TURBO HARNESS: INSTANCE-ADAPTIVE HARNESS OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

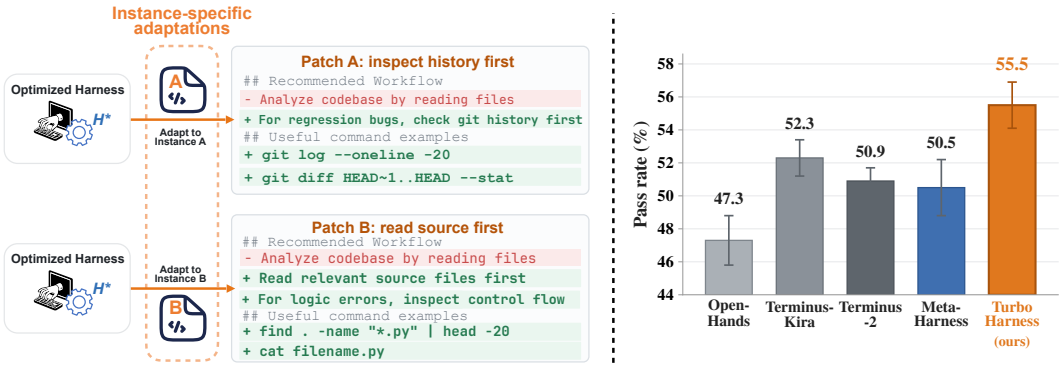
Automating the search for effective harnesses is an important step toward enabling agents to recursively self-improve. Existing harness optimizations typically produce a single global harness that is applied uniformly across task instances. However, a harness that works well on average may not be optimal for every instance. We introduce *Turbo Harness*, a framework that can adapt a globally optimized harness to each instance by reusing information generated during the original optimization process. Specifically, Turbo Harness recycles artifacts produced during a completed global harness optimization run, and summarizes them into a structured playbook. We train a harness editor to leverage this prior optimization experience to generate instance-specific patches to the global harness. At inference time, the editor uses the instance and the playbook to construct a tailored harness in which the execution model operates. Through numerical experiments, we show that Turbo Harness consistently outperforms existing harness optimization baselines across seven benchmarks spanning interactive agent tasks, software engineering, and long-horizon terminal tasks.

1 INTRODUCTION

Large language models (LLMs) and their surrounding harnesses are two fundamental building blocks of modern agents (Tian et al., 2026). The notion of a harness has expanded well beyond simply exposing an LLM to an environment or a collection of tools (Yao et al., 2022b). Today, it encompasses nearly all of the runtime machinery surrounding the model (e.g., context management, memory, workflow orchestration, and verification) (Huang et al., 2026; Zhong & Zhu, 2026; Zhou et al., 2026). These mechanisms can substantially extend the effective capabilities of a static LLM, particularly on long-horizon, scientific, and open-ended tasks (Ma et al., 2026; Wu et al., 2026). At the same time, this broad definition creates an enormous design space: for an application of interest, one must decide what information to expose to the model, what state to maintain, when and how to intervene, and how much computation to allocate. Designing an effective harness is therefore a challenging optimization problem in its own right.

A growing line of work seeks to automate this process through *harness optimization* (Anthropic, 2025; Lee et al., 2026b;a). For example, Meta-Harness (Lee et al., 2026b) adopts an outer-loop program-search approach, in which a separate agent proposes new harnesses based on the execution traces of previously evaluated candidates; Harness-R1 (Shao et al., 2026) trains a harness engineer with reinforcement learning (RL) to transform batches of agent failures into executable runtime patches. Despite their promise, existing approaches have two limitations. First, they optimize a single global harness that is then applied uniformly across all, or a batch of, test instances. Yet a harness that is effective for one instance may be suboptimal for another. For example, SWE-Bench-style instances may come from different GitHub repositories with distinct development workflows and testing conventions, and may benefit from different harness configurations. Second, harness optimization is expensive and much of the information generated during this process is ultimately underutilized. During optimization, the system accumulates a rich history of candidate harnesses and evidence about which strategies succeed or fail under different conditions. Existing methods use this experience to improve the global harness (Lee et al., 2026b;a; Sengupta & Wang, 2026). Once optimization is complete, however, much of the instance-specific signal contained in this history is discarded. This leads to a fundamental question:

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.



(a) Instance-specific harness adaptation

(b) Results on Terminal-Bench 2.1

Figure 1: **Turbo Harness adapts a globally optimized harness $H^\star$ to each instance.** *(a)* A harness editor produces an instance-specific patch to $H^\star$, yielding a per-instance harness. *(b)* On Terminal-Bench-2.1 (Claude Sonnet 4.5, 44-task test split), Turbo Harness achieves the highest pass rate compared with several widely-used human-designed harnesses.

Can we adapt a globally optimized harness to each individual instance by reusing the experience accumulated during harness optimization?

In this paper, we introduce ***Turbo Harness***[1]. It is a new harness optimization framework that reuses artifacts from a completed outer-loop harness search to adapt a globally optimized harness to individual task instances.

Specifically, we reuse the search archive produced during outer-loop optimization and summarize it into a structured playbook (Zhang et al., 2026c). This playbook records both successful and unsuccessful harness-editing strategies, including the specific edits that lead to performance improvements and those that fail to do so. We then train a *harness editor*, an RL-fine-tuned open-source model responsible for adapting the global harness to each instance. At deployment time, the harness editor receives a test instance together with access to the playbook and proposes a patch to the globally optimized harness. The resulting instance-specific harness is then used by the main execution model, which can be any model, to solve the test instance. Figure 1 illustrates this instance-specific adaptation and shows its gains on Terminal-Bench-2.1.

Our framework offers three advantages. First, instance-adaptive harnessing provides more flexibility than a single global harness across all task instances, consistently improving task performance in our experiments. For example, on SWE-smith-MR, Turbo Harness improves pass rate from 50.7% to 64.0% with Claude Haiku 4.5 and from 70.7% to 88.0% with Gemini 3.7 Flash. Second, training the harness editor is lightweight: rather than generating a new harness from scratch, it only needs to patch a globally optimized harness, guided by a playbook that summarizes successful and unsuccessful edits from the outer-loop search. This allows us to use a relatively small editor model (Qwen3.5-9B in our experiments). Finally, Turbo Harness adds little inference-time overhead because the editor is called only once per instance; moreover, the adapted harness can often reduce the number of execution steps required by the much larger execution model, potentially even lowering overall inference-time compute (see e.g., Table 2 and 3).

In short, our contributions are:

- We formulate *instance-adaptive harness optimization* and highlight the limitations of deploying a single globally optimized harness across all task instances.
- We propose Turbo Harness, which reuses the artifacts collected during the outer-loop harness search to train a harness editor that adapts the global harness to each individual task instance.
- We show that Turbo Harness is lightweight: the editor only patches a globally optimized harness, can be instantiated with a small model, and is invoked only once per task instance.

[1]We adopt the name "Turbo Harness" by analogy to a turbocharger, which recovers energy from exhaust gas and reuses it to improve engine performance.

- We demonstrate that instance-specific harness adaptation consistently outperforms globally optimized harnesses across seven benchmarks spanning interactive agent tasks, software engineering, and long-horizon terminal tasks, while often reducing execution steps and costs.

2 Related Work

Prompt and Context Optimization. A growing line of work improves frozen language models by optimizing their prompts or, more broadly, the information provided to them during inference (Ramnath et al., 2025; Li et al., 2025). For example, GEPA (Agrawal et al., 2026) evaluates execution trajectories to evolve prompt candidates via a Pareto-based search. ACE (Zhang et al., 2026c) accumulates and refines an evolving playbook of strategies through generation, reflection, and curation. MCE (Ye et al., 2026) further expands this design space by co-evolving context-engineering skills and query-conditioned context functions, whose artifacts may include files and code. They primarily optimize the model's context. Harness engineering, however, encompasses a broader design space: beyond deciding what information is presented to the model, it can modify the executable runtime itself, including control flow, tool-use workflows, permissions, and verification mechanisms.

