y_k \sim \pi_\theta(\cdot \mid x; h_k), \quad k = 2, \dots, K,$$

and the latest trajectory is returned $\hat{y} = y_K$.

When a unit test case is available, the update also conditions on the outcome of the previous trajectory,

$$h_k = \mathcal{M}(x, \Phi(h_{k-1}, y_{k-1}, R(y_{k-1}, g))), \quad y_k \sim \pi_\theta(\cdot \mid x; h_k), \quad k = 2, \dots, K,$$

so that the meta agent can revise the harness against the signal. We return any trajectory accepted by the outcome,

$$\hat{y} \in \{y_k : R(y_k, g) = 1\}.$$

4 Experiments

In this section, we study whether automatic harness evolution outperforms simple test-time discovery under comparable feedback and inference budgets. We examine three settings: without unit test cases (Section 4.2), with them (Section 4.3), and a generalization setting with disjoint search and evaluation tasks (Section 4.4). Across all three, we find that harness evolution does not consistently outperform simple test-time discovery, and that its advantage does not generalize when the search and evaluation tasks are disjoint.

4.1 Experimental Setup

Benchmark and Models Following prior work [Lee et al., 2026, Lin et al., 2026, Zhang et al., 2026], we evaluate on Terminal-Bench 2.1 [Merrill et al., 2026], a verified revision of Terminal-Bench 2.0. It repairs 28 of the 89 tasks, correcting external dependency drift, resource budget mismatches, and instructions misaligned with their tests, while keeping the suite at 89 terminal tasks, providing a more accurate assessment of code agents. We experiment with three frontier models: Claude Opus 4.6, GPT-5.4, and GPT-5.4 mini. Unless otherwise noted, every model is run with a maximum generation budget of 128k tokens and a high reasoning effort setting. To reduce variance from stochastic rollouts, all results are averaged over two independent runs.

Implementation To ensure a fair comparison, both Harness Scaling and Harness Evolution are initialized with the same initial harness used in AHE [Lin et al., 2026], denoted by h_1 . We also use this harness as the fixed harness h for Parallel Sampling and Sequential Refinement. For AHE, we sample one rollout per task for each harness, setting $m = 1$, to match the rollout budget across methods. We use a compute budget of $K = 5$ across methods.

4.2 Without Unit Test Cases, Automatic Harness Evolution Underperforms Test-Time Scaling

We first present results in the setting where unit test cases are unavailable. In this setting, the agent must rely solely on its own generated trajectories to revise either its solution trajectory or its harness. For parallel methods, the final answer is selected by a self judge over multiple sampled candidates. For sequential methods, the final answer is taken from the last refinement step. This setting therefore tests whether the agent can improve through self evaluation alone, rather than through explicit correctness signals.

Method	Claude Opus 4.6	GPT-5.4	GPT-5.4 mini	Average
direct sampling using initial harness	69.9	75.3	59.4	68.2
Test-Time Scaling				
Parallel Sampling	<u>74.7</u>	79.2	62.9	72.3
Sequential Refinement	73.0	73.0	<u>61.8</u>	69.3
Automatic Harness Evolution				
Harness Evolution	71.4	69.7	61.3	67.4
Harness Scaling	76.0	<u>78.1</u>	61.2	<u>71.8</u>

Table 1: Experimental results on Terminal-Bench 2.1 when unit test cases are unavailable. All results report pass@1. Best results in **bold** and second best in underline. Under this setting, automatic harness evolution algorithms underperform simple test-time scaling methods.

Table 1 shows the results. Overall, automatic harness evolution does not deliver a stronger performance gain than test-time scaling methods, and its effectiveness varies substantially across models. Parallel Sampling is the most consistent approach, improving the average score from 68.2 to 72.3 and yielding gains on all three models. Sequential Refinement provides only a marginal average improvement of 1.1 points and even slightly degrades GPT-5.4 relative to direct sampling. In contrast, Harness Evolution fails to outperform the direct sampling baseline on average. The degradation is most pronounced on GPT-5.4, where performance drops sharply from 75.3 to 69.7, indicating that iterative harness revision can actively hurt a strong model when the revision process is guided only by the agent’s own judgment. Harness Scaling partially closes this gap, reaching the best single result on Claude Opus 4.6 and a competitive average of 71.8.

