AutoSaddler: Automatic Harness Optimization with Durable Updates from Agent Execution Traces

Sungho Park[1]* **Wonjoong Kim**[2]* **Rongyuan Tan**[3]* **Jue Zhang**[4]† **Wook-Shin Han**[1]†
Pengfei Gao[4] **Chanyoung Park**[2] **Yongqiang Yao**[4] **Rao Fu**[4] **Elsie Nallipogu**[4]
Qingwei Lin[4] **Saravan Rajmohan**[4] **Dongmei Zhang**[4]

[1]POSTECH [2]KAIST [3]Southern University of Science and Technology [4]Microsoft
juezhang@microsoft.com wshan@dblab.postech.ac.kr

Abstract

LLM agents remain unreliable on long-horizon tasks, where small local failures can compound over extended interactions and lead to overall task failure. Although external harnesses can substantially improve robustness, harness design remains a manual and expensive process that requires searching over a large space of prompts, tool configurations, and control logic. We propose **AutoSaddler**, an automatic harness optimization framework that formulates harness improvement as an offline learning problem and iteratively updates the harness using failure signals from mini-batches. AutoSaddler combines failure-trace diagnosis, structured patch generation that treats the harness as code, and validation-based update selection. Experiments on GAIA2, SWE-Bench Pro, and Terminal-Bench 2.0 show that AutoSaddler substantially improves agent performance over the corresponding base harnesses, achieving gains of **9.0, 9.6, and 10.0** percentage points, respectively. Ablation studies further suggest that effective harness optimization benefits from three ingredients: deep debugging rather than shallow reflection, targeted modifications rather than unconstrained editing, and generalization-aware selection rather than trajectory-specific repair. Together, these results suggest that automatic harness optimization is a promising path toward more performant and reliable agent systems. Project website and code will be available at https://aka.ms/AutoSaddler-website.

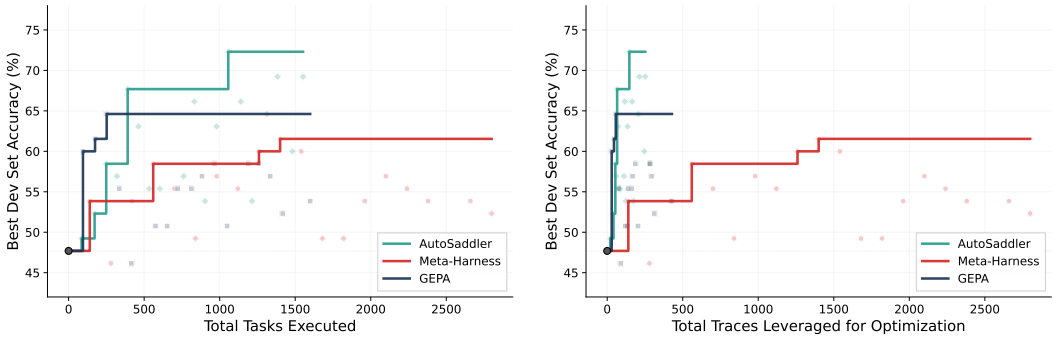
1 Introduction

Large Language Models (LLMs) have improved rapidly in capability in the past years. Despite this progress, they continue to exhibit "jagged intelligence": strong performance on some tasks coexists with poor performance on others that appear closely related or even simpler. This unevenness creates a fundamental reliability challenge for agentic applications, especially in autonomous, multi-step, and long-horizon settings, where success depends on sustained competence across many consecutive decisions. In response, an emerging line of work seeks to build external harness layers around LLMs to make agent behavior more robust on long-horizon tasks [61, 3, 5, 31, 21]. Empirical evidence suggests that a well-designed harness can substantially improve agent performance [21].

Manual harness tuning, however, is time-consuming and difficult to scale. It requires exploring a large design space, including prompt specifications, tool configurations, and other system-level choices. Moreover, evaluating each candidate harness is costly, since the agent may need to execute many steps before task success or failure becomes clear. Analyzing the resulting long-horizon trajectories is itself nontrivial and often requires substantial manual effort. These challenges create a strong need

*Work done during an internship at Microsoft.
†Corresponding authors.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).



(a) Compute Efficiency

(b) Learning Efficiency

Figure 1: **Comparison of optimization performance and efficiency on GAIA2. (a)** AutoSaddler reaches **72.3%** dev accuracy with ∼**1,000** total task executions, whereas GEPA and Meta-Harness saturate at **64.6%** and **61.5%**, respectively, despite consuming ∼**2,800** task executions. **(b)** When measured by the number of execution rollouts leveraged for optimization, AutoSaddler achieves its best performance after consuming only 147 traces, ∼**10**× fewer than Meta-Harness (1,400 traces).

for automated harness optimization methods that can efficiently adapt the harness when switching to a new underlying LLM or deploying the agent in a new domain.

In this work, we formulate automatic harness optimization as an offline learning problem and introduce **AutoSaddler**, a framework that iteratively refines the harness using failure signals from batches of training tasks. At each iteration, we first construct a candidate harness from the accumulated exploration history and evaluate it on the training examples in the current mini-batch. We then diagnose failed agent trajectories and use the resulting insights to guide patch generation, treating the harness itself as code. For each accepted harness update, we further assess generalization on a validation set. Together, these components enable AutoSaddler to optimize harnesses in a principled and scalable manner, while promoting updates that generalize beyond the observed training samples.

We evaluate AutoSaddler on three agent benchmarks: GAIA2 [10], SWE-Bench Pro [9], and Terminal-Bench 2.0 [28]. Across all three benchmarks, AutoSaddler substantially improves long-horizon agent performance. It surpasses the corresponding base harness by **9.0** percentage points on GAIA2, **9.6** points on SWE-Bench Pro, and **10.0** points on Terminal-Bench 2.0, and exceeds the strongest automated baseline by **7.4**, **4.4**, and **6.7** points, respectively. Ablations and qualitative analysis further support the design rationale of AutoSaddler, showing that effective harness optimization depends on three key ingredients. First, **in-depth diagnosis** is necessary because long-horizon failures require deep debugging rather than shallow reflection on agent execution failures. Second, **structured intervention** is important because the large and diverse harness space favors targeted modifications over unconstrained editing. Third, **generalization-aware selection** is critical because improving performance over a task distribution requires retaining broadly useful updates rather than repairing a single trajectory.

Our main contributions are summarized as follows:

- We formulate automatic harness optimization as an offline learning problem over prompts, tools, and middleware, leveraging failure signals from agent execution traces.
- We introduce AutoSaddler, which combines evidence-grounded diagnosis, structured patching, and generalization-aware selection to produce durable harness updates.
- We conduct extensive experiments on three challenging agent benchmarks, demonstrating the effectiveness of the proposed approach in improving long-horizon agent performance.

2 Related Work

Auto Prompt Optimization. Automatic optimization of LLM-based systems has been extensively studied in the context of prompt optimization, including gradient-free search [62], textual-gradient and mini-batch methods [36], history-aware and evolutionary optimization [47, 14, 1], planning-

based search [43], programming abstractions [18], autodiff-style optimization [50], and Bayesian multi-prompt tuning [32]. While AutoSaddler draws on this line of work, harness optimization is broader than prompt optimization: long-horizon traces often require deeper diagnosis than simple reflection, and the search space spans prompts, tools, and runtime control logic.

Self-Evolving Agents and Experience-Based Improvement. Learning from failure traces is related to experience-based agent improvement [60] and self-evolving agents [2], where experience is used to build libraries [6], create tools [37, 45], and construct memory, skills, or knowledge bases [33, 19, 29]. Unlike these typically online continual-learning approaches, we focus on offline harness optimization for generalization across environments.

Some prior work further emphasizes self-referentiality, i.e., whether the meta-agent coincides with the task agent, as in Darwin-Gödel Machines [54, 55], Gödel agents [49], and Huxley-Gödel Machines [42]. In contrast, we do not impose such constraints, as our objective is to improve the task agent's external harness under a finite rollout budget rather than to model self-referential dynamics.

Agent Systems and Harness Optimization. Automatic optimization of the external layers of LLM-based systems builds on a broad line of prior work, including LLM inference hyperparameter tuning [41, 11], workflow optimization [56, 64, 44], and system optimization by treating the system itself as code [59, 16]. More recently, this line of work has expanded to optimizing the full agent harness [22, 25, 34, 40]. One prominent work is Meta-Harness [22], which likewise employs a coding agent to build an end-to-end optimization pipeline. In contrast, we place greater emphasis on structured patching and on a systematic training pipeline that explicitly accounts for generalization.

Recent contemporaneous studies [3] have further expanded automatic harness optimization through observability-driven evolution [23], diagnosis- and weakness-guided repair [7, 52], Bayesian configuration search [38], historical-experience recalibration [13], self-supervised retrospective optimization [35], generalization-oriented constrained evolution [57], experience-driven test-time adaptation [17], and joint optimization of harnesses and model weights [15, 8]. Collectively, these works explore complementary aspects of harness optimization, while AutoSaddler emphasizes the joint roles of in-depth diagnosis, structured intervention, and generalization-aware selection.

Agent Trace Failure Diagnosis and Repair. Long-horizon agent traces with complex tool interactions have motivated work on failure diagnosis [58, 51, 26, 12, 53]. These methods primarily target single-trace diagnosis and often assume traces fit within the LLM context window, limiting their applicability to very long traces and mini-batch diagnosis. We instead build on the Claude Agent SDK (CA-SDK) [4], leveraging its file access and context management capabilities. Recent work also studies trace-level interventions for repairing failed runs [27, 63]. Unlike such task-specific "hot fixes", our goal is to produce persistent harness repairs that improve future behavior.

LLM-Driven Evolutionary Search. Our candidate selection method uses LLMs to select and recombine patches from prior candidates based on exploration history, aligning with recent LLM-augmented evolutionary algorithms [1, 30, 39, 46, 24]. We instantiate this idea with CA-SDK-based candidate selection, leaving specialized LLM-driven evolutionary algorithms for future work.

3 Preliminaries

Agentic Task and Execution Trace. We consider agentic tasks that require multi-step reasoning, repeated tool use, and environment interaction. Let $x \in \mathcal{X}$ be a task description sampled from a task distribution $\mathcal{T}$, and assume that task instances in $\mathcal{X}$ are stateless and independent.

Conditioned on a harness parameter θ, executing the task agent on a task x induces a stochastic execution process. We write $(\tau, \hat{y}) \sim P_\theta(\cdot \mid x)$, where τ is the execution trace and $\hat{y}$ is the final output. Equivalently, the harness H_θ defines the conditional execution distribution $P_\theta(\tau, \hat{y} \mid x)$. This stochasticity captures randomness from LLM sampling, tool-use decisions, and environment interactions.

[3] These works appeared during the final stage of this work preparation or the subsequent peer-review period.

Optimization Space: Agent Harness. Although the term *harness* is often used colloquially to refer to everything outside the LLM that contributes to effective agent behavior, its precise scope is still evolving in the community [61]. In this work, following harness engineering practices in LangChain [21], we focus on three classes of harness parameters and do not consider other components, such as memory or skill curation, since our setting assumes tasks are largely stateless and independent. Formally, we define the optimization space as:

$$\theta = (\theta_{\text{prompt}}, \theta_{\text{tool}}, \theta_{\text{middleware}}) \in \Theta, \tag{1}$$

where θ_{prompt} specifies the instructions and system prompts, θ_{tool} defines the available tools and their interfaces, and $\theta_{\text{middleware}}$ captures runtime control logic, including hooks and agent-loop behavior.

Objective: Budget-Constrained Optimization. Our population objective is to maximize expected task performance over the target task distribution. Let $\mu(\hat{y}, y^*)$ be a task-level metric, such as accuracy or pass/fail success, that evaluates the final output $\hat{y}$ against the gold answer y^*. Since executing a harnessed LLM agent may be stochastic, we define

$$J(\theta) = \mathbb{E}_{(x,y^*)\sim\mathcal{T}}\mathbb{E}_{(\tau,\hat{y})\sim P_\theta(\cdot|x)}[\mu(\hat{y}, y^*)], \qquad \theta^* = \arg\max_{\theta\in\Theta} J(\theta),$$

where θ^* denotes the optimized harness.

In practice, $J(\theta)$ is not directly observable and each execution consumes a rollout. Given a rollout budget K, AutoSaddler searches over a set of candidate harnesses $\mathcal{V}_K \subset \Theta$ explored within that budget. For a finite dataset D, we use the empirical estimate

$$\widehat{J}_D(\theta) = \frac{1}{|D|}\sum_{(x_i,y_i^*)\in D}\frac{1}{R_i}\sum_{r=1}^{R_i}\mu(\hat{y}_{i,r}, y_i^*), \qquad (\tau_{i,r}, \hat{y}_{i,r}) \sim P_\theta(\cdot \mid x_i),$$

where R_i is the number of repeated executions for task i. AutoSaddler returns the candidate with the highest *development-set* empirical score among candidates evaluated within the rollout budget:

$$\hat{\theta}_{\text{AS}} = \arg\max_{\theta\in\mathcal{V}_{K,\text{dev}}} \widehat{J}_{D_{\text{dev}}}(\theta).$$

The final reported test performance is then $\widehat{J}_{D_{\text{test}}}(\hat{\theta}_{\text{AS}})$.