Automatic Harness Optimization. Recent methods (Sengupta & Wang, 2026; Lin et al., 2026; Pan et al., 2026; Park et al., 2026) optimize the agent scaffold itself. Meta-Harness (Lee et al., 2026b) uses a strong external coding agent to search over executable harnesses by inspecting previous implementations, scores, and execution traces. Self-Harness (Zhang et al., 2026b) removes the stronger external proposer, allowing the target model to identify failure patterns, propose harness modifications, and validate them through testing. Both methods produce harnesses intended to generalize across a collection of instances. AutoDesign (Luo et al., 2026) likewise applies meta-harness optimization to long-horizon agentic design tasks. Recursive Harness Self-Improvement (Lee et al., 2026a) performs lightweight optimization by representing the agent loop as a prompt-level object and repeatedly revising it using pairwise feedback from its revision history. Unlike them, our method introduces an instance-adaptive harness without requiring a new multi-round search for each instance. This strategy enables more flexibility, which translates into consistent performance gains in our experiments.

Adaptive Prompt and Harness Optimization. A related line of work studies prompt and harness optimization that tailors the resulting prompt or harness to individual task instances. For example, Advisor Models (Asawa et al., 2025) trains an advisor model that enables instance-specific prompt optimization. TTHE (Nie et al., 2026) performs harness search on unlabeled test batches and carries the selected program to subsequent batches. Harness-R1 (Shao et al., 2026) trains a harness engineer from batches of failure trajectories and evaluates each candidate patch by rerunning the same tasks under the modified harness. A concurrent work JIT-Agent (Zhang et al., 2026a) also targets instance-specific harnesses, but generates each harness from scratch via teacher distillation and evolutionary RL. In contrast, Turbo Harness at deployment time introduces only a single additional call to a small model, avoiding the two-pass procedure required by Harness-R1 (Shao et al., 2026) to first collect a failure trajectory and then rerun the task under the resulting patch. Moreover, Turbo Harness edits an existing global harness rather than synthesizing a new harness from scratch for each task instance.

3 Preliminaries: Harness Optimization

Objective of Harness Optimization. An agent consists of a language model $\mathtt{M}$ and a harness $\mathtt{H}$. Given a task instance x, the agent executes $\mathtt{M}$ under $\mathtt{H}$ to interact with the task environment and produce an output. The harness is an executable program that governs this interaction, including context construction, tool use, state management, and execution control.

Given a frozen model $\mathtt{M}$, a task distribution $\mathcal{T}$, and an admissible harness space $\mathcal{H}$, harness optimization seeks a harness that maximizes the expected task performance:

$$\mathtt{H}^\star \in \arg\max_{\mathtt{H}\in\mathcal{H}} \mathbb{E}_{x\sim\mathcal{T}} \left[r(\mathsf{Agent}(\mathtt{M}, \mathtt{H}, x)) \right], \tag{1}$$

where r denotes the task-level evaluation function, with the expectation also accounting for stochasticity in agent execution.

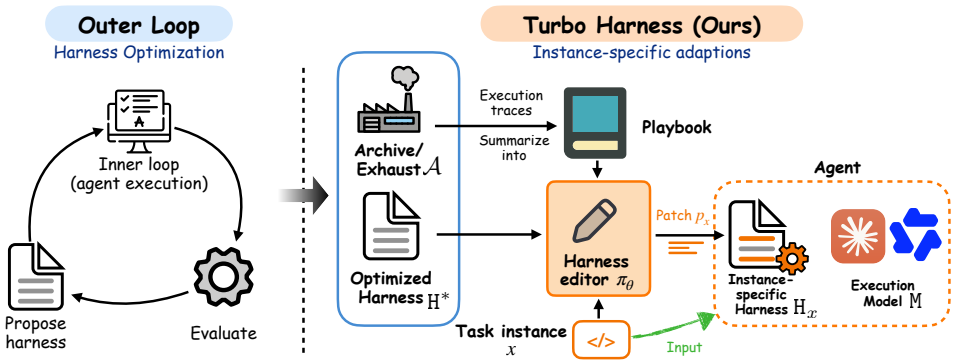


Figure 2: **Turbo Harness overview.** *Left:* an outer loop optimizes a global harness $\mathtt{H}^\star$, producing "exhaust" (traces, reflections, evaluations) as a byproduct. *Right:* we recycle this exhaust into a *playbook*; a small RL-trained *harness editor* π_θ conditions on an instance x and the playbook to patch $\mathtt{H}^\star$ into an instance-specific harness $\mathtt{H}_x$, which the frozen model $\mathtt{M}$ then runs.

Inner- and Outer-loop Optimization. A standard approach to optimizing Equation 1 is to organize search into an *inner loop* and an *outer loop* (Lee et al., 2026b;a; Zhang et al., 2026b). The inner loop evaluates a candidate harness by running the corresponding agent on a set of task instances and collecting feedback, including task scores, execution traces, tool outputs, and failures. The outer loop then uses this feedback to diagnose the current harness and propose an improved candidate. Repeating these stages yields an iterative search process in which agent executions provide the evidence used to update the harness.

The outer-loop optimizer may range from a hand-designed update rule (Agrawal et al., 2026; Yang et al., 2024a; Yuksekgonul et al., 2024) to an LLM-based agent that automatically modifies prompts, control logic, or the harness program itself (Ursekar et al., 2026; Lee et al., 2026b). In the latter case, Meta-Harness (Lee et al., 2026b), for example, uses a coding agent to search over harness programs by inspecting previous candidates, evaluation results, and execution traces. However, Meta-Harness produces a globally optimized harness that is applied across future task instances. This can be suboptimal when different instances benefit from different harness strategies. For example, a simple coding task can be solved by a lightweight edit-and-test loop, whereas a difficult task may require specialized tools or more extensive verification.

4 Our Method

We introduce Turbo Harness (Figure 2), which adapts a global harness to each task instance by reusing experience collected during the outer-loop harness search. Our goal is: for each task instance x, find an optimal harness $\mathtt{H}_x$ such that

$$\mathtt{H}_x^\star \in \arg\max_{\mathtt{H}_x \in \mathcal{H}} \left[r(\mathsf{Agent}(\mathtt{M}, \mathtt{H}_x, x)) \right]. \tag{2}$$

To achieve it, we train a lightweight LLM, which we call the harness editor, to take as input a task instance together with experience gathered during the outer-loop search and propose edits to the global harness. Below, Section 4.1 describes our instance-specific adaptation method and Section 4.2 shows how we train the harness editor.

4.1 Instance-Specific Harness Optimization

Harness Editor. Searching for an optimal harness independently for each task instance can be expensive, as it may require evaluating multiple candidate harnesses per instance. Instead, we train a harness editor π_θ to propose instance-specific code edits to the globally optimized harness $\mathtt{H}^\star$ obtained from the outer-loop optimization in Eq. 1. The harness editor is a small open-weight LLM with trainable parameters θ (e.g., we use Qwen3.5-9B in our experiments). Because it is lightweight and called only once per task instance, its deployment overhead is minimal.