Taken together, these results suggest that, in the absence of unit test cases, even very strong agents struggle to reliably extract useful learning signals from their own trajectories to revise the harness. Self generated feedback is noisy, and sequential revision based on such feedback risks compounding early mistakes. Simpler parallel sampling proves more effective in this feedback limited setting. This indicates that harness revision likely requires a reliable external correctness signal to ground the revision process.

4.3 With Unit Test Cases, Automatic Harness Evolution Underperforms Test-Time Scaling

Method	Claude Opus 4.6		GPT-5.4		Average	
	pass@1	pass@5	pass@1	pass@5	pass@1	pass@5
direct sampling using initial harness	69.9	-	75.9	-	72.9	-
Test-Time Scaling						
Parallel Sampling	84.8	84.8	87.1	87.1	86.0	86.0
Sequential Refinement	<u>83.1</u>	90.4	<u>85.4</u>	93.3	<u>84.3</u>	91.8
Automatic Harness Evolution						
Harness Evolution	73.0	83.2	78.6	<u>89.3</u>	75.8	86.2
Harness Scaling	<u>83.1</u>	<u>89.9</u>	82.0	88.8	82.6	<u>89.3</u>

Table 2: Experimental results on Terminal-Bench 2.1 when unit test cases are available. Best results in **bold** and second best in underline. Automatic harness evolution algorithms still underperform test-time scaling baselines in this setting, on both pass@1 and pass@5.

Second, we compare all methods under the setting where unit test cases are available. In this case, the unit test cases are used both as feedback for iterative refinement and as an oracle for selecting the final trajectory. Because oracle selection allows the best candidate among multiple sampled trajectories to be identified, reporting pass@5 becomes meaningful alongside pass@1, and we therefore report both metrics under this setting.

As shown in Table 2, we observe a similar trend under this setting. All test-time scaling and automatic harness evolution methods substantially improve over direct sampling with the initial harness, confirming that unit test feedback helps regardless of the mechanism. However, neither Harness Evolution nor Harness Scaling outperforms the simpler baselines on either metric. On

pass@1, Harness Evolution barely improves over direct sampling, while Parallel Sampling achieves the best average of 86.0. On pass@5, Sequential Refinement remains the strongest method with an average of 91.8, surpassing Harness Evolution (86.2) and Harness Scaling (89.3) on both models.

If harness revision genuinely produced better harnesses, we would expect the improvement to be reflected in pass@1. Instead, the benefit only materializes when we can select among multiple trajectories. This suggests that existing harness evolution algorithms do not reliably enable the agent to solve previously unsolved tasks through improved harness design. Rather, their gains largely stem from making multiple attempts, an effect that Parallel Sampling and Sequential Refinement achieve more directly and more effectively. This further indicates that refining solutions is a more productive use of extra compute than revising the harness itself.

4.4 Harness Evolution Fails to Generalize Beyond Training Tasks

Method	Claude Opus 4.6	GPT-5.4	Average
direct sampling using initial harness	63.3	72.1	67.7
Harness Evolution	64.5 (+1.2)	72.1 (+0.0)	68.3 (+0.6)

Table 3: Experimental results on Terminal-Bench 2.1 with disjoint search and evaluation tasks. We perform Harness Evolution on a training set with unit test cases, select the resulting harness based on validation performance, and report pass@1 on the test set. Compared with the pass@1 test results using the initial harness, Harness Evolution yields no significant gains, suggesting limited generalization and a strong tendency to overfit the search set.

Third, we evaluate the generalization ability of Harness Evolution, namely whether a harness evolved on one set of tasks can transfer to unseen tasks. Specifically, we split Terminal-Bench 2.1 into 45 training tasks, 10 validation tasks, and 34 held-out test tasks. We run Harness Evolution on the training set, select the best evolved harness based on validation performance, and evaluate it on the test set. The other three methods are not directly applicable in this setting, since they rely on instance-level scaling or refinement rather than producing a reusable artifact.

As shown in Table 3, the evolved harness yields only marginal gains over the initial harness on the test set: it improves Claude Opus 4.6 by a mere 1.2 points and brings no improvement at all on GPT-5.4, resulting in an average gain of just 0.6 points. This stands in sharp contrast to the improvements Harness Evolution achieves when evaluated on the same tasks it is optimized on, as reported in the previous experiments. The discrepancy suggests that the revisions discovered during evolution encode task-specific shortcuts rather than genuinely better harness design principles. In other words, current harness evolution algorithms exhibit limited generalization ability and appear prone to severe overfitting to the training tasks.