4 Proposed Method: AutoSaddler

We now present AutoSaddler, an iterative framework for automatic harness optimization of LLM agents. We begin with an overview of the workflow and then describe each key component in detail.

Overview. We formulate harness optimization as an offline learning problem. This formulation reflects the practical setting in which agent harnesses are typically tuned during development before being deployed to production. Since training may involve many tasks and rollout-based evaluation is expensive, we adopt a mini-batch training paradigm, following standard machine learning practice and prior work on automatic prompt optimization [36, 1].

Specifically, we partition the task set $\mathcal{X}$ into training, development, and test splits, denoted by D_{train}, D_{dev}, and D_{test}, respectively. As illustrated in Figure 2, each iteration n begins by evaluating the current harness H_n, parameterized by θ_n, on a mini-batch $B_n \subset D_{\text{train}}$ (❶). The workflow then enters the **Diagnosis–Patch Session**, which analyzes failed execution traces in the mini-batch and derives a structured patch $\Delta\theta_n$, producing an updated harness $H'_n = H_n + \Delta\theta_n$ (❷). The patched harness is verified on the same mini-batch (❸); we regard the patch as a mini-batch improvement if $\widehat{J}_{B_n}(H'_n) > \widehat{J}_{B_n}(H_n)$. When this criterion is satisfied, we further evaluate H'_n on D_{dev} to estimate whether the update generalizes beyond the observed mini-batch (❹).

Regardless of whether the patch is accepted, the **Reflection Session** compares pre- and post-patch traces; records fixed, regressed, still-failing, and still-passing cases (❺); and stores the resulting lessons and scores in **EvoDAG**, a directed acyclic graph that tracks the evolution history of harness updates (❻). The **Evolution Session** then uses EvoDAG to propose the next candidate harness H_{n+1} (❼). After the rollout budget K is exhausted, AutoSaddler returns the candidate with the highest

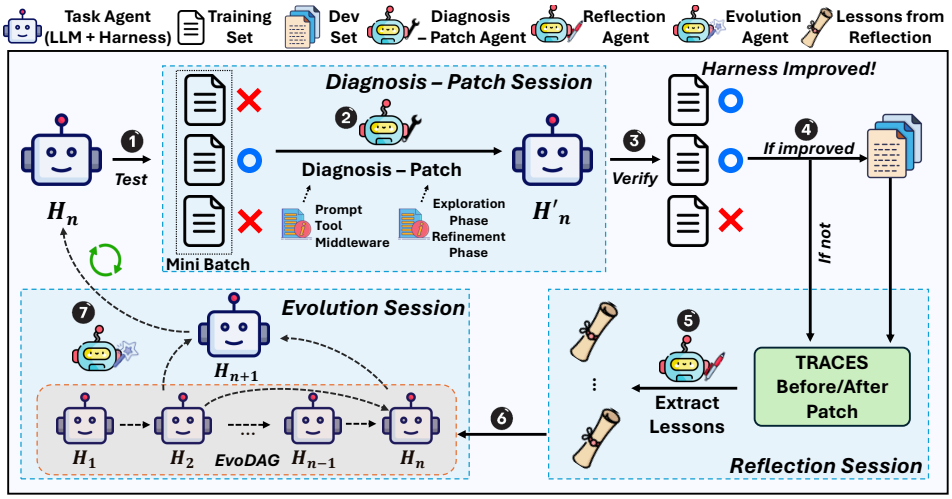


Figure 2: **Overview of AutoSaddler**. The iterative optimization loop: the current harness is tested on a mini-batch, diagnosed and patched across harness components, verified for improvement, and then reflected upon to extract lessons into the EvoDAG, which guides the evolution of the next harness.

empirical development-set score among all candidates evaluated on D_{dev}. The selected harness is then evaluated once on the held-out test set and is not further modified using test feedback.

This workflow mirrors standard mini-batch learning, but adapts it to textual harness optimization. The Diagnosis–Patch–Verification steps (❷ and ❸) play the role of backpropagation under textual gradients: because textual error signals are not automatically checked like numerical gradients, each update requires explicit hypothesis generation, harness intervention, and empirical verification. EvoDAG and the Evolution Session play an optimizer-like role by accumulating reflection signals to perform history-aware harness evolution. We detail this analogy and its discrepancies in Appendix A.

Diagnosis-Patch Session. During this session, the execution traces from the mini-batch, including both successful and failed runs, are passed to the *Diagnosis-Patch Agent* for failure diagnosis and patch generation. We do *not* separate diagnosis from patch generation, allowing the agent to fully leverage the contextual information gathered during diagnosis when producing patches. Our diagnosis procedure is designed to be principled and evidence-based. In addition to each failure trace, we provide the agent with the harness codebase θ_n and structured guidance that enables it to progressively retrieve relevant trace details, thereby mitigating the long-context challenges posed by long agent trajectories. Based on this evidence, the agent identifies the suspected root cause, considers alternative hypotheses, and proposes recommendations for patch generation.

Patch generation is likewise structured. Specifically, the agent produces a *structured patch* $\Delta\theta_n$ that targets one or more harness components. Rather than allowing unconstrained edits over the full harness codebase, we expose only the source files that implement the harness's functional logic, while prohibiting access to code related to evaluation or benchmark data. We further organize the patch space into three categories: *Prompt*, *Tool*, and *Middleware*, each corresponding to a distinct layer of the agent harness. Table 1 enumerates the subtypes of each category and provides their descriptions.

We further divide these patch types into two higher-level groups: **Capability Patches** and **Steering Patches**. Capability Patches modify executable code or orchestration logic, whereas Steering Patches consist of textual edits that leave the underlying code unchanged, including modifications to prompts, tool descriptions, and hook reminder texts. Since these two patch groups may affect the harness in different ways, we introduce **Phased Patch Scheduling**, analogous to learning-rate scheduling in standard model training. In this schedule, optimization begins with a Capability Patch phase and then transitions to a Steering Patch phase. Additional details on Phased Patch Scheduling are provided in Appendix A, while the diagnosis and patch-generation instructions are described in Appendix P.

Reflection Session. After evaluating the patched harness H'_n on the same mini-batch, and optionally on the development set, we enter the *Reflection Session* to extract lessons from the current harness

Table 1: Adopted patch categories, with subtypes labeled as Capability (C) or Steering (S) Patch.

Category	Subtype	Description
Prompt Patch	Prompt Rule Addition (S)	Add or modify behavioral rules in the system prompt.
	Prompt Rule Modification (S)	Revise existing prompt rules to resolve conflicts or sharpen guidance.
Tool Patch	New Tool Addition (C)	Add a new tool when no existing tool supports the required action.
	Argument Modification (C)	Add or fix tool parameters to enhance filtering or selection capabilities.
	Implementation Fix (C)	Fix bugs or extend internal tool functionality to produce correct results.
	Tool Description Fix (S)	Modify tool docstrings to avoid tool misuse.
Middleware Patch	PreToolUse Hook (S)	Inject a just-in-time reminder before a specific tool call.
	Infrastructure Change (C)	Modify agent configuration, iteration budget, or environment settings.
	Agent Loop Logic Change (C)	Add preprocessing steps or budget reminders to the agent loop.

update. Specifically, the *Reflection Agent* is instructed to compare agent performance before and after applying the patch on the mini-batch. The resulting outcomes are grouped into four categories: *"fixed"*, *"regressed"*, *"still-failing"*, and *"still-passing"*. For each category, we provide targeted self-reflection questions to elicit insights into why the patch was effective, which failure patterns it addressed, why regressions occurred, why the patch was insufficient, or whether the patch had any effect at all. The agent may also inspect trace details during reflection. Furthermore, if the patch triggers an extended evaluation on the development set, we additionally prompt reflection on the development-set performance, with particular emphasis on whether and how the patch generalizes beyond the mini-batch. These lessons, together with the patch description, mini-batch results, and evaluation metric results $\mu(H'_n)$, are stored as node-level attributes in the EvoDAG to facilitate harness evolution. More details on the reflection instructions can be found in Appendix P.

Evolution Session. The EvoDAG is a directed acyclic graph $\mathcal{G} = (V, E)$ that serves as the cumulative memory of the optimization process. Each node $v_n \in V$ corresponds to a previously explored harness and is annotated with its associated lessons and performance signals. Each directed edge $e \in E$ represents the diff $\Delta\theta$ between a parent harness and its descendant.

During the Evolution Session, the *Evolution Agent* consults the full EvoDAG to synthesize a new harness H_{n+1}. Rather than continuing solely from H'_n, the agent can compose elements from any subset of previously explored harnesses, guided by the accumulated lessons stored in the graph. This merge operation is analogous to evolutionary search, enabling the framework to escape local optima by recombining successful components across different lineages in the optimization history. The resulting harness H_{n+1} is then evaluated on a fresh mini-batch in the next iteration. Further details of the evolution instructions are provided in Appendix P.

Design Rationale. Overall, AutoSaddler is designed around three requirements for effective automatic harness optimization: *in-depth diagnosis*, *structured intervention*, and *generalization-aware selection*. The Diagnosis-Patch Session supports in-depth diagnosis by grounding updates in explicit analysis of long-horizon execution traces and the harness codebase, enabling deep debugging rather than shallow reflection. The patch taxonomy and phased schedule support structured intervention by restricting optimization to targeted changes over prompts, tools, and middleware, rather than unconstrained editing. The Reflection and Evolution Sessions support generalization-aware selection by combining mini-batch verification, optional development-set evaluation, and EvoDAG-based memory, so that retained updates are more likely to generalize beyond a single trajectory. Together, these components turn harness optimization into an iterative process of diagnosing failures, applying targeted updates, and retaining changes that are more likely to generalize.

5 Experiments

5.1 Experimental Setup

Benchmarks and Base Harness Systems. We evaluate AutoSaddler on three diverse benchmarks: GAIA2 [10], SWE-Bench Pro (SBP) [9], and Terminal-Bench 2.0 (TB2) [28]. GAIA2 evaluates general-purpose assistant capabilities in a simulated smartphone environment spanning 10 distinct *Universes*, each emulating a persona's daily digital life. We use the default ReAct-based agent

Table 2: Test-set Pass@1 results on GAIA2, reported as mean ± standard deviation over three runs. Values in parentheses indicate the number of tasks; boldface denotes the best result.

Harness (Type)	GAIA2 Universe (Pass@1)			Avg.
	21 (107)	22 (112)	27 (81)	
Default Agent (Manual)	54.8 ± 4.8	51.5 ± 4.9	52.7 ± 4.3	53.0 ± 1.5
GEPA (Auto)	60.1 ± 3.9	47.9 ± 3.4	56.4 ± 0.7	54.6 ± 2.5
Meta-Harness (Auto)	53.0 ± 1.1	51.5 ± 5.2	56.0 ± 0.7	53.2 ± 2.2
AutoSaddler (Auto)	$\mathbf{61.4 \pm 2.4}$	$\mathbf{60.7 \pm 2.4}$	$\mathbf{64.6 \pm 3.1}$	$\mathbf{62.0 \pm 1.2}$
w/o In-depth Diagnosis (Auto)	56.7 ± 4.2	57.1 ± 3.9	60.1 ± 5.0	57.8 ± 3.8
w/o Structured Intervention (Auto)	58.9 ± 2.5	53.3 ± 7.6	59.3 ± 3.3	56.9 ± 3.8
w/o Generalization-Aware Selection (Auto)	53.3 ± 1.9	44.9 ± 6.7	54.7 ± 4.0	50.6 ± 4.0

Table 3: Test-set Pass@1 results on SWE-Bench Pro and Terminal-Bench 2.0, reported as mean ± standard deviation over three runs. Parentheses indicate the number of tasks; bold denotes the best result per column.

SWE-Bench Pro (Pass@1)

Harness (Type)	`Ansible` (96)	`Flipt` (85)	`Element-web` (56)	Avg.
SWE-agent (Manual)	40.6 ± 1.9	31.0 ± 3.5	41.1 ± 1.7	37.3 ± 4.8
GEPA (Auto)	50.0 ± 1.3	32.2 ± 1.4	$\mathbf{45.2 \pm 0.6}$	42.5 ± 1.2
Meta-Harness (Auto)	36.9 ± 2.9	31.3 ± 1.7	38.7 ± 0.8	35.3 ± 2.0
AutoSaddler (Auto)	$\mathbf{58.0 \pm 1.8}$	$\mathbf{36.5 \pm 1.8}$	43.5 ± 1.6	$\mathbf{46.9 \pm 1.8}$

Terminal-Bench 2.0 (Pass@1)

Harness (Type)	Test-Split (40)
Terminus 2 (Manual)	40.0 ± 0.0
Terminus KIRA (Manual)	47.5 ± 2.5
GEPA (Auto)	42.5 ± 2.5
Meta-Harness (Auto)	43.3 ± 5.8
AutoSaddler (Iter2) (Auto)	45.0 ± 0.0
AutoSaddler (Iter34) (Auto)	$\mathbf{50.0 \pm 0.0}$

provided by GAIA2 as the base harness. SBP targets enterprise-scale software engineering tasks drawn from popular repositories, for which we use SWE-agent [48] as the base harness. Finally, TB2 comprises 89 realistic tasks across domains including system administration, machine learning, and cybersecurity; for TB2, we adopt Terminus 2 as the base harness.