To further reduce training cost, we reuse the experience collected during outer-loop harness optimization. In particular, the search archive records the outcomes of all evaluated harnesses, including

Algorithm 1 Turbo Harness

Offline training

Require: Outer-loop harness-search archive $\mathcal{A}$; training instances $\mathcal{D}_{\text{train}}$; global harness $\texttt{H}^\star$; frozen execution model $\texttt{M}$; initial editor π_θ; proposals per instance G

Output: Trained editor π_θ and playbook $\mathcal{P}$

$\mathcal{P} \leftarrow \textsc{Summarize}(\mathcal{A})$ ▷ Distill the outer-loop harness-search archive into a playbook

for each training step **do**

  $\mathcal{D}_b \leftarrow \textsc{SampleBatch}(\mathcal{D}_{\text{train}})$ ▷ Sample a mini-batch of instances

  $\{p_{x,g}\}_{g=1}^{G} \sim \pi_\theta(\cdot \mid x, \texttt{H}^\star, \mathcal{P}), \quad x \in \mathcal{D}_b$ ▷ G candidate patches per instance

  $\texttt{H}_{x,g} \leftarrow \texttt{H}^\star \oplus p_{x,g}$ ▷ Apply each patch to $\texttt{H}^\star$

  $R_{x,g} \leftarrow \textsc{Evaluate}(\textsf{Agent}(\texttt{M}, \texttt{H}_{x,g}, x))$ ▷ Reward from agent execution

  $\theta \leftarrow \text{GRPO}(\theta, \{(x, p_{x,g}, R_{x,g})\}_{x,g})$ ▷ Update the editor model (Eq. 4)

end for

Instance-adaptive inference

Require: Task instance x; global harness $\texttt{H}^\star$; frozen execution model $\texttt{M}$; trained editor π_θ; playbook $\mathcal{P}$

Output: Task result y

$p_x \sim \pi_\theta(\cdot \mid x, \texttt{H}^\star, \mathcal{P})$ ▷ One patch for the test instance

$\texttt{H}_x \leftarrow \texttt{H}^\star \oplus p_x$ ▷ Apply patch to the global harness

$y \leftarrow \textsf{Agent}(\texttt{M}, \texttt{H}_x, x)$

candidates discarded during search. Following ACE (Zhang et al., 2026c), we summarize these successes and failures into a *playbook* $\mathcal{P}$ of editing strategies and the conditions under which they are useful. This way, the harness editor does not need to discover new strategies from scratch. It only needs to learn how to use the playbook and identify which previously discovered strategies are relevant to a given task instance. This strategy significantly reduces the training cost and enables the use of a small LLM.

Specifically, given the instance x, the source code of $\texttt{H}^\star$, and the playbook $\mathcal{P}$, the harness editor π_θ generates a patch p_x that produces the adapted harness $\texttt{H}_x$ when applied to $\texttt{H}^\star$:

$$\pi_\theta(\cdot \mid x, \texttt{H}^\star, \mathcal{P}) \text{ outputs a patch } p_x \text{ and let } \texttt{H}_x = \texttt{H}^\star \oplus p_x, \tag{3}$$

where $\oplus$ denotes applying code edits to a harness.

4.2 Training a Harness Editor

We train the harness editor π_θ to favor edits to the global harness $\texttt{H}^\star$ that improve task performance when the frozen LLM executes the task. We optimize the editor with RL, using the resulting task performance as the reward. For each training instance, we evaluate multiple candidate edits and optimize the policy based on their relative rewards.

Training Rewards. For each instance x in the training set $\mathcal{D}_{\text{train}}$, we sample G candidate harness edits $\{p_{x,g}\}_{g=1}^{G}$ from π_θ. We apply each edit to $\texttt{H}^\star$ to obtain $\texttt{H}_{x,g}$, execute $\textsf{Agent}(\texttt{M}, \texttt{H}_{x,g}, x)$, and compute a reward $R_{x,g}$ from the resulting task performance and task-specific evaluation metrics. Edits that cannot be applied or fail validation receive zero reward and are not executed.

Optimization. We optimize π_θ with GRPO (Shao et al., 2024). Let $\mathcal{B}$ be the set of candidate indices in an update minibatch, with $i = (x, g)$ and $p_i = p_{x,g}$. For proposals sampled from the pre-update policy $\pi_{\theta_{\text{old}}}$, the standard GRPO objective is

$$\mathcal{J}_{\text{GRPO}}(\theta; \mathcal{B}) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{1}{|p_i|} \sum_{t=1}^{|p_i|} \min[\rho_{i,t}\widehat{a}_i,\ \text{clip}(\rho_{i,t}, 1-\epsilon, 1+\epsilon)\,\widehat{a}_i] - \beta \mathrm{D}_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}})\,. \tag{4}$$

Here, $|p_i|$ is the token length of proposal p_i, $p_{i,t}$ its t-th token, and $p_{i,<t}$ its preceding prefix. The advantage $\widehat{a}_i$ is computed from the rewards $\{R_{x,g}\}_{g=1}^{G}$ of candidates for the same instance x. The token probability ratio is $\rho_{i,t} = \frac{\pi_\theta(p_{i,t} \mid x, \texttt{H}^\star, \mathcal{P}, p_{i,<t})}{\pi_{\theta_{\text{old}}}(p_{i,t} \mid x, \texttt{H}^\star, \mathcal{P}, p_{i,<t})}$.

The reference policy π_{ref} is the frozen initial editor, $\mathrm{D}_{\mathrm{KL}}(\pi_\theta \| \pi_{\text{ref}})$ is the sampled KL penalty (Shao et al., 2024; Ouyang et al., 2022; Schulman et al., 2017), and ϵ, β control clipping and KL regularization; the execution model $\mathtt{M}$ remains frozen throughout training.

At inference, a single editor call conditions on $\mathtt{H}^\star$ and the playbook $\mathcal{P}$ to turn each test instance x into a patch p_x, adapting $\mathtt{H}^\star$ into an instance-specific harness $\mathtt{H}_x$ (falling back to $\mathtt{H}^\star$ if the patch fails to apply), which the frozen execution model $\mathtt{M}$ then runs; Algorithm 1 and Figure 2 give the end-to-end pipeline.

5 Experiments

We evaluate Turbo Harness on seven benchmarks spanning interactive agent tasks, software engineering, and long-horizon terminal tasks, comparing it with recent harness optimization and prompt-based baselines. We assess gains in task performance and execution efficiency, and conduct ablation studies to investigate where the gains come from.

5.1 Experimental Setup

Benchmarks. We evaluate Turbo Harness on seven benchmarks spanning agentic tasks, software engineering, and long-horizon terminal interaction. For the four agentic benchmarks, ALFWorld (Shridhar et al., 2020), ScienceWorld (Wang et al., 2022), DBBench (Liu et al., 2024), and WebShop (Yao et al., 2022a), we report success rate, mean environment score in $[0, 100]$, accuracy, and Success@1, respectively. For software engineering, we use *SWE-smith-MR*, our multi-repository subset of SWE-smith (Yang et al., 2026), and SWE-bench Verified (Jimenez et al., 2024). On both coding benchmarks, pass rate is the fraction of test issues resolved by explicitly submitted code patches, with unsubmitted attempts counted as failures. For Terminal-Bench-2.1 (TB2.1) (Merrill et al., 2026), we report task pass rate. Higher values indicate better performance for all seven task metrics.

Data splits and budgets. We evaluate on held-out test instances using the benchmark-specific splits in Appendix A.2. SWE-smith-MR contains 50 training and 50 test issues sampled from 25 Python repositories. SWE-bench Verified uses a repository-stratified split of 251 training, 99 validation, and 150 test issues. Both coding benchmarks hold out issues within the selected repositories and impose a 40-step execution limit and a \$3 cost limit per issue. Compared with the 250-step budget commonly used for SWE-bench Verified evaluation, this tighter budget is intended to stress-test harness quality by limiting the extent to which additional interaction can compensate for ineffective runtime guidance. TB2.1 uses a held-out test split of 44 tasks.

Models. We use Qwen3.5-9B (Qwen Team, 2026) for the four agentic benchmarks, evaluate each coding benchmark with Claude Haiku 4.5 (Anthropic, 2025a) and Gemini 3.7 Flash (Google DeepMind, 2026) separately, and use Claude Sonnet 4.5 (Anthropic, 2025b) for TB2.1. Qwen3.5-9B serves as the trainable harness editor in our experiment. Meta-Harness proposals and playbook curation use Claude Sonnet 4.5, except on TB2.1, which use Claude Opus 4.6 (Anthropic, 2026).

Baselines. Our primary comparison is *Meta-Harness* (Lee et al., 2026b), whose globally optimized harness and search archive provide the starting point for Turbo Harness. This comparison evaluates the added value of instance-specific adaptation. We also compare with *Default*, the initial scaffold without harness optimization: mini-swe-agent (Yang et al., 2024b) for the coding benchmarks and a basic tool-calling loop for each agentic benchmark. Terminus-Kira (KRAFTON AI & Ludo Robotics, 2026) and Terminus-2 (Merrill et al., 2026) serve as the default harnesses on TB2.1. Additional agentic baselines include ReAct (Yao et al., 2022b), Self-Refine (Madaan et al., 2023), Reflection (Shinn et al., 2023), and *Harness-R1* (Shao et al., 2026). We adapt Harness-R1 to ScienceWorld by porting its code-hook method into the Harness-R1 runtime. Appendix C details the Default and prompt-based implementations, including Reflection's second attempt; Appendix C.3 documents Harness-R1's runtime and rollout-budget differences.