5 Discussion

5.1 Rational Harness Edits but Marginal Gains

To understand where the gains and limits of automatic harness evolution come from, we take a closer look at the trajectories produced by Harness Evolution and Harness Scaling. Specifically, we examine what the meta agent modifies at each iteration, which changes are kept or rolled back, and which failure classes the changes target. We find that while the meta agent makes rational, well motivated edits across the prompt, middleware, and tool layers, a stable core of hard tasks remains unaffected, and the resulting improvements stay limited beyond simple test-time discovery baselines.

Harness Evolution Under Harness Evolution, the meta agent’s first move is usually at the prompt layer, adding behavioral rules to the system prompt and long-term memory that target recurring failure classes, such as instructions to produce deliverables early with greater budget awareness, to copy fragile state before mutating it, and to recheck task constraints before finishing. When such advisory text plateaus, the meta agent escalates to runtime enforcement through middleware, such as turn budget trackers that remind the agent to ship output once turn or time thresholds are reached, truncation of oversized tool outputs, and finalization gates that block completion when deliverables

are missing or unverified. It also edits the tool layer, correcting misleading tool guidance and injecting recovery hints at command execution time.

Harness Scaling The dominant modification pattern in Harness Scaling is to encode knowledge about specific failures from previous rollouts into the next iteration’s prompt or long-term memory. For example, the meta agent records known bugs, implementation templates, verification checks, file paths, and command sequences for particular tasks. It also frequently improves workflow efficiency. Many changes replace repeated polling, exploratory commands, or slow setup sequences with compact command plans: batching installation and verification commands, skipping unnecessary environment detection, allowing longer shell timeouts for slow package installations, and prescribing workflows with explicit message budgets (Appendix B). These changes are especially helpful for tasks where the agent already knew the correct high level solution but lost time or failed due to timeouts, excessive polling, or incomplete setup. Another common class of edits strengthens verification behavior. The meta agent adds syntax checks, unit tests, or local checks that mimic the verifier before final submission.

Despite the breadth of these modifications, the overall improvement remains limited relative to test-time discovery baselines. The core issue is that most edits memorize fixes rather than distilling strategies. Much of this information is precisely what a competent agent can rediscover through exploration within a single rollout, so persisting it in the harness saves time on tasks the agent could already solve but rarely converts failures into successes. Meanwhile, the stable core of hard failures, which stem from deep domain reasoning demands or constraints outside harness control, remains unaffected by accumulated knowledge, and the growing volume of persistent prompt text introduces context bloat that can offset the remaining gains.

5.2 Task Difficulty and Harness Sensitivity as Factors

Following prior work, we evaluate all algorithms on Terminal-Bench and find that automatic harness evolution is often less effective than simple methods that scale computation at test time. While this may simply reflect that current models are not yet capable enough to revise the entire harness, in which case stronger priors could help, we offer two additional explanations for this result. First, agents already achieve relatively high scores on Terminal-Bench, and the small set of remaining failures may stem from limitations in the underlying model rather than deficiencies in the harness. Second, Terminal-Bench may simply not be very sensitive to harness design: a minimal setup consisting of a shell tool and a basic prompt already suffices for most solvable tasks, so performance is bottlenecked by the model’s reasoning rather than by the surrounding scaffolding. Modifying prompts, tool configurations, or middleware therefore yields only marginal gains.

These observations suggest that the value of automatic harness evolution is likely task dependent. We therefore encourage future work to study its effectiveness on benchmarks that satisfy two conditions: (1) the tasks are difficult enough that current agents leave substantial headroom for improvement, and (2) performance depends heavily on the harness, as when specialized tools, skills, or workflows are crucial to success. Under these conditions, a better harness can effectively expand the set of problems an agent can solve.

6 Conclusions

We study how automatic harness design for LLM agents is evaluated and observe that current methods commonly use feedback from unit test cases to search over harness configurations, then report final performance on the same public benchmark. This practice conflates genuine advances in harness design with simple baselines that perform discovery at test time, and introduces a risk of overfitting. Comparing automatic harness evolution against test-time discovery baselines under a unified budget, we find that it does not consistently outperform these baselines and that its gains generalize poorly beyond the training distribution. These findings suggest that current evidence for automatic harness evolution should be interpreted with caution, highlighting the need for fairer evaluation protocols for automatic harness design.