Implementation and Baseline Methods. We implement the three agents in AutoSaddler (i.e., the *Diagnosis-Patch Agent*, *Reflection Agent*, and *Evolution Agent*) using Claude Agent SDK (CA-SDK) [4]. Detailed agent instructions are provided in Appendix P, and EvoDAG implementation details are given in Appendix B. We compare AutoSaddler against GEPA [1] and Meta-Harness [22]. GEPA is a prompt-centric baseline that iteratively optimizes system prompts. By contrast, Meta-Harness shifts the optimization target from prompts to the agent harness, using a minimal system design and adopting CA-SDK for end-to-end system optimization. Since only TB2 was evaluated by Meta-Harness, we adapt GEPA to GAIA2 and TB2, and Meta-Harness to GAIA2; adaptation details are provided in Appendix B.

Experimental Protocol and Data Splits. Unless otherwise noted, all optimization methods use `Claude Opus 4.6` as the underlying LLM with default endpoint settings. To evaluate generalization, we ensure that the training, development, and test sets contain tasks from distinct task groups. For example, in SBP, the training set consists of tasks from `qutebrowser`; the development set contains tasks from `Vuls` and `NodeBB`; and the test set comprises tasks from `Ansible`, `Flipt`, and `Element-web`. This split enables us to evaluate generalization across repositories. Additional details on the data splits for all three benchmarks are provided in Appendix B. Unless otherwise noted, for each optimization method, we perform a single evolution run on the training and development sets, and conduct *three* repeated runs when evaluating on the test sets.[4] We report *Pass@1* success rates as the evaluation metric.

5.2 Main Results

We report Pass@1 success rates on the test sets for GAIA2 in Table 2, and for SBP and TB2 in Table 3. Two main observations are made. First, **AutoSaddler discovers more effective harnesses**

[4]Because Meta-Harness does not natively support a train–dev split, we provide it with the union of the training and development sets to ensure comparable data exposure.

than the corresponding base harnesses across all three benchmarks. Specifically, AutoSaddler improves over the default agent on GAIA2 by **+9.0 pp** ($53.0\% \rightarrow 62.0\%$), over SWE-agent on SBP by **+8.4 pp** ($37.3\% \rightarrow 46.9\%$), and over Terminus 2 on TB2 by **+10.0 pp** ($40.0\% \rightarrow 50.0\%$). Second, **AutoSaddler outperforms the automated baselines GEPA and Meta-Harness.** Compared with the strongest automated baseline on each benchmark, AutoSaddler achieves gains of **+7.4 pp** on GAIA2 (54.6% vs. 62.0%), **+6.2 pp** on SBP (42.5% vs. 46.9%), and **+4.4 pp** on TB2 (43.3% vs. 50.0%). On TB2, AutoSaddler also surpasses the manually expert-tuned harness, *Terminus KIRA* [20], by **+2.5 pp** (47.5% vs. 50.0%).

We further verify robustness to optimization stochasticity and training-distribution shift. An independent AutoSaddler run on GAIA2 achieves 58.6% Pass@1, remaining well above the base harness and rerun baselines (Appendix C), while optimization on a different training universe achieves 57.4%, a **+5.9 pp** gain over the base harness (Appendix D).

Beyond cross-task-group evaluation, we further analyze **cross-model transferability**. Specifically, we replace the underlying LLM of the GAIA2 task agent, switching from `Opus 4.6` to `Haiku 4.5`, while retaining the harnesses optimized with `Opus 4.6`. As shown in Appendix E, AutoSaddler still improves over the base harness by **+5.6 pp**, suggesting potential transferability across models.

To analyze optimization efficiency, we compare the optimization trajectories of AutoSaddler and the automated baselines on GAIA2 in Figure 1; a visualization of the full search trajectory of AutoSaddler is provided in Appendix J. We find that **AutoSaddler achieves higher performance with substantially fewer task-agent rollouts.** Specifically, AutoSaddler reaches 72.3% development accuracy with only $\sim$1,000 rollouts, whereas GEPA and Meta-Harness saturate at 64.6% and 61.5%, respectively, despite consuming $\sim$2,800 task executions (Figure 1a). The contrast is even sharper when efficiency is measured by the number of rollouts leveraged for learning (Figure 1b): AutoSaddler reaches its best dev-set score after consuming 147 rollouts, $\sim\mathbf{10}\times$ fewer than Meta-Harness ($1{,}400$ rollouts). Similar efficiency gains are observed on TB2, as shown in Appendix H. We further characterize end-to-end optimization cost in Appendix I. Although AutoSaddler incurs moderately higher optimizer-side monetary cost per patch, its selective evaluation strategy reaches 67.7% development accuracy with only 391 task-agent rollouts, already exceeding Meta-Harness's 61.5% peak after $1{,}400$ rollouts.

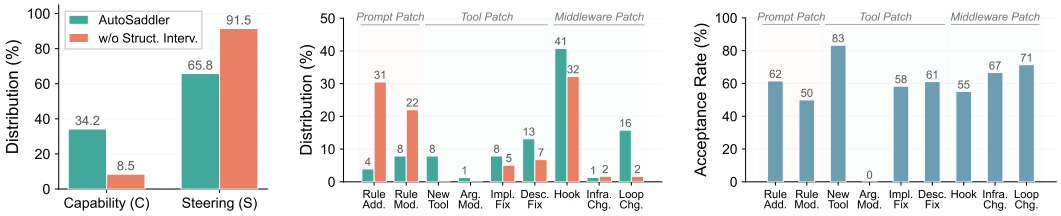
5.3 Ablation Studies and Analysis

We next present ablation studies on GAIA2, focusing on the three key design principles underlying AutoSaddler: *in-depth diagnosis*, *structured intervention*, and *generalization-aware selection*.

RQ1: Does in-depth diagnosis enable superior root-cause identification? We first conduct an ablation study to assess whether in-depth diagnosis improves root-cause identification and ultimately contributes to overall performance gains. Specifically, we evaluate a "*w/o in-depth diagnosis*" variant that replaces CA-SDK-based diagnosis with a shallow diagnostic baseline: a single LLM call receives the execution trace and evaluation results, and infers the failure reason, a strategy commonly used in automatic prompt optimization pipelines for failure reflection [36, 1]. The inferred failure reason is then passed back to CA-SDK for patch generation. In contrast, AutoSaddler's in-depth diagnosis actively explores both execution traces and source code to investigate failures.

As shown in Table 2, removing in-depth diagnosis substantially degrades test-set performance on GAIA2, **reducing Pass@1 from 62.0 to 57.8**. The benefit of in-depth diagnosis is further supported by the detailed file-access and tool-call analysis in Appendix K: compared with the patch-only session, the combined diagnosis–patch session invokes, on average, 6.2 additional tool calls and 5.8 additional file accesses per optimization step. We also quantitatively compare the cumulative number of accepted patches during training under the two settings, where an accepted patch is defined as one that improves performance on the same mini-batch; see Figure 10 in Appendix K. In-depth diagnosis consistently yields more accepted patches. By the end of Epoch 1 (Iteration 25), the gap is substantial (**13** vs. **5**), and this advantage is maintained throughout Epoch 2 (Iterations 26–50). Finally, we provide several qualitative case studies in Appendix K that illustrate how shallow diagnosis and AutoSaddler's in-depth diagnosis differ in identifying root causes.

RQ2: How does structured intervention shape patch diversity and effectiveness? To evaluate the effectiveness of structured intervention, we consider an ablated setting, *w/o Structured Inter-*

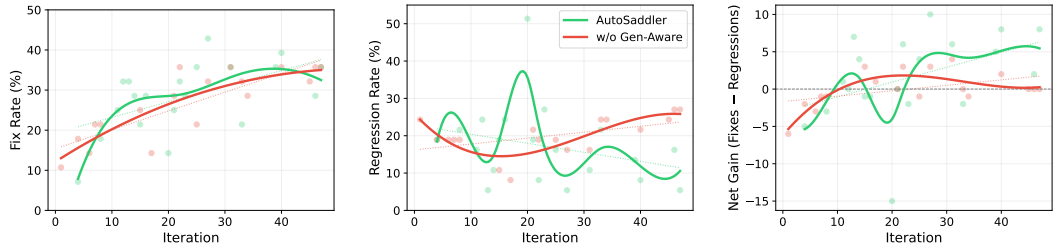


(a) Capability vs. Steering. (b) Patch subtype distribution. (c) Acceptance rate by subtype.

Figure 3: **Patch type distribution and acceptance: AutoSaddler vs. *w/o Structured Intervention*.** Without structural intervention, patches collapse onto Steering (91.5%), while AutoSaddler produces a balanced mix spanning Prompt, Tool, and Middleware edits.

vention, in which we remove the proposed patch taxonomy and phased patch scheduling. This ablation leaves the agent to perform unconstrained edits without explicit search-space boundaries (as in Meta-Harness). As shown in Table 2, removing structured patching substantially reduces Pass@1 on the GAIA2 test set, **from 62.0% to 56.9%**. This result suggests that framing harness optimization as a targeted structural search is critical for achieving meaningful performance gains. Fine-grained ablations in Appendix F further show that removing only Phased Patch Scheduling reduces Pass@1 from 60.7% to 54.8%, while removing the full structured intervention further reduces it to 53.3%.

Further analysis shows that the performance degradation in the *w/o Structured Intervention* setting stems from biased patch exploration and reduced patch diversity, despite allowing unconstrained patch edits. As shown in Figure 3a and Figure 3b, without structured intervention, the system becomes heavily concentrated on Steering patches (91.5%), which largely correspond to straightforward textual edits, while rarely exploring higher-value interventions such as infrastructure or tool improvements. The varying effectiveness of different patch types is further supported by the acceptance-rate breakdown across patch subtypes in Figure 3c. Capability-centric, non-prompt-layer patches, i.e., *New Tool* (83%), *Loop Change* (71%), and *Infra Change* (67%), achieve the highest acceptance rates. However, under *w/o Structured Intervention*, these critical patch types account for only **4%** of generated patches; by contrast, AutoSaddler increases their share to over **25%** through structured patching. Representative patches from these categories discovered by AutoSaddler during evolution are provided in Appendix L. Together, these results indicate that structured intervention encourages the agent to move beyond prompt-layer edits and explore capability-level improvements. Moreover, Appendix G shows that Capability Patches achieve a comparable fix rate to Steering Patches (55% vs. 58%) while inducing substantially fewer regressions (8% vs. 17%), suggesting more durable updates.



(a) Fix Rate (Base-Failed). (b) Regression Rate (Base-Passed). (c) Dev-Set Net Gain.

Figure 4: **Performance comparison on the dev-set across iterations.** AutoSaddler maintains a lower regression rate while achieving comparable fix rates to the *w/o Generalization-Aware Selection* ablation, yielding a consistently positive net gain that the ablation fails to sustain.

RQ3: How does generalization-aware selection prevent overfitting? Finally, we conduct an ablation study to examine how generalization-aware optimization mitigates overfitting to the training set and reduces regressions on unseen scenarios. Specifically, the "*w/o Generalization-Aware Selection*" ablation removes both the reflection session and dev-set evaluation, forcing optimization to rely solely on training-set execution outcomes. As shown in Table 2, removing this component

substantially degrades GAIA2 test-set performance, **reducing Pass@1 from 62.0% to 50.6%**. This is the largest performance drop among all component ablations. Fine-grained ablations in Appendix F further show that removing dev-set filtering reduces Pass@1 from 60.7% to 50.0%, while additionally removing Reflection with EvoDAG further reduces it to 44.9%.

To quantify how this mechanism affects dev-set behavior, Figure 4 tracks both settings on the dev-set and decomposes net performance into the *fix rate*, defined as the success rate on scenarios failed by the initial base harness, and the *regression rate*, defined as the failure rate on scenarios passed by the base harness. Notably, both settings achieve similar fix rates (panel a), indicating that the performance gap is not primarily driven by differences in problem-solving capability. Instead, the key divergence lies in the regression rate (panel b). Although both curves fluctuate during optimization, AutoSaddler exhibits an overall decreasing regression trend (**-0.24 pp/iter**), whereas the ablation shows an increasing trend (**+0.16 pp/iter**).