Editor training. For each benchmark and execution model, we construct a playbook from the completed Meta-Harness search and hold it fixed during editor training and evaluation. We train the editor with GRPO (Shao et al., 2024) using the task-specific rewards in Appendix B.3. Appendix B.4 specifies checkpoint selection and the WebShop-specific training configuration.

Table 1: **Task performance on four agentic benchmarks** (frozen Qwen3.5-9B). Higher is better; metrics are success rate (%, ALFWorld), mean score (/100, ScienceWorld), accuracy (%, DBBench), and Success@1 (%, WebShop), with *Avg.* their equal-weight mean. Harness-R1 uses two rollouts (†: author-reported). Bold denotes the best in each column.

Method	ALFWorld	ScienceWorld	DBBench	WebShop	Avg.
Default	40.7	25.7	57.5	34.5	39.6
Prompt-based methods					
ReAct	60.7	26.7	55.8	33.5	44.2
Self-Refine	40.7	21.3	53.3	18.5	33.5
Reflection	50.0	38.2	60.8	41.0	47.5
Harness optimization					
Harness-R1	54.0	28.2	60.0	**42.2**†	46.1
Meta-Harness	60.7	35.1	65.0	40.0	50.2
Turbo Harness (ours)	**70.7**	**42.4**	**69.2**	42.0	**56.1**

5.2 Task Performance

Agentic benchmarks. Turbo Harness improves over Meta-Harness on all four agentic benchmarks (Table 1): by 10.0% on ALFWorld, 7.3 on ScienceWorld's $[0, 100]$ score, 4.2% on DBBench, and 2.0% on WebShop. It also surpasses the three prompt-based baselines (ReAct, Self-Refine, and Reflection) on every benchmark, and outperforms Harness-R1 on ALFWorld, ScienceWorld, and DBBench while performing comparably on WebShop (42.0% versus 42.2%).

SWE-smith-MR. On SWE-smith-MR, instance-specific adaptation produces large gains with both execution models (Table 2): Turbo Harness raises the pass rate from 50.7% to 64.0% with Haiku and from 70.7% to 88.0% with Gemini, improvements of 13.3 and 17.3 percentage points over Meta-Harness. The remaining baselines isolate the harness as the operative lever: averaged across both coding benchmarks, reflective prompt optimization (GEPA) (Agrawal et al., 2026) and context engineering (ACE) (Zhang et al., 2026c) improve over Default but do not reach Meta-Harness (45.9 and 44.3 versus 54.1), let alone Turbo Harness (66.4). A single global harness thus leaves substantial headroom across issues from different repositories, even under a fixed execution model, and adapting the harness per instance recovers much of it.

SWE-bench Verified. The same pattern holds on SWE-bench Verified, a second coding benchmark drawn from real-world GitHub issues, where Turbo Harness again outperforms or matches Meta-Harness. With Gemini, the pass rate rises from 38.4% to 54.4%; with Haiku, it rises from 56.7% to 59.3%, a smaller 2.7-point improvement. The size of the gain tracks the executor's remaining headroom: Meta-Harness already lifts Haiku from 31.6% to 56.7%, capturing most of the improvement over the default harness and leaving little for instance-specific adaptation, whereas Gemini's lower Meta-Harness baseline leaves ample room that Turbo Harness exploits.

5.3 Execution Efficiency

Beyond raising accuracy, Turbo Harness also lowers execution cost in three of the four coding settings, resolving each issue in fewer steps and at lower API cost than the global Meta-Harness (Table 2). The savings are largest on SWE-smith-MR with Gemini, where Turbo Harness averages 8.7 steps per issue rather than 23.1 and \$0.046 rather than \$0.310, a roughly $6.7\times$ reduction in execution cost; with Haiku it uses 17.3 steps rather than 18.5 at \$0.136 rather than \$0.151. On SWE-bench Verified, Gemini shows the same direction, 25.9 steps rather than 32.5 and \$0.331 rather than \$0.470. The exception is Haiku on SWE-bench Verified, the same setting where the pass-rate gain was smallest ($+2.7$ points): there Turbo Harness spends slightly more than Meta-Harness, 20.4 steps rather than 18.8 per issue. All costs in Table 2 cover only the execution model; they exclude both the offline optimization and the one-time per-instance editor call.

Table 2: **Issue resolution and execution efficiency on two coding benchmarks** (frozen Haiku 4.5 and Gemini 3.7 Flash). Per benchmark: pass rate (%, higher is better), average steps, and cost per issue (both lower is better), as mean ± standard error of the mean (SEM) over three runs under a 40-step, $3 budget. Verified is our held-out 150-issue subset. Bold marks the best in each column.

Method	SWE-smith-MR			SWE-bench Verified		
	Pass rate	Steps	Cost ($)	Pass rate	Steps	Cost ($)
Execution model: Claude Haiku 4.5						
Default	42.0 ±4.2	29.5 ±0.8	0.325 ±0.015	31.6 ±0.2	35.5 ±0.2	0.426 ±0.006
GEPA	44.7 ±2.9	25.8 ±0.7	0.308 ±0.014	41.3 ±0.4	27.2 ±1.0	0.352 ±0.013
ACE	51.3 ±1.8	26.8 ±0.4	0.314 ±0.011	34.7 ±1.9	35.5 ±0.4	0.473 ±0.009
Meta-Harness	50.7 ±2.9	18.5 ±0.2	0.151 ±0.004	56.7 ±2.0	**18.8** ±0.5	**0.153** ±0.006
Turbo (ours)	**64.0** ±1.2	**17.3** ±0.5	**0.136** ±0.005	**59.3** ±1.4	20.4 ±0.4	0.179 ±0.006
Execution model: Gemini 3.7 Flash						
Default	44.0 ±5.0	26.6 ±0.9	0.373 ±0.016	24.4 ±0.9	33.5 ±0.2	0.542 ±0.017
GEPA	57.3 ±3.5	17.8 ±0.5	0.213 ±0.026	40.4 ±1.1	29.3 ±0.7	0.433 ±0.016
ACE	56.7 ±3.5	21.6 ±0.2	0.314 ±0.013	34.7 ±0.4	33.4 ±0.3	0.525 ±0.014
Meta-Harness	70.7 ±2.4	23.1 ±0.7	0.310 ±0.015	38.4 ±0.4	32.5 ±0.6	0.470 ±0.012
Turbo (ours)	**88.0** ±1.2	**8.7** ±0.4	**0.046** ±0.004	**54.4** ±2.3	**25.9** ±0.3	**0.331** ±0.010

Table 3: **Terminal-Bench-2.1: performance and efficiency** (frozen Sonnet 4.5, held-out 44-task test split). Pass rate (%) is mean ± SEM over five runs; turns, cost, and input tokens are per-task means. OpenHands, Terminus-Kira, and Terminus-2 are human-designed harnesses; Meta-Harness is automated. Bold marks the best pass rate.

Harness	Pass rate (%)	Turns	Cost ($/task)	Input tokens
OpenHands	47.3 ±1.5	54.4	0.85	1.55M
Terminus-Kira	52.3 ±1.1	72.2	1.42	2.76M
Terminus-2	50.9 ±0.8	75.2	1.54	3.00M
Meta-Harness	50.5 ±1.7	76.3	1.56	3.15M
Turbo Harness (ours)	**55.5** ±1.4	71.2	1.38	2.65M

5.4 Long-Horizon Terminal Tasks

We evaluate Turbo Harness on Terminal-Bench-2.1 (TB2.1), a benchmark for long-horizon terminal tasks, using a frozen Claude Sonnet 4.5 as the execution model. TB2.1 provides a more challenging test of our premise: unlike on the other benchmarks, the model-optimized harness produced by Meta-Harness does not outperform the human-crafted harnesses (Table 3). Even so, per-instance adaptation recovers the lead, as Turbo Harness attains the highest mean pass rate of 55.5%, 5.0 points above Meta-Harness and 3.2 above the strongest default, Terminus-Kira. The gain is consistent, with Turbo Harness posting the top mean in all five runs.