References

- Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.
- Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.
- Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.
- 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>.
- Xiaochuan Li, Ryan Ming, Pranav Setlur, Abhijay Paladugu, Andy Tang, Hao Kang, Shuai Shao, Rong Jin, and Chenyan Xiong. Benchmark test-time scaling of general llm agents. *arXiv preprint arXiv:2602.18998*, 2026.
- Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, et al. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026.
- Ryan Lopopolo. Harness engineering: Leveraging codex in an agent-first world, February 2026. URL <https://openai.com/zh-Hans-CN/index/harness-engineering/>.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.
- Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.
- Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.
- 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 *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 9340–9366, 2024.
- Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.
- Vivek Trivedy. Improving deep agents with harness engineering, February 2026. URL <https://www.langchain.com/blog/improving-deep-agents-with-harness-engineering>.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.
- John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.
- Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative ai by backpropagating language model feedback. *Nature*, 639(8055):609–616, 2025.

Jiayi Zhang, Yongfeng Gu, Jianhao Ruan, Maojia Song, Yiran Peng, Zhiguang Han, Jinyu Xiang, Zhitao Wang, Caiyin Yang, Yixi Ouyang, et al. Harnessing agentic evolution. *arXiv preprint arXiv:2605.13821*, 2026.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025.

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, pages 19632–19642, 2024.

A Experimental Details

A.1 Initial Harness

All four methods initialize the code agent from an identical minimal harness: the model is granted access to a single bash tool and nothing else, without skills, middleware, or persistent memory.

A.2 Summarization Map

To distill actionable evidence from raw rollout trajectories, we employ the Agent Debugger, a summarization agent introduced in the AHE framework [Lin et al., 2026]. Given multiple traces collected for each task, the Agent Debugger frames the trajectories as a navigable environment of files, where each trajectory message resides in its own file and is explored through generic shell and scripting tools. Traces that share the same query are grouped into a single environment, and the debugger is tasked with identifying the root cause of each failure or the pattern behind each success. Its findings are written to an analysis report for every task, annotated with pass or fail status, and then aggregated into a benchmark level overview that serves as an entry point for downstream analysis. The original traces remain available in both raw and lightly processed form, so claims in the reports can be verified on demand. All of this content is exposed as files that support progressive disclosure, which reduces token consumption while preserving access to full detail when needed.

A.3 Agentic Harness Engineering (AHE)

Explore Agent AHE ordinarily couples two components: an explore agent that retrieves harnesses tuned for the benchmark under evaluation from external sources, and an evolution loop that proposes and refines the harness from the feedback collected on the benchmark itself. Because the explore agent imports benchmark-specific harnesses rather than discovering improvements from feedback, it conflates reusable harness evolution with retrieval of solutions already fitted to the evaluation tasks. We therefore disable it and retain only the propose and refine loop, so that every reported gain is attributable to evolution over the feedback signal rather than to externally sourced, benchmark-specific harnesses.

A.4 Configuration

The full configuration used throughout is listed in Table 4; these values come from a snapshot of the reference run and remain fixed in every experiment. Each rollout executes in its own freshly provisioned E2B remote sandbox.

A.5 Evaluation Metrics

pass@1 Our primary metric is pass@1. Let $\mathcal{D}$ denote the task set with k rollouts per task, and let $r_{i,j} \in \{0, 1\}$ denote the binary reward of rollout j on task i . The pass@1 score is the average reward over all rollouts:

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

Rollouts that terminate on an infrastructure exception, such as a sandbox crash or an API timeout, are scored as $r = 0$ rather than excluded from the average.

pass@k To measure how much repeated sampling helps, we additionally report pass@ k , the fraction of tasks solved by at least one of the k rollouts, in the setting where unit test cases are available (Section 4.3):

$$\text{pass@k} = \frac{1}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} \max_{j \in \{1, \dots, k\}} r_{i,j}. \quad (2)$$

The same scoring convention applies: rollouts lost to infrastructure exceptions count as failures. Since a single success suffices for a task to count as solved, pass@5 upper bounds pass@1, and the gap between the two reflects the variance of the agent across independent attempts.