We further examine the regression spike observed for the ablation at Iteration 21 (panel b) ($8\% \rightarrow 22\%$). At Iteration 20, the ablation introduces a new tool, `send_progress_message_to_user`, and modifies the hook for the widely used `send_message_to_user` tool to forcibly redirect the agent to this new tool. Without reflection to assess collateral damage, this overly broad patch is retained, disrupting agent behavior across many unrelated development scenarios. Interestingly, the same failure pattern—an overly broad hook on a high-frequency tool that acts beyond its intended scope—also appears in AutoSaddler at Iteration 4 (see Figure 12b and the extended discussion in Appendix M). However, because of the reflection mechanism in AutoSaddler, such regression-inducing patches are blocked. This direct contrast demonstrates that generalization-aware selection helps sustain performance gains on unseen scenarios by identifying and filtering over-scoped patches.

Lastly, Figure 4c shows the resulting net gain, indicating that AutoSaddler consistently outpaces the ablation after Iteration 20. This suggests that AutoSaddler can recover from early-iteration regression challenges and achieve steadier improvements in later iterations by abstracting generalizable principles from specific optimization lessons. Additional discussion of principle generalization and qualitative case studies is provided in Appendix M.

6 Conclusion

In this work, we introduced AutoSaddler, an automatic harness optimization framework that formulates harness improvement for LLM agents as an offline learning problem over execution traces. AutoSaddler combines in-depth failure diagnosis, structured patch generation across prompts, tools, and middleware, and generalization-aware update selection via validation and EvoDAG-based evolution. As a result, it produces durable harness updates rather than trajectory-specific fixes. On GAIA2, SWE-Bench Pro, and Terminal-Bench 2.0, AutoSaddler consistently improves over the corresponding base harnesses by **9.0, 9.6, and 10.0** percentage points, respectively, and outperforms the strongest automated baseline on each benchmark by **7.4, 4.4, and 6.7** points. These results show that effective harness optimization requires deep debugging, targeted interventions, and explicit selection for generalization, positioning automatic harness optimization as a promising direction for building more performant and reliable long-horizon agent systems.

7 Acknowledgments

We would like to express our sincere appreciation to the Microsoft Data, Knowledge, and Intelligence (DKI) team for their valuable support throughout this research. In particular, we are grateful to Bo Qiao for his generous assistance in setting up the computing servers used for our experiments. We also thank the creators and maintainers of GAIA2, SWE-Bench Pro, Terminal-Bench 2.0, SWE-agent, Terminus 2, GEPA, and Meta-Harness for making their benchmarks, systems, and tools publicly available, which enabled the experiments in this work. Wook-Shin Han was partly supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2024-00509258, Global AI Frontier Lab, 40%; IITP-2026-RS-2026-25616370, AI Star Fellowship Support Program, 20%) and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS. 2025-00517736, 40%).

References

[1] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*, 2026.

[2] Huan ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Qihan Ren, Yiran Wu, Hongru WANG, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Cheng Qian, Zhenhailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *Transactions on Machine Learning Research*, 2026. Survey Certification.

[3] Anthropic. Effective harnesses for long-running agents. `https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents`, 2025. Accessed 2026-04-21.

[4] Anthropic. Agent sdk overview - claude code docs. `https://code.claude.com/docs/en/agent-sdk/overview`, 2026. Accessed 2026-04-21.

[5] Anthropic. Harness design for long-running application development. `https://www.anthropic.com/engineering/harness-design-long-running-apps`, 2026. Accessed 2026-04-21.

[6] Zhicheng Cai, Xinyuan Guo, Yu Pei, Jiangtao Feng, Jiangjie Chen, Ya-Qin Zhang, Wei-Ying Ma, Mingxuan Wang, and Hao Zhou. FLEX: continuous agent evolution via forward learning from experience. *CoRR*, abs/2511.06449, 2025.

[7] Mengzhuo Chen, Junjie Wang, Zhe Liu, Yawen Wang, Haiming Zheng, and Qing Wang. From failed trajectories to reliable llm agents: Diagnosing and repairing harness flaws. *arXiv preprint arXiv:2606.06324*, 2026.

[8] Zhengyu Chen, Teng Xiao, Huaisheng Zhu, Yige Yuan, Luan Zhang, and Jingang Wang. Co-harness: Co-evolving harnesses and model weights for llm agents. *arXiv preprint arXiv:2607.22688*, 2026.

[9] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can AI agents solve long-horizon software engineering tasks? *CoRR*, abs/2509.16941, 2025.

[10] Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, Kunal Malkan, Dheeraj Mekala, Pierre Ménard, Gerard Moreno-Torres Bertran, Ulyana Piterbarg, Mikhail Plekhanov, Mathieu Rita, Andrey Rusakov, Vladislav Vorotilov, Mengjue Wang, Ian Yu, Amine Benhalloum, Grégoire Mialon, and Thomas Scialom. Gaia2: Benchmarking LLM agents on dynamic and asynchronous environments. *CoRR*, abs/2602.11964, 2026.

[11] Jia Fu, Xiaoting Qin, Fangkai Yang, Lu Wang, Jue Zhang, Qingwei Lin, Yubo Chen, Dongmei Zhang, Saravan Rajmohan, and Qi Zhang. Autorag-hp: Automatic online hyper-parameter tuning for retrieval-augmented generation. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*, Findings of ACL, pages 3875–3891. Association for Computational Linguistics, 2024.

[12] Yu Ge, Linna Xie, Zhong Li, Yu Pei, and Tian Zhang. Who is introducing the failure? automatically attributing failures of multi-agent systems via spectrum analysis. *arXiv preprint arXiv:2509.13782*, 2025.

[13] Hanghui Guo, Weijie Shi, Zhangze Chen, Shengxiang Xu, Yishu Wang, Yimei Zhang, Wangze Ni, Jia Zhu, and Shimin Di. Drevo: Distilling recalibrated historical experience for harness self-evolution. *arXiv preprint arXiv:2607.26722*, 2026.

[14] Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[15] Prannay Hebbar, Yogendra Manawat, Samuel Verboomen, Alesia Ivanova, Selvam Palanimalai, Kunal Bhatia, and Vignesh Baskaran. Sia: Self improving ai with harness & weight updates. *arXiv preprint arXiv:2605.27276*, 2026.

[16] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025.

[17] 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.

[18] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2024.

[19] Shashank Kirtania, Param Biyani, Priyanshu Gupta, Yasharth Bajpai, Roshni Iyer, Sumit Gulwani, and Gustavo Soares. Improving language agents through BREW. *CoRR*, abs/2511.20297, 2025.

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

[21] LangChain. Improving deep agents with harness engineering. `https://www.langchain.com/blog/improving-deep-agents-with-harness-engineering`, 2026. Accessed 2026-04-21.

[22] 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*, 2026.

[23] 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.

[24] Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z Pan, et al. Evox: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026.

[25] Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P Murphy. Autoharness: improving llm agents by automatically synthesizing a code harness. *arXiv preprint arXiv:2603.03329*, 2026.

[26] Guoqing Ma, Jia Zhu, Hanghui Guo, Weijie Shi, Jiawei Shen, Jingjiang Liu, and Yidan Liang. Automatic failure attribution and critical step prediction method for multi-agent systems based on causal inference. *arXiv preprint arXiv:2509.08682*, 2025.

[27] Ming Ma, Jue Zhang, Fangkai Yang, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Dover: Intervention-driven auto debugging for LLM multi-agent systems. In *The Fourteenth International Conference on Learning Representations*, 2026.

[28] Mike A. Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, Estefany Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis-Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin 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, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *CoRR*, abs/2601.11868, 2026.

[29] Nous Research. Hermes agent — the agent that grows with you. `https://hermes-agent.nousresearch.com/`, 2026. Accessed 2026-04-21.

[30] Alexander Novikov, Ngân Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery. *CoRR*, abs/2506.13131, 2025.

[31] OpenAI. Harness engineering: leveraging codex in an agent-first world. `https://openai.com/index/harness-engineering/`, 2026. Accessed 2026-04-21.

[32] Krista Opsahl-Ong, Michael J. Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 9340–9366. Association for Computational Linguistics, 2024.

[33] Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *The Fourteenth International Conference on Learning Representations*, 2026.

[34] Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Hai-Tao Zheng. Natural-language agent harnesses. *arXiv preprint arXiv:2603.25723*, 2026.

[35] 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.

[36] Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 7957–7968. Association for Computational Linguistics, 2023.

[37] Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, Xing Zhou, Dongrui Liu, Ling Yang, Yue Wu, Kaixuan Huang, Shilong Liu, Hongru Wang, and Mengdi Wang. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *CoRR*, abs/2505.20286, 2025.

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

[39] Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent. `https://github.com/algorithmicsuperintelligence/openevolve`, 2025. Accessed 2026-04-21.

[40] Varun Ursekar, Apaar Shanker, Veronica Chatrath, Sam Denton, et al. Vero: An evaluation harness for agents to optimize agents. *arXiv preprint arXiv:2602.22480*, 2026.

[41] Chi Wang, Xueqing Liu, and Ahmed Hassan Awadallah. Cost-effective hyperparameter optimization for large language model generation inference. In Aleksandra Faust, Roman Garnett, Colin White, Frank Hutter, and Jacob R. Gardner, editors, *International Conference on Automated Machine Learning, 12-15 November 2023, Hasso Plattner Institute, Potsdam, Germany*, Proceedings of Machine Learning Research, pages 21/1–17. PMLR, 2023.

[42] Wenyi Wang, Piotr Piękos, Li Nanbo, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley-g\"odel machine: Human-level coding agent development by an approximation of the optimal self-improving machine. In *The Fourteenth International Conference on Learning Representations*, 2026.

[43] Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric P. Xing, and Zhiting Hu. Promptagent: Strategic planning with language models enables expert-level prompt optimization. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[44] Yingxu Wang, Siwei Liu, Jinyuan Fang, and Zaiqiao Meng. Evoagentx: An automated framework for evolving agentic workflows. In Ivan Habernal, Peter Schulam, and Jörg Tiedemann, editors, *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, EMNLP 2025 - System Demonstrations, Suzhou, China, November 4-9, 2025*, pages 643–655. Association for Computational Linguistics, 2025.

[45] Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-swe-agent: Can software engineering agents self-evolve on the fly? *CoRR*, abs/2511.13646, 2025.

[46] Minghao Yan, Bo Peng, Benjamin Coleman, Ziqi Chen, Zhouhang Xie, Shuo Chen, Zhankui He, Noveen Sachdeva, Isabella Ye, Weili Wang, et al. Pacevolve: Enabling long-horizon progress-aware consistent evolution. *arXiv preprint arXiv:2601.10657*, 2026.

[47] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[48] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024.

[49] Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursively self-improvement. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 27890–27913. Association for Computational Linguistics, 2025.

[50] Mert Yüksekgönül, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text. *Nature*, 639, 2025.

[51] Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. Agentracer: Who is inducing failure in the LLM agentic systems? *CoRR*, abs/2509.03312, 2025.

[52] 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*, 2026.

[53] Heng Zhang, Yuling Shi, Xiaodong Gu, Haochen You, Zijian Zhang, Lubin Gan, Yilei Yuan, and Jin Huang. Graphtracer: Graph-guided failure tracing in llm agents for robust multi-turn deep search. *arXiv preprint arXiv:2510.10581*, 2025.

[54] Jenny Zhang, Shengran Hu, Cong Lu, Robert Tjarko Lange, and Jeff Clune. Darwin gödel machine: Open-ended evolution of self-improving agents. In *The Fourteenth International Conference on Learning Representations*, 2026.

[55] Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob N. Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents. *CoRR*, abs/2603.19461, 2026.

[56] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. Aflow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025.

[57] Luan Zhang, Ruochen Zhou, Dandan Song, Zhengyu Chen, Yuhang Tian, Jun Yang, Huipeng Ma, Chenhao Li, Guangyuan Feng, Xudong Li, et al. Harnesscompass: Guiding automatic harness evolution toward generalizable and effective agent harnesses. *arXiv preprint arXiv:2608.01918*, 2026.

[58] Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? on automated failure attribution of LLM multi-agent systems. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu, editors, *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, Proceedings of Machine Learning Research. PMLR / OpenReview.net, 2025.

[59] Shaokun Zhang, Jieyu Zhang, Jiale Liu, Linxin Song, Chi Wang, Ranjay Krishna, and Qingyun Wu. Offline training of language model agents with functions as learnable weights. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pages 60315–60335. PMLR / OpenReview.net, 2024.

[60] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: LLM agents are experiential learners. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan, editors, *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pages 19632–19642. AAAI Press, 2024.

[61] 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.

[62] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[63] Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, et al. Where llm agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*, 2025.

[64] Runchuan Zhu, Bowen Jiang, Lingrui Mei, Fangkai Yang, Lu Wang, Haoxiang Gao, Fengshuo Bai, Pu Zhao, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Adaptflow: Adaptive workflow optimization via meta-learning. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng, editors, *Findings of the Association for Computational Linguistics: EMNLP 2025, Suzhou, China, November 4-9, 2025*, pages 3287–3302. Association for Computational Linguistics, 2025.