We observe that Turbo Harness lies on the accuracy-cost Pareto frontier. The lightweight OpenHands (Wang et al., 2025) harness is the cheapest overall ($0.85 per task, 54.4 turns, 1.55M input tokens) but also the least accurate (47.3%), spending little because it attempts less. Every harness that is competitive on pass rate costs more, yet Turbo Harness is at once the most accurate and the cheapest among them: relative to Meta-Harness it cuts turns per task from 76.3 to 71.2, cost from $1.56 to $1.38, and input tokens from 3.15M to 2.65M, and it likewise undercuts the other competitive harnesses, Terminus-Kira and Terminus-2. Per-instance adaptation thus buys higher accuracy while trimming cost, whereas the heavier global and default harnesses spend more for less.

5.5 Ablation Studies

We next study what enables Turbo Harness to improve over the globally optimized harness. We focus on two questions: whether RL and the recycled playbook are both necessary, and how the editor model's capability and RL training affect the ability to leverage the playbook.

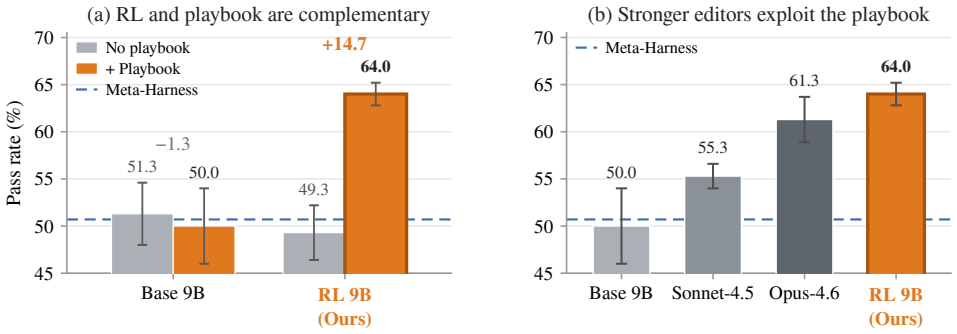


Figure 3: **Ablation on SWE-smith-MR** (frozen Haiku 4.5; pass rate %, mean ± SEM over three runs; dashed line = Meta-Harness). *(a)* For the lightweight Qwen3.5-9B editor, neither the playbook nor RL alone beats Meta-Harness; combined they reach 64.0%. *(b)* Stronger editors exploit the playbook better, yet our small RL-trained editor matches the best.

RL and the playbook are complementary. We first isolate the contributions of RL and playbook conditioning using the same Qwen3.5-9B editor on SWE-smith-MR with a frozen Haiku 4.5 executor (Figure 3 Left). The globally optimized Meta-Harness achieves a pass rate of 50.7%. Introducing an untrained 9B editor without the playbook yields 51.3%, while providing the same editor with the playbook gives 50.0%. Likewise, an RL-trained editor without the playbook reaches only 49.3%. In contrast, combining RL training with playbook conditioning raises the pass rate to 64.0%, a 14.7-point improvement over the same RL-trained editor without the playbook. Thus, for the lightweight editor, neither generic per-instance editing, access to the playbook alone, nor RL alone explains the gain. The improvement emerges when RL trains the editor to identify and apply useful strategies distilled from the outer-loop search.

RL enables a small editor to exploit the playbook. We next vary the editor model while holding the playbook fixed (Figure 3 Right). The untrained Qwen3.5-9B editor achieves 50.0%, indicating that simply exposing a small model to the playbook is insufficient. A stronger frozen Sonnet-4.5 editor improves to 55.3%, while Opus-4.6 reaches 61.3%, showing that sufficiently capable models can better leverage the information stored in the playbook even without task-specific fine-tuning. Using our RL-trained Qwen3.5-9B editor enables the agent to achieve a 64.0% pass rate, matching the performance of Opus-4.6 as the harness editor and outperforming Sonnet-4.5, while remaining small enough to run efficiently on local hardware. These results suggest that the outer-loop search archive already contains useful harness editing strategies, while RL enables a lightweight model to select and apply them effectively.

6 Conclusion, Limitations, and Future Work

We introduced Turbo Harness, an instance-adaptive harness optimization framework that reuses experience from a completed global harness search. Turbo Harness summarizes this search experience into a playbook and trains a lightweight editor to adapt the global harness to each instance. Across our experiments, these instance-specific adaptations consistently outperform the corresponding global harness. Moreover, it provides more flexibility, which we find often translates into lower execution costs.

Our method has two main limitations. First, Turbo Harness builds on a completed global harness search, so its effectiveness depends on the quality and diversity of the harness candidates, trajectories, and feedback collected during that process. If the original search explores only weak strategies or provides limited useful experience, the gains from instance-specific adaptation may also be limited. Second, training the harness editor requires additional environment rollouts. This can be costly on agentic tasks. Future work could reduce this training cost through more sample-efficient optimization or reuse of existing rollout data, and investigate whether editors and playbooks can transfer across tasks, domains, or harness-search runs. More broadly, Turbo Harness suggests that the intermediate artifacts produced during harness optimization can serve as reusable experience rather than disposable byproducts.

AI USE STATEMENT

We wrote the initial draft of this paper ourselves and then used generative AI tools to correct grammatical issues, revise language, and improve the clarity and polish of the writing. We also used AI tools to help identify potentially missing references and to provide preliminary feedback that helps us improve the paper. All AI-generated content and suggestions were carefully reviewed and edited by us. In addition, we used coding agents to assist with implementing our ideas, debugging, and refining our research code. We reviewed and validated the implementations and code generated with AI assistance. We take full responsibility for the final content of this work, including all text, claims, experiments, code, and other artifacts produced with the assistance of generative AI.

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. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026.

Anthropic. Introducing claude haiku 4.5. https://www.anthropic.com/news/claude-haiku-4-5, October 2025a.

Anthropic. Introducing claude sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5, September 2025b.

Anthropic. Effective harnesses for long-running agents. *Anthropic Engineering Blog, Nov*, 2025.

Anthropic. Introducing claude opus 4.6. https://www.anthropic.com/news/claude-opus-4-6, February 2026.

Parth Asawa, Alan Zhu, Abigail O'Neill, Matei Zaharia, Alexandros G Dimakis, and Joseph E Gonzalez. How to train your advisor: Steering black-box llms with advisor models. *arXiv preprint arXiv:2510.02453*, 2025.

Google DeepMind. Gemini 3.7 flash. https://deepmind.google/models/model-cards/gemini-3-7-flash/, August 2026.

Yue Huang, Wenjie Wang, Han Bao, Yuchen Ma, Xiaonan Luo, Yi Nian, Haomin Zhuang, Zheyuan Liu, Yue Zhao, and Xiangliang Zhang. Memoharness: Agent harnesses that learn from experience. *arXiv preprint arXiv:2607.14159*, 2026.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

KRAFTON AI and Ludo Robotics. Terminus-kira: Boosting frontier model performance on terminal-bench with minimal harness, 2026. URL https://github.com/krafton-ai/kira.

Hyunin Lee, Jinglue Xu, Jeffrey Seely, Donghyun Lee, Matei Zaharia, and Yujin Tang. Recursive harness self-improvement. *arXiv preprint arXiv:2607.15524*, 2026a.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026b.

Wenwu Li, Xiangfeng Wang, Wenhao Li, and Bo Jin. A survey of automatic prompt engineering: An optimization perspective. *arXiv preprint arXiv:2502.11560*, 2025.

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.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pp. 52989–53046, 2024.

Yaxin Luo, Haobin Jiang, Jialv Zou, Xu Huang, Wenhao Yan, Haodong Li, Zhengrong Yue, Jing Li, Xiaofu Chen, Xiaohan Zhao, et al. Autodesign: Meta-harness optimization for long-horizon agentic design. *arXiv preprint arXiv:2608.13560*, 2026.