Agent	Hyperparameter	Value
Code Agent	Model	Claude Opus 4.6 / GPT-5.4 / GPT-5.4 mini
	Reasoning effort	high
	Max context	200,000 tokens
	Max generation per turn	128,000 tokens
	Max model turns	300
Agent Debugger	Model	Claude Opus 4.6 / GPT-5.4 / GPT-5.4 mini
	Reasoning effort	high
	Max context	200,000 tokens
	Max generation per turn	128,000 tokens
	Max model turns	25
	Context compaction threshold	0.75
	Retry attempts	3
	Retry backoff	2.0 s
Meta Agent	Model	Claude Opus 4.6 / GPT-5.4 / GPT-5.4 mini
	Reasoning effort	high
	Max context	200,000 tokens
	Max generation per turn	128,000 tokens
	Max model turns	500
	Context compaction threshold	0.75

Table 4: Experiment configurations.

B Case Study

Figure 3 presents representative examples of the modifications the meta agent applies to the harness. These modifications target recurring weaknesses observed in failed rollouts, including fragile setup and build procedures, wasteful polling loops and short shell timeouts, unsafe cleanup behavior, unpinned dependency versions, incorrect assumptions about data formats, destructive inspection of state that must be preserved, misconfigured embedding prompt types, and attempts by the agent to weaken tests. Notably, many fixes take the form of task-specific rules and facts embedded directly into the prompt, such as prescribed command orderings and verified dataset properties, echoing our earlier observation that edits tend to memorize fixes rather than distill general strategies. As discussed in Section 5.1, however, although these edits are individually reasonable, the stable core of hard failures remains unaffected.

caffe-cifar-10: timeout -> pass at iter 2	
1	- foreground apt-get + Makefile detour + slow CIFAR polling loop
2	- repeated sleep/check turns; training never started before timeout
1 +	force background installs and logs; use cmake-first legacy Caffe path
2 +	use Python/Keras CIFAR source first; avoid polling loops
3 +	result: exception -> PASS (iteration 2 onward)
cancel-async-tasks: fail -> pass at iter 2	
1	- cancel all unfinished tasks again after gather cancellation
2	- async finally blocks can receive a second CanceledError
3	- agent weakened tests after seeing unexpected cleanup behavior
1 +	add t.cancelling() guard before cancel()
2 +	embed reference run_tasks pattern and "do not weaken tests" rule
3 +	result: FAIL -> PASS (iteration 2 onward)
compile-compcert: fail -> pass at iter 3	
1	- installed wrong Coq version first; rebuilt after discovering bounds
2	- short shell timeout and sleep polling consumed build budget
3	- functional test ran before make install, causing link failures
1 +	install Coq 8.16.1 directly and increase shell timeout
2 +	smart wait loop replaces repeated sleep commands
3 +	run make install before ccomp functional tests
4 +	result: FAIL -> FAIL -> PASS (iteration 3 onward)
count-dataset-tokens: fail -> pass at iter 2	
1	- concatenated reasoning+solution before tokenization
2	- tokenizer.encode used default special-token behavior
3	- answer drifted to 79566 instead of verified separate-field count
1 +	make separate-field tokenization the first mandatory rule
2 +	use add_special_tokens=False and show correct/wrong code patterns
3 +	embed dataset config/domain/column facts directly in prompt
4 +	result: FAIL -> PASS (iteration 2, 4, 5)
db-wal-recovery: fail -> pass at iter 2	
1	- opened sqlite3 and ran PRAGMA wal_checkpoint before backup
2	- corrupted WAL was deleted; rest of run tried unrecoverable forensics
3	- later passing run risked not writing fixed WAL back to /app/main.db-wal
1 +	never open SQLite before backing up .db/.db-wal/.db-shm
2 +	prohibit wal_checkpoint; repair binary WAL before querying
3 +	require writing fixed WAL back and verifying all 11 records
4 +	result: FAIL -> PASS (iteration 2 onward)
mteb-retrieve: four fails -> pass at iter 5	
1	- encoded query and passages identically with same task_name
2	- LongTermMEMORY over-prescribed STS22/T2Retrieval without prompt_type
3	- agent repeated same ranking loop across four failed iterations
1 +	use MTEB PromptType.query for query embedding
2 +	encode passages with passage/default prompt type
3 +	add diagnostics: inspect enum values and print full ranking
4 +	result: FAIL x4 -> PASS (iteration 5)

Figure 3: Examples of harness modifications made in Harness Scaling.