Traditional Mini-batch Training	AutoSaddler Stage	Interpretation
Step 1: Sample mini-batch	❶ Sample mini-batch for evaluating current harness	Same role
Step 2: Run forward pass	❶ Execute the current harness	The agent rollout is the forward computation. It produces both a final answer and a trajectory.
Step 3: Compute loss	❶ Record outcomes and traces	The task outcome provides the sparse metric signal; the trace provides evidence needed to explain the outcome.
Step 4: Backpropagate gradients	❷ Diagnose and patch + ❸ verify on the same mini-batch	AutoSaddler has no numerical gradient; it constructs a textual update through hypothesis generation on root cause, intervention, and verification.
Step 6: Validate/checkpoint	❹ Evaluate on the dev-set	Dev-set evaluation checks whether the candidate improvement generalizes beyond the mini-batch.
Step 5: Apply optimizer update	❺ Reflect + ❻ store in EvoDAG + ❼ evolve the next harness	The committed update is not merely the raw patch. AutoSaddler updates optimizer state and uses it to synthesize the next candidate harness.
Step 7: Repeat	Next iteration	The evolved harness is evaluated on a fresh mini-batch.

Table 4: Mapping between conventional mini-batch training and the AutoSaddler loop in Figure 2. AutoSaddler follows the outer structure of mini-batch learning but replaces differentiable gradient computation with trace-grounded symbolic optimization.

A AutoSaddler as Mini-Batch Learning over Textual Harness Parameters

This appendix clarifies how the AutoSaddler loop in Figure 2 relates to the standard mini-batch training loop. The analogy is useful because AutoSaddler preserves the outer structure of mini-batch learning: it repeatedly evaluates a candidate system on a mini-batch, extracts an update signal, checks whether the update generalizes, and uses accumulated optimization history to propose the next candidate. However, the analogy should not be taken literally. Unlike neural parameters, harness parameters are textual, executable, and non-differentiable. AutoSaddler therefore replaces differentiable gradient computation with a trace-grounded symbolic optimization procedure.

Mapping to traditional mini-batch training. Table 4 summarizes the correspondence. We denote the stages of traditional mini-batch training as Step 1–Step 7, and refer to AutoSaddler stages using the circled labels from Figure 2.

Why backpropagation becomes diagnosis, patching, and verification. The central mismatch between the two settings lies in the analogue of backpropagation. In neural training, once the loss is computed, backpropagation provides a mathematically defined gradient. The update direction is therefore relatively well specified.

In AutoSaddler, a failed rollout does not directly specify how the harness should change. The root cause may lie in the prompt, a tool interface, a tool implementation, a middleware hook, or the agent loop. The system must therefore infer a root-cause hypothesis from the trace, implement a targeted intervention, and empirically test whether the intervention improves behavior on the same mini-batch:

$$\text{diagnosis} \longrightarrow \text{patch-as-intervention} \longrightarrow \text{same-batch verification.} \tag{2}$$

This sequence plays the role of constructing a credible textual gradient. Importantly, the patch at this stage is not yet best understood as the final committed parameter update. Rather, it is an intervention used to test whether the inferred root cause is plausible. This distinction is important because numerical gradients are derived from a fixed mathematical operator, whereas textual gradients are inferred, semantic, and fallible. Same-batch verification is therefore necessary to avoid treating an untested explanation as a reliable update direction.

Algorithm 1 AutoSaddler as mini-batch learning over harness parameters

Require: Initial harness H_0, training set D_{train}, development set D_{dev}, rollout budget K

Initialize EvoDAG G_0 with H_0
for iteration $n = 0, 1, 2, \ldots$ until budget K is exhausted **do**
  Sample mini-batch $B_n \subset D_{\text{train}}$ ▷ Figure 2: ❶; Step 1
  Execute H_n on B_n, collecting outcomes and traces τ_n ▷ ❶; Steps 2–3
  Diagnose failures in τ_n and infer root-cause hypotheses ▷ ❷; textual credit assignment
  Generate targeted patch $\Delta\theta_n$, yielding candidate $H'_n = H_n + \Delta\theta_n$ ▷ ❷; patch as intervention
  Re-evaluate H'_n on B_n ▷ ❸; verify textual gradient
  if same-batch performance improves **then**
    Evaluate H'_n on D_{dev} ▷ ❹; Step 6 generalization gate
  end if
  Reflect on before/after traces and evaluation results ▷ ❺; convert evidence into lessons
  Update EvoDAG G_n with patch, metrics, and lessons to obtain G_{n+1} ▷ ❻; symbolic optimizer state
  Evolve next harness H_{n+1} from G_{n+1} ▷ ❼; Step 5 committed update
end for
return best harness selected by development performance

Why the optimizer update is reflection plus EvoDAG evolution. A second mismatch concerns the notion of an optimizer update. In ordinary training, the optimizer directly modifies numeric parameters. In AutoSaddler, the raw patch is only one candidate change. Before allowing this change to influence future search, the system must determine whether it is useful beyond the current mini-batch and what reusable lesson, if any, should be retained.

This is why the committed update is better understood as

$$\text{dev-set validation} \longrightarrow \text{reflection} \longrightarrow \text{EvoDAG update} \longrightarrow \text{evolution of the next harness.} \quad (3)$$

Reflection distills before/after evidence into reusable lessons, and EvoDAG stores these lessons together with patch descriptions and performance signals. The Evolution Session then uses this symbolic optimizer state to select, revise, or recombine components from previously explored harnesses when constructing the next candidate. In this sense, EvoDAG serves as a form of optimizer memory. Analogous to how momentum accumulates past gradient information to smooth noisy updates and steer future steps toward historically productive directions, EvoDAG accumulates past patches, outcomes, and reflections to guide harness evolution toward changes with evidence of generalizable benefit. Thus, AutoSaddler's update is history-aware and compositional, rather than a single local arithmetic step.

A note on ordering. The mapping above intentionally swaps the usual order of Step 5 and Step 6. In standard mini-batch training, one typically applies the optimizer update before validation or checkpointing. In AutoSaddler, development-set validation precedes the final symbolic optimizer update because EvoDAG is part of the optimizer state. Once a misleading lesson is stored, it can influence future evolution. Validation therefore serves as a generalization gate before the system commits lessons to EvoDAG and evolves H_{n+1}.

Algorithmic view. Algorithm 1 presents the same view algorithmically. The key point is that the patch produced in ❷–❸ functions as an intervention for testing an inferred textual gradient. The final update to the search trajectory occurs only after validation, reflection, and EvoDAG-based evolution.

Phased patch scheduling. We divide the patch types in Table 1 into two categories based on the type of harness change they make. **Capability patches** modify executable code or orchestration logic, including tool implementations, tool arguments, infrastructure settings, and agent-loop logic. These patches can change what actions the agent is able to perform or how the harness executes those actions. **Steering patches** are textual edits that leave the underlying executable code unchanged, including modifications to prompts, tool descriptions, and hook reminder texts. These patches primarily refine how the agent selects among existing capabilities and follows task-specific constraints.

This distinction is analogous to, but not identical with, large versus small learning-rate steps in gradient-based optimization. Capability patches often behave like larger steps because they can introduce new functionality, change control flow, or alter the available action space. Steering patches often behave like smaller steps because they adjust the agent's behavior within an already

established capability set. However, this analogy is only approximate: prompt or hook edits can sometimes produce large behavioral changes, especially when they affect high-frequency decisions. We therefore use the capability–steering distinction as a scheduling heuristic rather than a strict guarantee about effect size, and rely on verification and development-set validation to detect overly broad or regression-inducing changes.

Motivated by this analogy, AutoSaddler adopts a two-phase schedule with transition point k:

1. **Exploration phase** ($n \leq k$). The search prioritizes capability patches to address fundamental gaps in tooling, infrastructure, and agent-loop behavior.
2. **Refinement phase** ($n > k$). The search switches to steering patches to refine behavior after the capability set has stabilized.

The transition point k can be specified directly as an iteration count or implicitly through the number of training epochs:

$$k = E \cdot \left\lceil \frac{|\mathcal{D}_{\text{train}}|}{B} \right\rceil,$$

where E is the number of capability-phase epochs and B is the mini-batch size. As in learning-rate scheduling, setting k too small may under-explore high-impact capability improvements, while setting it too large may delay lower-risk behavioral refinement. In our experiments, a single capability-phase epoch ($E = 1$) works well, allowing the search to traverse the training set once with high-impact patches before switching to targeted steering adjustments.

Takeaway. AutoSaddler can be viewed as mini-batch offline learning over externalized agent parameters. Its design differs from standard gradient training because harness parameters are textual and non-differentiable; the performance signal is sparse and delayed; and credit assignment across long-horizon traces is ambiguous. The additional machinery is therefore not incidental. Trace collection, diagnosis, patch-as-intervention, same-batch verification, development-set validation, reflection, EvoDAG memory, and evolution are the symbolic counterparts of the mechanisms that are compactly handled by loss functions, backpropagation, optimizer state, and checkpointing in traditional mini-batch training.

B Additional Details on Experiment Setup

Implementation of EvoDAG. The DAG underlying EvoDAG can be implemented straightforwardly, but providing effective agent access to the DAG is more challenging: as candidates and traces accumulate across iterations, navigating EvoDAG directly becomes impractical. Rather than serializing the full DAG into the prompt, we introduce the `evo-dag` command-line interface (CLI) to facilitate interaction between the CA-SDK and EvoDAG. The interface provides on-demand access to the DAG's structured summaries, including patch history, lessons learned, the scenario registry, and harness code diffs. This allows the agent to identify relevant context before drilling down into specific raw traces or source files for detailed analysis. The full command set is summarized in Table 5.

Adapting Baselines to Our Benchmarks. GEPA treats the system prompt as a single unified string. By contrast, the default ReAct-based agent in GAIA2 distributes its system prompt across multiple sections, such as core behavioral principles, the ReAct JSON tool-calling format, and Meta's Agents Research Environments (ARE) simulation-environment instructions. These sections are defined as separate variables and concatenated only at inference time, creating a structural mismatch that prevents the direct application of GEPA. To address this mismatch, we concatenate the sections into a single string, using `===== VAR_NAME =====` delimiters to mark section boundaries, and instruct GEPA's reflection LLM to preserve these delimiters during prompt evolution. The evolved prompt is then split along the delimiters, and the resulting sections are passed to CA-SDK to update the corresponding variables. For SBP and TB2, we adopt SWE-agent and Terminus 2 as the base harnesses, respectively; since both agents define their system prompts as a single unified string, GEPA can be applied directly without further modification.

The original Meta-Harness assumes that the harness is contained within a single Python script and requires inheritance from a fixed base agent class. In the GAIA2 default agent, however, the harness may be distributed across an entire repository. We therefore extend the Meta-Harness adaptation

Table 5: Commands exposed by the `evo-dag` CLI, grouped into read operations for scoped EvoDAG views and write operations for per-session node or scenario updates.

Command	Purpose	When to use
Read operations		
`evo-dag summary`	DAG topology, best candidate, all edges	Start of session; quick orientation
`evo-dag show history`	Full patch history: diffs, reflections, lessons	Deep dive into prior attempts
`evo-dag show node <idx>`	Node details: scores, intent, verdict, output dirs	Inspecting a specific candidate
`evo-dag show edge <parent> <child>`	Code diff, per-scenario impacts, files changed	Understanding a specific patch
`evo-dag show scenario <id>`	Per-scenario history, root causes, attempted fixes	Before diagnosing a failing scenario
`evo-dag show current-batch`	Current mini-batch scenario IDs and output dirs	Orienting to the current iteration
`evo-dag show lineage`	DAG lineage with edge types	Understanding branching structure
`evo-dag show lessons`	Accumulated good/bad patterns	Checking known patterns before patching
Write operations		
`evo-dag update-selection`	Record selected parent candidate(s) and the reasoning behind the choice	Evolution Session, after parent selection
`evo-dag update-intent`	Record target scenarios, diagnosis, approach, files changed, and change summary	Diagnosis-Patch Session, after applying patches
`evo-dag update-reflection`	Record per-scenario status (`fixed` / `regressed` / `still_failing` / `still_passing`), root cause, post-patch explanation, prevention or next step, and generalization note	Reflection Session, per scenario

to the repository level by instructing CA-SDK to apply the same patching procedure used in the original Meta-Harness across the full repository. This enables holistic harness optimization beyond the single-file setting.

Benchmark-Specific Evaluation Protocol. For each benchmark, we follow its official evaluation infrastructure to ensure that the reported results are directly comparable to prior leaderboard numbers.

GAIA2. We run the default ReAct-based agent through ARE using the `are-run` command, which executes each task in its corresponding sandboxed `Universe` and records the full trajectory. Following the GAIA2 protocol [10], we use `Llama-3.3-70B-Instruct` as the judge model to determine task success from the final agent state.

SWE-Bench Pro. Following the official protocol of [9], each agent-generated patch is applied inside a per-instance Docker image preconfigured with the repository at the issue commit. The patch is then verified using the instance's *fail-to-pass* and *pass-to-pass* unit tests. A task is counted as resolved only if all fail-to-pass tests pass and no pass-to-pass test regresses.