Ziyu Ma, Hailang Huang, Shun Zou, Yong Wang, Shidong Yang, Yiming Hu, Fei Wei, and XiangXiang Chu. Longhorizon-harness: Advancing long-horizon agents for real-world tasks. *arXiv preprint arXiv:2608.01964*, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, 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 Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Jenia Jitsev, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Jun Nie, Yonggang Zhang, Jun Song, Qianshu Cai, Dahai Yu, Yike Guo, Xinmei Tian, and Bo Han. Tthe: Test-time harness evolution. *arXiv preprint arXiv:2607.08124*, 2026.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35: 27730–27744, 2022.

Wenbo Pan, Shujie Liu, Chin-Yew Lin, Jingying Zeng, Xianfeng Tang, Xiangyang Zhou, Yan Lu, and Xiaohua Jia. Evolving agents in the dark: Retrospective harness optimization via self-preference. *arXiv preprint arXiv:2606.05922*, 2026.

Sungho Park, Wonjoong Kim, Rongyuan Tan, Jue Zhang, Wook-Shin Han, Pengfei Gao, Chanyoung Park, Yongqiang Yao, Rao Fu, Elsie Nallipogu, et al. Autosaddler: Automatic harness optimization with durable updates from agent execution traces. *arXiv preprint arXiv:2608.23041*, 2026.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Kiran Ramnath, Kang Zhou, Sheng Guan, Soumya Smruti Mishra, Xuan Qi, Zhengyuan Shen, Shuai Wang, Sangmin Woo, Sullam Jeoung, Yawei Wang, et al. A systematic survey of automatic prompt optimization techniques. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 33066–33098, 2025.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Biswa Sengupta and Jinhua Wang. Harbor: Automated harness optimization. *arXiv preprint arXiv:2604.20938*, 2026.

Shuai Shao, Kangning Zhang, Qingyao Li, Shijian Wang, Hao Wang, Wenxiang Jiao, Yuan Lu, Yi Guo, Weiwen Liu, and Weinan Zhang. Harness-r1: Learning to edit executable runtime harnesses from agent failure trajectories. *arXiv preprint arXiv:2608.02276*, 2026.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*, 2020.

Muxin Tian, Zhe Wang, Zhenwei Tang, Blair Yang, Kunlun Zhu, Honghua Dong, Hanchen Li, Xinni Xie, Guangjing Wang, and Jiaxuan You. Swe-bench mobile: Can large language model agents develop industry-level mobile applications? In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pp. 8077–8087, 2026.

Varun Ursekar, Apaar Shanker, Yash Maurya, Shehab Yasser, Vijay S Kalmath, Veronica Chatrath, and Yuan Xue. Harnessopt-bench: Evaluating llms at harness optimization. *arXiv preprint arXiv:2608.06301*, 2026.

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. Scienceworld: Is your agent smarter than a 5th grader? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 11279–11298, 2022.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Wenyi Wu, Sibo Zhu, Kun Zhou, Aayush Salvi, Zixuan Song, and Biwei Huang. Structagent: Harness long-horizon digital agents with unified causal structure. *arXiv preprint arXiv:2607.11388*, 2026.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 12028–12068, 2024a.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024b. URL https://arxiv.org/abs/2405.15793.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022b.

Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta context engineering via agentic skill evolution. *arXiv preprint arXiv:2601.21557*, 2026.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

Guibin Zhang, Leo Lu, Fangzhou Xie, Kang Zhu, Junhao Wang, Zhifei Xie, Zhaochen Yu, Zihang Liu, Zhongxiang Sun, Qiankun Li, et al. Jit-agent: Scaling harness intelligence via just-in-time harness evolution. *arXiv preprint arXiv:2608.25593*, 2026a.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026b.

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. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026c.

Hailin Zhong and Shengxin Zhu. Ai harness engineering: A runtime substrate for foundation-model software agents. *arXiv preprint arXiv:2605.13357*, 2026.

Chenyu Zhou, Huacan Chai, Wenteng Chen, Zihan Guo, Rong Shan, Yuanyi Song, Tianyi Xu, Yingxuan Yang, Aofan Yu, Weiming Zhang, et al. Externalization in llm agents: A unified review of memory, skills, protocols and harness engineering. *arXiv preprint arXiv:2604.08224*, 2026.

A BENCHMARKS AND DATA SPLITS

A.1 BENCHMARK DESCRIPTIONS

Benchmark descriptions. Our seven benchmarks span three families that differ in interaction modality, horizon, and how much the harness can influence behavior.

- **ALFWorld** (Shridhar et al., 2020) is a text-based embodied environment in which the agent completes household tasks (e.g., pick-and-place, clean, heat, cool, look, and pick-two) by issuing textual actions and observing environment feedback. Success is a binary indicator of whether the goal configuration is achieved.
- **ScienceWorld** (Wang et al., 2022) is an interactive text environment covering many elementary-science task types (e.g., changes of state, electrical conductivity, and life stages), each requiring a multi-step experimental procedure. The environment returns a partial-credit score in $[0, 100]$, and progress is gated by a `focus` action, making it sensitive to procedure and timing.
- **DBBench** (Liu et al., 2024) is a database-interaction benchmark in which the agent answers a question about a relational database by issuing SQL queries and then committing a final answer. Accuracy is measured against the ground-truth answer; we run it against a MySQL backend.
- **WebShop** (Yao et al., 2022a) is a simulated e-commerce environment in which the agent searches and navigates product pages to purchase an item matching a natural-language instruction. The environment returns a dense attribute-match score in $[0, 1]$; we report Success@1, the fraction of goals attaining a score of 1.
- **SWE-smith-MR** is our multi-repository subset of SWE-smith (Yang et al., 2026). Each instance is a bug-fix issue in a Python repository with a hidden test suite; the agent edits the repository and submits a code patch, which resolves the issue if the associated tests pass. Its construction and rationale are described in Appendix A.2.
- **SWE-bench Verified** (Jimenez et al., 2024) is a human-validated set of 500 real-world GitHub issues with associated test suites. As in SWE-smith-MR, an issue is resolved when the agent's submitted patch passes the hidden tests.
- **Terminal-Bench-2.1 (TB2.1)** (Merrill et al., 2026) is a benchmark of long-horizon terminal tasks in which the agent operates a shell to accomplish goals verified by task-specific test scripts. It provides live terminal feedback over long interaction horizons, making the harness an operative lever.

A.2 DATA SPLITS

Data splits. We use the following benchmark-specific splits:

- **ALFWorld:** 100 training games, 169 loadable validation games from `valid_seen`, and 150 loadable test games from `valid_unseen`.
- **ScienceWorld:** 120 training instances, 60 validation instances from held-out development variations, and 149 test instances.
- **DBBench:** a stratified split of 300 tasks into 120 training, 60 validation, and 120 test instances, using seed 42.
- **WebShop:** training goals 1000–1119, validation goals 1500–1559, and 200 test goals (0–199), with `goal_seed`= 233. The reported editor is trained on a filtered subset of the training goals, as described below.
- **SWE-smith-MR:** we construct this subset from the SWE-smith training split by selecting 25 Python repositories and sampling two training and two test issues per repository with seed 42, yielding 50 training and 50 test issues. The issue sets are disjoint, but share the same repositories; this evaluates held-out issues within the selected repositories, not generalization to unseen repositories. We do not use a validation set for this subset.
- **SWE-bench Verified:** 251 training, 99 validation, and 150 test issues, using a repository-stratified split with seed 42. The three issue sets are disjoint and together cover all 500 issues. Meta-Harness search and editor training both use the full 251-issue training split; the playbook

is derived from the training search archive. This is an issue-level split, not a held-out-repository evaluation.

- **Terminal-Bench-2.1:** a difficulty-stratified split of 45 training and 44 test tasks. Harness search, playbook construction, and editor training use only the training split.