Terminal-Bench 2.0. Following [28], we use `Harbor` to run each task in an isolated Docker container. Success is determined by a task-specific test script that inspects the final terminal and system state.

Data Splits across Benchmarks. To evaluate whether optimized harnesses generalize beyond the tasks seen during optimization, we construct train, development, and test splits separately for each benchmark, as summarized in Table 6. For benchmarks with a natural grouping structure,

we split by task group rather than by individual tasks. In GAIA2, the split axis is the `Universe`, where each `Universe` corresponds to a distinct persona and simulated digital environment. This design ensures that the test set contains tasks from groups unseen during optimization, providing a stronger measure of out-of-distribution generalization than random task-level splits. In SWE-Bench Pro, the split axis is the repository, which also induces shifts in programming language, codebase structure, and issue distribution. This design ensures that the test set contains tasks from groups unseen during optimization, providing a stronger measure of out-of-distribution generalization than random task-level splits. For Terminal-Bench 2.0, tasks span diverse domains but do not provide a natural grouping axis; we therefore use a uniform random partition into train, dev, and test sets.

Table 6: Data splits across the three benchmarks. To evaluate generalization under distribution shift, we partition each benchmark such that train, development, and test sets contain tasks from *disjoint task groups*— `Universes` (personas) for GAIA2 and repositories (programming languages) for SWE-Bench Pro—rather than random task-level splits. Terminal-Bench 2.0 contains only 89 tasks across diverse domains and offers no natural grouping axis, so we adopt a uniform random partition.

Benchmark	Split Axis	Split	Task Group	# Tasks
GAIA2	`Universe` (persona)	Train	`Universe 29`	75
		Dev	`Universe 30`	65
		Test	`Universe 21`	107
			`Universe 22`	112
			`Universe 27`	81
SWE-Bench Pro	Repository (language)	Train	`qutebrowser` (Python)	79
		Dev	`Vuls` (Go)	40
			`NodeBB` (JavaScript)	40
		Test	`Ansible` (Python)	96
			`Flipt` (Go)	85
			`Element-web` (TypeScript)	56
Terminal-Bench 2.0	Random[†]	Train	—	30
		Dev	—	19
		Test	—	40

[†] Only 89 tasks across heterogeneous domains (system administration, machine learning, cybersecurity); no natural axis for distribution-shifted splits exists.

Optimization Budgets. For AutoSaddler and GEPA, we optimize for 2 epochs on GAIA2 and SWE-Bench-Pro, and for 4 epochs on Terminal-Bench 2.0; the doubled budget on TB2 compensates for its training set being roughly half the size of GAIA2. For Meta-Harness, applying the same epoch-based budget would be misleading: unlike AutoSaddler and GEPA, which perform mini-batch optimization, Meta-Harness operates in a full-batch regime, so a 2-epoch budget corresponds to only two patch updates and would unfairly handicap the baseline. To ensure a fair comparison, we instead match optimization budgets in terms of total task executions consumed during training. Concretely, we train Meta-Harness for 20 epochs on GAIA2 and 15 epochs on Terminal-Bench 2.0, and 8 epochs on SWE-Bench-Pro, ensuring its total task executions are no smaller than those used by AutoSaddler on every benchmark.

C Robustness to Optimization Stochasticity

Our main experiments perform one optimization run for each method because end-to-end harness optimization is substantially more expensive than repeated test-time evaluation. To assess whether the observed gains are sensitive to stochastic optimization trajectories, we conduct an additional independent optimization run for AutoSaddler, GEPA, and Meta-Harness on GAIA2. We evaluate each resulting harness on Universe 22, the largest held-out test universe with 112 scenarios, using three repeated executions per harness.

Table 7: **Robustness to optimization stochasticity**. Pass@1 on GAIA2 Universe 22 for harnesses obtained from two independent optimization runs. Test performance is reported as mean ± standard deviation over three executions.

Method	GAIA2 Universe 22 (112 scenarios; Pass@1)
Default Agent (Manual)	51.5 ± 4.9
GEPA (Run 1)	47.9 ± 3.4
GEPA (Run 2)	50.6 ± 0.5
Meta-Harness (Run 1)	51.5 ± 5.2
Meta-Harness (Run 2)	51.2 ± 2.2
AutoSaddler (Run 1)	$\mathbf{60.7 \pm 2.4}$
AutoSaddler (Run 2)	58.6 ± 0.5

Table 7 reports the results. The second AutoSaddler optimization run achieves 58.6% Pass@1, only 2.1 percentage points below the first run and 7.1 points above the default agent. It also outperforms the independently rerun GEPA and Meta-Harness harnesses by 8.0 and 7.4 percentage points, respectively. Although two optimization trajectories are insufficient for a comprehensive statistical characterization of the optimizer, the consistent gains across independent runs indicate that the improvement of AutoSaddler is not specific to a single favorable search trajectory.

D Robustness to Training-Distribution Shift

Table 8: **Robustness to training-distribution shift**. Pass@1 on GAIA2 Universe 22 when AutoSaddler is independently optimized using different training universes. Values are mean ± standard deviation over three test executions.

Method	GAIA2 Universe 22 (112 scenarios; Pass@1)
Default Agent (Manual)	51.5 ± 4.9
AutoSaddler (Run 1; Universe 29)	$\mathbf{60.7 \pm 2.4}$
AutoSaddler (Run 2; Universe 29)	58.6 ± 0.5
AutoSaddler (Different Train Set; Universe 24)	57.4 ± 2.1

The main GAIA2 experiments optimize the harness using Universe 29 as the training set. To examine whether the resulting gains depend critically on this particular training distribution, we replace Universe 29 with the comparably sized Universe 24 and independently optimize AutoSaddler while keeping the remaining optimization and evaluation protocol unchanged. The resulting harness is evaluated three times on Universe 22.

As shown in Table 8, the harness optimized on Universe 24 achieves 57.4% Pass@1. Despite being optimized using a different training universe, it improves over the default agent by 5.9 percentage points and remains within 1.2 points of the second independent AutoSaddler run trained on Universe 29. These results suggest that AutoSaddler can discover effective harness updates from different persona and task distributions, rather than relying on properties specific to the original Universe 29 training split.

E Cross-Model Transferability to Weaker Agent Backbone

While Table 2 reports results with `Claude Opus 4.6` serving as both the optimizer and the agent backbone, a key practical question is whether harnesses optimized by a strong model retain their benefits when deployed with a weaker task agent. To test this, we re-evaluate all methods on GAIA2 using `Claude Haiku 4.5` as the task agent backbone while keeping the harnesses unchanged from those produced during Opus-based optimization. Table 9 shows that AutoSaddler achieves an overall improvement of $+5.6$ pp over the default agent, demonstrating effective cross-model transferability. Notably, AutoSaddler consistently outperforms all baseline methods and ablation settings across all evaluated universes.

Table 9: **Cross-model transferability**. Pass@1 success rates on the test-set of GAIA2 using `Claude Haiku 4.5` as the agent backbone, with harnesses optimized by `Claude Opus 4.6`. Values are mean ± standard deviation over three runs.

Harness (Type)	GAIA2 Universe (Pass@1)			Avg.
	21 (107)	22 (112)	27 (81)	
Default Agent (Manual)	31.2 ± 5.9	24.4 ± 1.4	36.2 ± 1.9	30.0 ± 1.8
GEPA (Auto)	33.6 ± 1.9	24.4 ± 0.5	38.3 ± 1.2	31.4 ± 0.4
Meta-Harness (Auto)	30.8 ± 0.0	26.8 ± 4.7	34.2 ± 4.0	30.2 ± 2.7
AutoSaddler (Auto)	$\mathbf{38.3 \pm 0.0}$	$\mathbf{30.4 \pm 1.8}$	$\mathbf{39.1 \pm 2.6}$	$\mathbf{35.6 \pm 0.8}$
w/o In-depth Diagnosis (Auto)	36.1 ± 5.5	19.0 ± 5.2	34.2 ± 0.7	29.2 ± 3.4
w/o Structured Intervention (Auto)	32.1 ± 3.0	28.0 ± 4.0	37.4 ± 5.8	32.0 ± 3.5
w/o Generalization-aware Selection (Auto)	28.3 ± 2.9	23.2 ± 4.1	30.0 ± 3.8	26.9 ± 1.3

F Fine-Grained Ablations of Structured Intervention and Generalization-Aware Selection

Table 10: **Fine-grained ablations**. Pass@1 on GAIA2 Universe 22. Values are mean ± standard deviation over three runs.

Method	GAIA2 Universe 22 (112 scenarios; Pass@1)
Default Agent (Manual)	51.5 ± 4.9
AutoSaddler	$\mathbf{60.7 \pm 2.4}$
w/o Structured Intervention	53.3 ± 7.6
w/o Phase Scheduling	54.8 ± 4.4
w/o Generalization-Aware Selection	44.9 ± 6.7
w/o Dev-Set Filtering	50.0 ± 5.0

The main ablations remove each design principle of AutoSaddler as a whole. We further disentangle the individual mechanisms underlying *structured intervention* and *generalization-aware selection*. For structured intervention, the main *w/o Structured Intervention* ablation removes both the patch taxonomy and Phased Patch Scheduling. We therefore introduce a finer-grained *w/o Phase Scheduling* variant that retains the patch taxonomy while removing only the phased schedule. Similarly, the main *w/o Generalization-Aware Selection* ablation removes both development-set filtering and Reflection with EvoDAG. We introduce a *w/o Dev-Set Filtering* variant that retains Reflection with EvoDAG but removes development-set filtering. For each variant, we rerun the full optimization procedure and evaluate the resulting harness on GAIA2 Universe 22 using three repeated test executions.

Table 10 shows the results. Removing only Phased Patch Scheduling reduces Pass@1 from 60.7% to 54.8%, a drop of 5.9 percentage points. Removing the patch taxonomy in addition further reduces performance to 53.3%, indicating an additional 1.5-point contribution from structuring the patch space. For generalization-aware selection, removing development-set filtering reduces Pass@1 from 60.7% to 50.0%, while additionally removing Reflection with EvoDAG further reduces performance to 44.9%. Thus, development-set filtering accounts for the larger effect in this comparison, while Reflection with EvoDAG provides an additional 5.1-point gain. Together, these finer-grained ablations indicate that both mechanisms within each design principle contribute to final harness performance.

G Patch Durability Analysis

A desirable harness update should not only repair the scenarios that motivate the patch, but should also preserve behavior that was already correct. We therefore analyze patch durability by comparing task outcomes before and after each generated patch on the same training mini-batch. We define the *fix rate* as the fraction of previously failing scenarios that pass after applying the patch, and the *regression rate* as the fraction of previously passing scenarios that fail after the patch.

We first group patches according to the three harness components used by the patch taxonomy: Prompt, Tool, and Middleware. As shown in Table 11, the three categories exhibit similar fix rates of 57–59%, while Tool patches have a moderately higher regression rate of 19%, compared with 14% for Prompt and Middleware patches. Thus, the component being edited alone does not reveal a clear durability pattern.

Table 11: **Patch durability by harness component**. Fix and regression rates are measured by comparing pre- and post-patch outcomes on the same training mini-batch.

Patch Category	Fix Rate (%)	Regression Rate (%)
Prompt	59	14
Tool	57	19
Middleware	57	14

One reason for the weak separation across these categories is that they mix different intervention mechanisms. For example, Tool patches include executable changes such as new tools and implementation fixes, but also textual changes to tool descriptions. Similarly, Middleware patches include both executable infrastructure or agent-loop changes and text-only PreToolUse hooks. We therefore reclassify all generated patches using the higher-level Capability–Steering taxonomy from the main paper. Capability Patches modify executable functionality or orchestration logic, whereas Steering Patches alter textual instructions without changing the underlying executable capability.

Table 12 reveals a clearer difference. Steering Patches achieve a slightly higher fix rate than Capability Patches (58% vs. 55%), but their regression rate is more than twice as high (17% vs. 8%). Capability Patches therefore obtain a comparable rate of local repairs while reducing regressions by 9 percentage points. This result suggests that capability-level interventions tend to produce more durable updates, whereas textual steering is more susceptible to spilling over to scenarios outside the intended scope. This quantitative pattern is also consistent with our qualitative examples: well-scoped executable fixes often address deterministic capability limitations, while overly broad hooks or prompt rules can affect unrelated scenarios.

Table 12: **Patch durability by intervention mechanism**. Capability and Steering Patches exhibit similar fix rates, but Steering Patches induce substantially more regressions.