A.3 SWE-SMITH-MR CONSTRUCTION AND RATIONALE

SWE-smith-MR design rationale. We construct this subset to evaluate instance-specific adaptation across issues with potentially different execution requirements. The selected repositories span web frameworks, data processing, developer tools, and other Python projects, whose project structures and testing conventions may call for different exploration and validation strategies. This motivates our hypothesis that a single global harness can make tradeoffs across repositories that instance-specific adaptation can help resolve. Sampling the same number of issues from each repository gives each repository equal weight in the evaluation, while holding out issues within these repositories tests adaptation to new issues.

B TRAINING AND EVALUATION PROTOCOL

B.1 PLAYBOOK CONSTRUCTION

Playbook construction. The playbook $\mathcal{P}$ is built once per benchmark and execution model from the completed Meta-Harness search archive, following ACE (Zhang et al., 2026c), and is then held fixed during editor training and inference. Construction proceeds in three offline stages.

First, *extraction* assembles one experience record per training instance from the search archive, without any model calls. Each record collects the per-harness outcomes and full execution traces recorded during search and classifies the instance as *contrastive* (some harness variant solved it while another failed), *uniformly solved*, or *uniformly unsolved*. For contrastive instances, we additionally pair the most dissimilar solving and failing variants and compute the source-level diff between them, which localizes the harness change responsible for the differing outcome.

Second, *reflection* uses a frontier model to analyze each record and emit a short structured reflection: the salient instance characteristics, the behavioral difference between the harness variants, a candidate editing strategy, a rationale, and a confidence score. Contrastive records are analyzed against both traces and the harness diff, whereas uniform records are analyzed to explain why the outcome did not depend on the harness. Reflections that name known anti-patterns or fall below a confidence threshold are discarded.

Third, *curation* consolidates the surviving reflections into the playbook with a single frontier-model call. Semantically equivalent strategies are clustered into one canonical entry that merges their applicability conditions and aggregates helpful and harmful evidence counts and confidence; strategies whose evidence is net-harmful become *anti-patterns*, and vague or over-specified strategies are filtered out. Each retained entry records the condition under which it applies, the strategy text, its evidence and confidence, and a type tag distinguishing edits to the scaffold's control flow (SCAFFOLD_MODIFICATION) from edits to the information shown to the model (TEXT_INJECTION). The result is a compact list of conditional strategies together with a list of anti-patterns and a few general notes; for example, the TB2.1 playbook comprises six strategies and nine anti-patterns. At both training and inference time, the entire playbook is placed in the editor's context, so the editor learns which previously discovered strategies are relevant to each instance rather than retrieving them.

B.2 HARNESS EDITOR TRAINING

Training configuration. All harness editors are Qwen3.5-9B models, full-parameter fine-tuned with FSDP (no low-rank adaptation) using GRPO. Unless noted below, we use a learning rate of 10^{-6}, a KL-penalty coefficient $\beta = 10^{-3}$, within-group advantage standardization, one gradient update per batch with a per-GPU micro-batch of 1, and rollout sampling at temperature 0.8 (top-p 0.999). Rollouts are served with vLLM at tensor-parallel size 1, and generations that do not terminate normally receive zero reward. We checkpoint every 10 steps (5 for TB2.1). At inference,

Table 4: **Harness-editor GRPO training settings per benchmark.** All editors are Qwen3.5-9B, full-parameter fine-tuned; G is the group size (candidate edits per instance). WebShop uses the tuned recipe (see below): it disables within-group standardization (Dr. GRPO), an improvement bonus, and an abstention penalty.

Benchmark	Epochs	Batch	Group G	Max prompt	Max gen.	GPUs
SWE-smith-MR	5	4	8	16384	4096	8
SWE-bench Verified	5	8	16	16384	4096	8
ScienceWorld	3	8	8	24576	8192	4
ALFWorld	3	8	8	16384	8192	4
DBBench	3	8	8	16384	8192	4
WebShop	8	4	16	16384	8192	4
Terminal-Bench-2.1	5	16	16	32768	8192	8

the editor decodes with temperature 0.3 and up to $16{,}384$ tokens. Table 4 lists the per-benchmark settings, where the group size G is the number of candidate edits sampled per instance. The abstention penalty and reward-shaping terms are described in Appendix B.3, and checkpoint selection in Appendix B.4.

WebShop training. The reported WebShop result uses a domain-specific GRPO configuration. We retain the 65 training goals whose precomputed global-harness scores lie strictly between 0 and 1, sample 16 proposals per instance, and train for eight epochs. We disable division by the within-group reward standard deviation when computing advantages and use the improvement bonus and abstention penalty in Appendix B.3. This configuration differs from the training recipe used for the other benchmarks.

B.3 Reward functions

Reward evaluation. We evaluate each valid editor proposal with one execution of the frozen model. A valid patch runs the adapted harness $\mathtt{H}_x$, whereas an explicit `NO_PATCH_NEEDED` response runs the global harness $\mathtt{H}^\star$. Malformed proposals and patches that fail application or validation receive zero reward without an execution. The training rewards below include shaping terms; the reported test metrics exclude these terms.

Task rewards. Before applying the abstention penalty described below, we compute rewards as follows:

- **SWE-smith-MR and SWE-bench Verified:** $1 - s/(2L)$ if the submitted code patch resolves the issue, and 0 otherwise, where s is the number of execution-model steps and $L = 40$ is the step limit. This reward favors successful fixes with fewer steps.
- **ScienceWorld:** the environment score divided by 100, with negative scores clamped to zero.
- **ALFWorld:** the binary task-success indicator in $\{0, 1\}$.
- **DBBench:** the binary task-correctness indicator in $\{0, 1\}$.
- **Terminal-Bench-2.1:** the binary task-success indicator returned by the task verifier.
- **WebShop:** the environment's dense score in $[0, 1]$, plus a bonus of 0.1 when a non-abstaining proposal yields a score greater than the precomputed global-harness score for the same training goal. This bonus rewards improvements over the global harness. Evaluation uses Success@1, defined as the fraction of goals that attain an environment score of 1.

Abstention penalty. On ScienceWorld, ALFWorld, DBBench, WebShop, and TB2.1, we subtract 0.05 from the reward of an explicit abstention and clamp the result at zero. Thus, if the unpatched harness earns task reward r, the abstention reward is $\max(0, r - 0.05)$. This selective penalty encourages exploration of edits while retaining abstention as an option. Neither coding benchmark uses an abstention penalty.

Reward integrity. We compute task outcomes outside the editor-generated harness code. On the four agentic benchmarks, a trusted runner obtains outcomes directly from the environment, while

the harness interacts through an interface that omits scoring methods. Source checks also reject references to protected scoring internals. On both coding benchmarks, the harness executes in a separate process and the parent process scores the submitted code patch. These safeguards reduce opportunities for a harness to manipulate its training reward.

B.4 Checkpoint selection

Checkpoint selection. We select editor checkpoints using validation performance on ALFWorld and ScienceWorld. For SWE-smith-MR and DBBench, we report the final checkpoint at step 60; for the tuned WebShop run, we report the final checkpoint at step 64. For TB2.1, we report the step-5 checkpoint, at which training was stopped. On SWE-bench Verified, we select by pass rate from one execution run on the 99 validation issues after excluding checkpoints with collapsed policy entropy. This selects step 20 for Gemini and step 10 for Haiku. For Haiku, step 60 has the highest validation pass rate (52.5%), but is excluded because its training policy entropy has fallen to 0.066, compared with approximately 0.45–0.59 for steps 10–50. Among these remaining checkpoints, step 10 has the highest validation pass rate (49.5%). This diagnostic exclusion is part of the reported selection rule; selection is not an unrestricted maximum over validation scores.

B.5 Execution and scoring protocol

Coding execution. Both coding benchmarks use the mini-swe-agent scaffold (Yang et al., 2024b) in Docker, with a limit of 40 execution-model steps and a $3 cost limit per issue. Only explicitly submitted code patches are evaluated for issue resolution, and the pass-rate denominator includes every test issue. For Turbo Harness, we generate one adapted harness per issue and reuse it across the three execution runs. We compute each run's pass rate, mean steps, and mean execution-model cost over the complete test set, then report the mean and sample standard deviation divided by $\sqrt{3}$ across runs. The Verified subset and execution budget differ from full-benchmark leaderboard settings, so our absolute rates should not be read as leaderboard results.