Patch Type	Fix Rate (%)	Regression Rate (%)
Steering Patch	58	17
Capability Patch	55	8

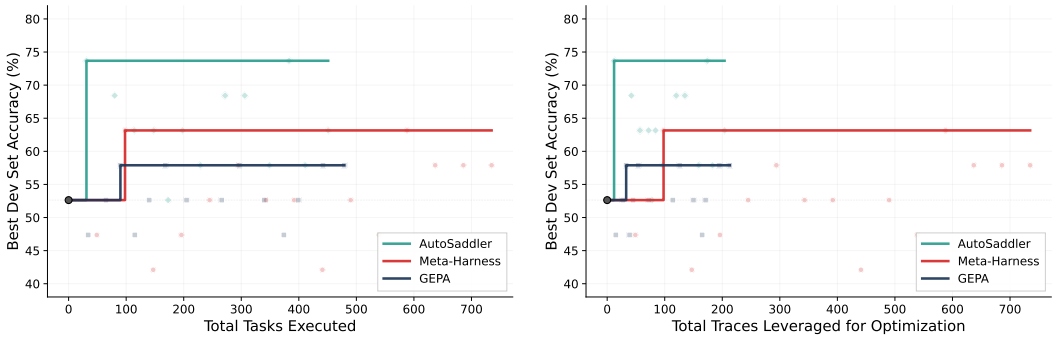
H Optimization Efficiency on Terminal-Bench 2.0

To assess whether AutoSaddler's optimization efficiency generalizes beyond GAIA2, we report compute and learning efficiency on Terminal-Bench 2.0, which targets a distinct agentic capability: long-horizon command-line problem solving. We follow the same protocol as in our main GAIA2 experiments (Section 5), comparing AutoSaddler against Meta-Harness and GEPA.

Figure 5 reports compute and learning efficiency on Terminal-Bench 2.0. From the common 52.6% starting point, AutoSaddler reaches **73.7%** dev accuracy after only 31 task executions and 12 leveraged traces, outperforming Meta-Harness (63.2%) by 10.5 percentage points and GEPA (57.9%) by 15.8 percentage points. The $\sim 8\times$ reduction in leveraged traces relative to Meta-Harness (98 traces) mirrors the trend observed on GAIA2, indicating that AutoSaddler's diagnosis-guided patching delivers consistent optimization efficiency gains on terminal-centric tasks where failures are dominated by tool-use and environment-interaction errors rather than multi-hop reasoning.

I End-to-End Optimization Cost Characterization

Task-agent rollouts alone do not fully characterize the computational cost of automatic harness optimization. In addition to harness evaluation, each optimizer may incur substantial optimizer-side



(a) Compute Efficiency (b) Learning Efficiency

Figure 5: Comparison of optimization performance and efficiency on Terminal-Bench 2.0. **(a)** AutoSaddler reaches 73.7% dev accuracy with only 31 task executions, whereas Meta-Harness requires 98 executions to plateau at 63.2% and GEPA reaches 57.9% at 90 executions. **(b)** When measured by the number of execution traces leveraged for optimization, AutoSaddler achieves its best performance after leveraging only 12 traces, over 8× fewer than Meta-Harness (98 traces), while surpassing it by 10.5 percentage points.

LLM overhead for candidate generation, diagnosis, patching, reflection, and candidate selection. We therefore separately profile (i) optimizer-side LLM overhead and (ii) task-agent evaluation cost on GAIA2.

Optimizer-side cost. Table 13 reports the number of generated, rejected, and accepted patches together with the average wall-clock time, monetary cost, LLM calls, and token usage per generated patch. GEPA incurs the lowest optimizer-side cost at $5.50 per patch, but searches only over the system prompt. Meta-Harness and AutoSaddler instead optimize broader harness components and therefore provide a more direct comparison. AutoSaddler costs $14.56 per generated patch, $1.91 more than Meta-Harness, while requiring 533 seconds rather than 883 seconds per patch, corresponding to 39.6% lower wall-clock time. This difference holds despite AutoSaddler explicitly performing diagnosis, structured patch generation, reflection, and evolution.

Table 13: **Optimizer-side cost on GAIA2**. Runtime, monetary cost, LLM calls, and token usage are averaged per generated patch.

Method	Generated Patches	Rejected Patches	Accepted Patches	Wall-Clock Time (s) / Patch	Cost ($) / Patch	LLM Calls / Patch	Output Tokens / Patch	Cache-Creation Input Tokens / Patch	Cache-Read Input Tokens / Patch
GEPA	76	58	18	386	5.50	21.3	11,416	156,788	280,940
Meta-Harness[5]	20	–	–	883	12.65	66.9	43,429	250,466	3,528,484
AutoSaddler	39	19	20	533	14.56	76.1	21,464	430,933	3,246,158

Task-agent evaluation cost. Harness evaluation is considerably more expensive than the optimizer-side operations above. A single GAIA2 task-agent rollout requires, on average, 20.2 LLM calls, 550,988 input tokens, 7,380 output tokens, and 203.9 seconds of wall-clock time, as summarized in Table 14.

Table 14: **Average task-agent evaluation cost per GAIA2 rollout**.

Metric	Average per Rollout
LLM calls	20.2
Input tokens	550,988
Output tokens	7,380
Wall-clock time	203.9 s

[5]Meta-Harness does not employ an explicit patch-acceptance gate; therefore, rejected and accepted patch counts are not applicable.

The evaluation protocols of the methods differ substantially in how often this expensive operation is invoked. Meta-Harness evaluates the union of all 140 GAIA2 training and development scenarios for every candidate update. In contrast, AutoSaddler evaluates six training scenarios per optimization iteration and invokes the 65-scenario development-set evaluation only for patches that improve the training mini-batch. Consequently, AutoSaddler reaches 67.7% development accuracy after 391 task-agent rollouts, already exceeding the 61.5% peak reached by Meta-Harness after 1,400 rollouts. Thus, while AutoSaddler incurs moderately higher optimizer-side monetary cost per generated patch, this overhead is offset by substantially more selective use of expensive task-agent evaluations and higher performance under a much smaller rollout budget.

J Search Trajectory Visualization

To illustrate how the reflection and evolution sessions jointly shape the search trajectory, Figure 6 visualizes the dev-score progression of AutoSaddler across 50 iterations (2 epochs) on GAIA2. Each node represents a harness that was accepted on its mini-batch and subsequently evaluated on the held-out dev set (21 of 51 total candidates); the displayed score is the dev-set accuracy. Solid edges denote base inheritance (thin gray for sequential, bold dark for rebase), while red dashed edges denote cherry-pick merges, the operations that lead to a DAG structure rather than a simple chain.

The trajectory exhibits four phases. In the *Foundation Building* phase (Iter0–Iter8), the evolution session selects the single available predecessor at each step, producing a linear chain. In the *Rapid Improvement* phase (Iter8–Iter15), five consecutive accepts build linearly on each other, culminating at Iter13 (67.7%). The DAG structure emerges in the *Selective Merging & Repair* phase (Iter16–Iter27): after Iter20's catastrophic regression to 33.8% caused by a hook on a high-frequency tool, the evolution session rebases to Iter13 and cherry-picks proven fixes from Iter13 and Iter14 into Iter21 and Iter22, while reverting harmful changes identified through reflection. Iter27 inherits these curated patches via a linear chain and achieves the global peak of 72.3%. In the *Consolidation & Rebase* phase (Iter31–Iter47), all branches rebase on Iter27. When Iter46 over-accumulates eight cherry-picked hooks and rules from Iter40/Iter44/Iter45, causing a -12.3 pp drop, the evolution session at Iter47 diagnoses this as collective steering overhead and prunes to only four conservative patches, recovering to 69.2%. Overall, this trajectory highlights how AutoSaddler uses reflection-guided rebasing and selective merging to preserve beneficial changes, discard harmful ones, and recover from regressions during harness search.

K Additional Analysis for RQ1

Tool-Call and File-Access Overhead of In-Depth Diagnosis A central design choice of AutoSaddler is the in-depth diagnosis stage, which is intended to perform a more thorough investigation of failure traces than shallow reflection as in the *"w/o In-Depth Diagnosis"* ablation setting. To verify that this stage indeed conducts substantively deeper analysis, rather than merely adding superficial overhead, we empirically measure the additional tool calls and file accesses that AutoSaddler expends compared to the ablation setting. In AutoSaddler, the Claude Agent SDK performs both diagnosis and patching, whereas in the ablated variant the SDK is responsible only for patching. The gap between the two therefore directly reflects the investigative effort attributable to in-depth diagnosis.

Counting tool calls is straightforward, as each invocation of a Claude Agent SDK tool is logged. However, accurately counting file accesses is non-trivial because the Claude Agent SDK occasionally delegates work to sub-agents, and tool calls issued inside sub-agents are not recorded in the main trace. To ensure a consistent measurement across both variants, we count file accesses for the main agent using the following criteria:

- **File access through dedicated file tools.** The number of invocations of the Claude Agent SDK's dedicated file-handling tools, namely, `Read`, `Write`, `Edit`, `MultiEdit`, and `Grep`.
- **File access through Bash commands.** The number of executions of shell commands that read or manipulate file contents, including `cat`, `wc`, `grep`, and Python invocations that open files (e.g., `python3` or `python` scripts containing `open()` calls).

The same accounting rule is applied to both AutoSaddler and the ablated variant, enabling a direct comparison of investigative effort.

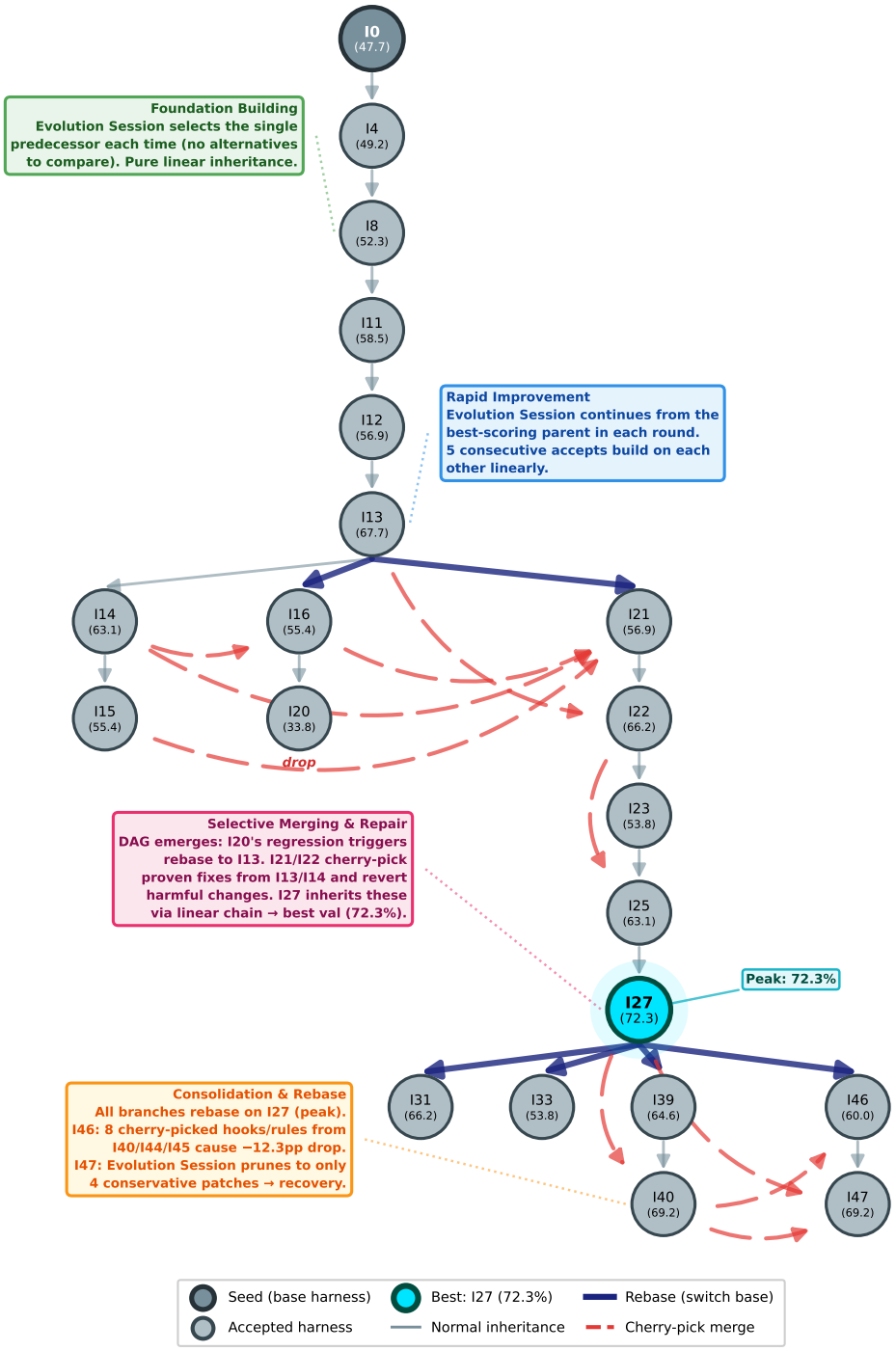


Figure 6: Evolutionary search trajectory of AutoSaddler as the EvoDAG. Four phases (foundation, rapid improvement, selective merging, and consolidation) lead to a global dev-set peak of 72.3% at Iter27, escaping the Iter20 regression and local optima.