Result aggregation. We aggregate results over repeated execution runs where available. For the coding benchmarks, we report the mean and standard error of the mean (SEM) over three runs on the same test issues, with Turbo Harness reusing the generated per-instance harnesses across runs. For TB2.1, pass rates and efficiency metrics report the mean and SEM over five runs. These error bars quantify variability across execution runs; variation across editor training seeds and data splits is not evaluated. Each of the four agentic benchmarks uses one evaluation run per test set.

C Baseline Details

C.1 Default harnesses

Coding Default. For SWE-smith-MR and SWE-bench Verified, Default runs mini-swe-agent's `DefaultAgent` with the system, task, observation, and format-error templates from its default configuration. The model receives the issue description, generates a shell command, and observes its output in a Docker environment; this loop continues until explicit submission or the execution limit. The prompt instructs the model to inspect the repository, edit the code, and verify the fix. We set the same 40-step and $3 execution limits used for the other coding harnesses. No optimized harness or per-instance harness edit is supplied to this baseline.

Agentic Defaults. Each agentic Default is a fixed, zero-shot tool-calling loop in our task runtime. It initializes the model with the domain's system prompt and task observation, appends actions and environment feedback to the conversation, and re-prompts when the model fails to produce the required tool call. The domain-specific interfaces are:

- **ALFWorld:** a `take_action` tool, with admissible actions shown at each step and model outputs canonicalized to an admissible action.
- **ScienceWorld:** a `take_action` tool accepting textual environment commands, with action-template hints in the initial prompt and subsequent observations truncated to 800 characters.
- **DBBench:** `execute_sql` and `commit_final_answer` tools, initialized with the question and database schema; committing the answer ends the rollout.

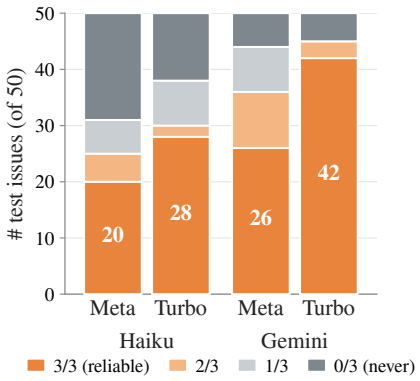


Figure 4: **SWE-smith-MR reliability.** The 50 test issues grouped by success count over three runs. Dark orange: all three succeed; dark gray: none.

- **WebShop:** `search_action` and `click_action` tools, with the page observation and available actions supplied after each interaction.

C.2 Prompt-based baselines

Prompt-based variants. Our ReAct implementation augments the basic tool-calling loop with reasoning–action demonstrations and an explicit instruction to produce a `Thought` alongside each action. This distinguishes the ReAct row from the zero-shot Default. Self-Refine adds a proposal–critique–revision cycle before executing an action. Reflection runs an initial episode and, if it is not fully successful, generates a verbal reflection to guide a second episode. It reports the better outcome across the two attempts, whereas Default and Turbo Harness each use a single execution rollout. These variants therefore differ in both interaction strategy and model-call budget.

C.3 Harness-R1 comparison

Harness-R1 comparison. We evaluate the released Harness-R1 checkpoint in its AgentBench runtime on our ALFWorld and DBBench test splits, because we could not reconstruct its exact evaluation sets from the public release. For WebShop, we use the authors' reported result on the same seed-233, 200-goal test set. Because ScienceWorld is not one of Harness-R1's benchmarks, we port its code-hook method into the same runtime as a new task and RL-train the released engineer on ScienceWorld with our optimizer; the reported value (28.2) is this RL-trained engineer, and the zero-shot released checkpoint scores comparably (29.7). Harness-R1 uses an initial rollout followed by a patched rerun, whereas Turbo Harness proposes its patch before a single execution rollout. The comparison therefore uses different runtimes and rollout budgets; it does not isolate the effect of the harness editor alone.

D Additional Results and Analyses

D.1 Reliability across repeated executions

Repeated-execution reliability. On SWE-smith-MR, Turbo Harness increases the number of issues resolved in all three runs (Figure 4). This count rises from 20 to 28 out of 50 issues with Haiku and from 26 to 42 with Gemini. At the same time, the number resolved in only one or two runs falls from 11 to 10 and from 18 to 3, respectively. Thus, the higher mean pass rates are accompanied by more consistently solved issues, especially with Gemini. This analysis measures execution consistency for fixed per-instance harnesses; it does not measure variability in editor training or harness generation.

D.2 QUALITATIVE CASE STUDIES OF INSTANCE-SPECIFIC PATCHES

To illustrate what the harness editor actually changes, we show representative patches. The three SWE examples are recovered verbatim from the reported runs and paired with the held-out test outcome; the Terminal-Bench-2.1 example is an outer-loop (search-stage) edit on a training task, included only to illustrate the mechanism because per-instance test patches were not persisted for that benchmark. Diffs are elided (`...`) but otherwise verbatim.

Same global harness, opposite per-instance guidance (SWE-smith-MR). On two issues from different libraries, the editor patches the same regions of the *verify-once-submit* $H^\star$ with different guidance, and both flip from failing to fully solved ($3/3$). For a Jinja2 regression it injects a history-inspection step:

```
  ## Recommended Workflow
  1. Analyze the codebase by finding and reading relevant files.
+    For regression bugs, check git history to understand what
+    changed before fixing.
+ ### Investigate changes (for regression bugs):
+ git log --oneline -20
+ git diff HEAD~1..HEAD --stat
```

whereas an `arrow` (datetime library) use-before-assignment bug receives unrelated guidance rather than the history-inspection step. The same $H^\star$ is thus specialized differently per instance.

Reallocating the step budget (SWE-bench Verified). On the `django-13315` issue, whose $H^\star$ imposes an explicit per-phase step budget, the editor relaxes an aggressive "submit immediately" gate to buy exploration and a real fix-and-verify budget:

```
- ### Phase 1: Understand (5-10 steps)
+ ### Phase 1: Understand (10-15 steps)
- ### Phase 3: Execute Fix (1 step)
+ ### Phase 3: Execute Fix (5-10 steps)
- ### Phase 4: Submit IMMEDIATELY (1 step)
+ ### Phase 4: Finalize and Submit (5-10 steps)
```

The outcome improves from $1/3$ (Meta-Harness) to $3/3$ (Turbo Harness). The budget is instance-specific: on `sympy-16766` (also $1/3 \rightarrow 3/3$) the editor enlarges only Phase 3 and *keeps* the one-step submit gate.

Editing executable loop code, not prompt wording (SWE-smith-MR). On a `gunicorn` issue, the editor rewrites the scaffold's control code rather than its text. It fires the verification gate one step earlier and teaches the edit detector to also recognize append redirection as a source edit:

```
  if (not self.verification_triggered and self.has_edited_source
-     and self.step_count >= 8
+     and self.step_count >= 6
      and self._is_verification_command(self.last_action)):
          observation += submit_reminder
  ...
  patterns = [ r'sed\s+.*\.py', r'cat\s+.*>\s*\S+\.py',
+              r'echo\s+.*>>\s*\S+\.py' ]
```

The instance flips from $0/3$ to $3/3$, and the threshold is set per instance (5, 6, or 10 across different bugs) rather than as a fixed global knob, confirming that adaptation reaches the executable harness and not only the prompt.

One knob, opposite optimum per task (Terminal-Bench-2.1). The winning $H^\star$ escalates to forced tool use only after two stalled turns. On `chess-best-move`, the model instead spends turns reasoning in prose, so an edit that forces a tool call every turn solves the task:

```
  completion_kwargs = {"model": ..., "tools": TOOLS,
+                      "tool_choice": "required",
                       ...}
```

Yet forcing tool use *globally* is a playbook anti-pattern that regresses the whole suite by 6.7%, and `extract-elf` is instead solved by the adaptive escalation and fails under the global forced setting. The same control knob therefore has opposite optima on different tasks, which a single global harness cannot satisfy but instance-specific adaptation can. (These are search-stage edits on training tasks, shown only to illustrate the mechanism.)