Table 15 reports the average tool calls and file accesses per optimization step for AutoSaddler and the *"w/o In-Depth Diagnosis"* ablation setting. AutoSaddler performs 69.7 tool calls and 45.5 file accesses on average, compared to 63.5 and 39.7 for the ablated variant, corresponding to 6.2 additional tool calls and 5.8 additional file accesses per step.

Table 15: Average tool calls and file accesses per optimization step for AutoSaddler and its ablated variant without in-depth diagnosis.

Method	Avg. Tool Calls	Avg. File Accesses
w/o In-Depth Diagnosis	63.5	39.7
AutoSaddler	69.7	45.5
Δ (Attributable to In-Depth Diagnosis)	+6.2	+5.8

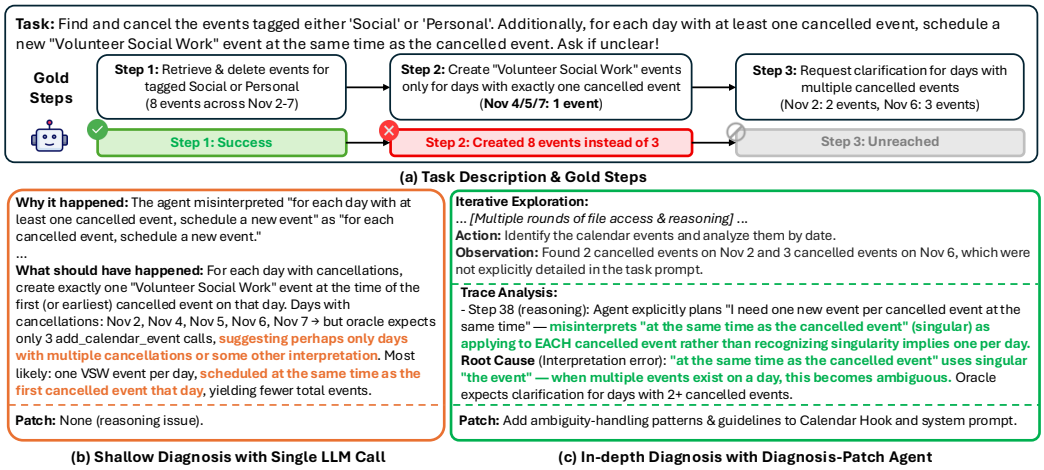


Figure 7: Case study on a calendar task with ambiguous cancellations, where AutoSaddler's in-depth diagnosis accurately identifies the true root cause via deep debugging, unlike shallow diagnosis.

Qualitative Case Studies. Figure 7 illustrates how this depth translates into diagnostic accuracy. In this task, the agent should have created new events only for days with a single cancellation (Nov 4, 5, 7) and requested clarification for days with multiple cancellations (Nov 2, 6). Instead, ignoring task ambiguity, it created new events for all canceled events. Without thorough investigation, shallow diagnosis incorrectly hypothesized that the agent simply misunderstood the instruction, concluding that it should have matched the time of the first cancellation on days with multiple cancellations. Consequently, it misclassified the issue as a simple LLM reasoning error and bypassed patching. In contrast, AutoSaddler identifies the calendar events and analyzes them by date, confirming multiple cancellations on Nov 2 and 6, and cross-verifies this with the oracle events. This enabled it to pinpoint the true root cause: the linguistic ambiguity of the singular phrase "the event". It successfully resolved the issue by patching the calendar hook and system prompt with an ambiguity-handling guideline.

Figure 8 illustrates a complementary failure mode in which shallow reflection produces a plausible-sounding but factually incorrect diagnosis. In this task, the agent should have located the existing `benchmarks` directory under `Documents/` before creating `sims3` inside it. Instead, the agent created `benchmarks/sims3/` at the filesystem root without verifying where `benchmarks` actually resides. Shallow reflection diagnoses the failure as the agent's use of an incorrect *relative* path despite knowing that the benchmarks are located under `Documents/`, which is inconsistent with the agent's actual behavior of never verifying the location at all. Consequently, it proposes a reactive patch that only modifies `mkdir` to emit a warning when the parent directory is newly created, leaving the agent's lack of pre-verification untouched. In contrast, AutoSaddler inspects the agent's filesystem trace and observes that the agent called `exists("benchmarks/sims3")` and immediately invoked `mkdir` on the same path without searching for the true location, then cross-verifies this against the environment filesystem structure to confirm that `benchmarks` resides at `Documents/benchmarks/`. This enabled it to pinpoint the true root cause: the absence of parent-directory verification before `mkdir` and `mv` calls. It successfully resolved the issue by adding automatic path resolution to `execute_tool_call()`, eliminating the silent misplacement at its source.

Figure 9 illustrates a case in which the failure stems from a candidate that the agent never observed. In this task, the agent should have detected two candidate rides on September 30 starting from a

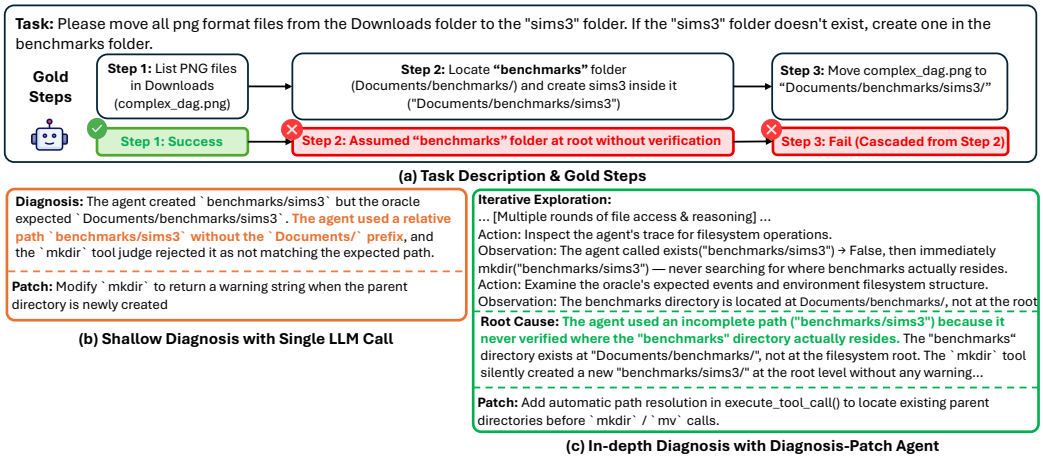


Figure 8: Case study on a file-management task that requires moving PNG files into a `sims3` folder under `benchmarks`. Shallow diagnosis misattributes the failure to a relative-path error, while AutoSaddler verifies the filesystem and locates the true cause: an unverified parent directory.

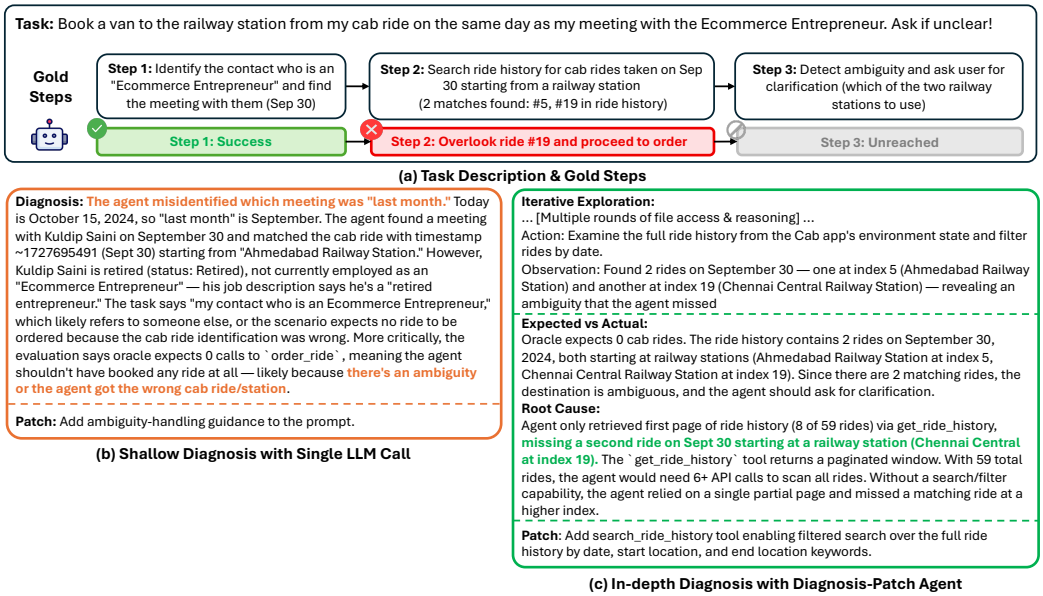


Figure 9: Case study on a cab-booking task with two candidate rides on the same day. Shallow diagnosis overlooks the second matching ride and proposes an off-target fix, while AutoSaddler inspects the full ride history and identifies the missed candidate.

railway station and asked the user to disambiguate the destination. Instead, the agent confidently booked a van using only the first matching ride, overlooking a second candidate at index 19. Without examining the full ride history, shallow reflection fails to recognize that a second matching ride exists at index 19 and instead produces a diagnosis centered on contact identification and oracle-call counts, missing the actual cause of the failure. Consequently, it proposes a fix that does not address the overlooked candidate. In contrast, AutoSaddler directly examines the full ride history from the Cab app's environment state and filters by date, finding two rides on September 30, one at index 5 (Ahmedabad Railway Station) and another at index 19 (Chennai Central Railway Station), and cross-verifies this against the oracle's expected zero `order_ride` calls. This enabled it to pinpoint the true root cause: the agent missed the matching ride at index 19 and thus failed to detect the destination ambiguity. It successfully resolved the issue by adding a `search_ride_history` tool

that enables filtered search over the ride history using date, start-location, and end-location keywords, surfacing conflicting candidates in a single call and triggering the correct clarification behavior.

Accumulation of Accepted Patches. Figure 10 visualizes the cumulative number of accepted patches over training iterations, complementing the discussion in the main text. The results show that the gap between AutoSaddler and the ablation emerges early in training, by Iteration 25, and persists throughout Epoch 2. This confirms that the in-depth diagnosis component leads to more accepted patches and ultimately better performance.

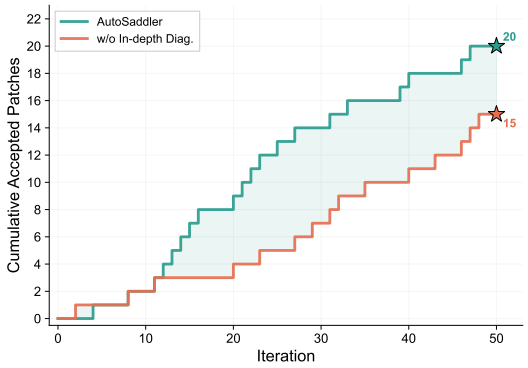


Figure 10: **Cumulative accepted patches.** AutoSaddler accumulates 20 accepted patches, whereas the *w/o In-depth Diagnosis* ablation reaches 15.

L Additional Analysis for RQ2

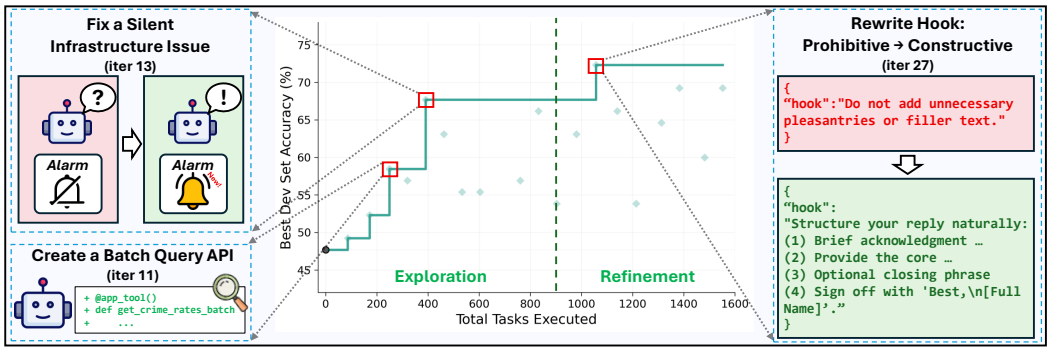


Figure 11: **Representative patches discovered over the optimization trajectory.** During the Exploration phase (Iterations 1–25), the search is dominated by Capability patches, including New Tool Addition, Implementation Fix, and Agent Loop Logic Change, which address structural limitations in the harness (e.g., Iterations 11 and 13). During the Refinement phase (Iterations 26–50), the search shifts toward Steering patches that refine prompt rules, tool descriptions, and PreToolUse hooks (e.g., Iteration 27). Together, these two phases produce a cumulative 24.6 *pp* improvement.

The development-set optimization trajectory in Figure 11 illustrates the concrete mechanism behind Structured Patching and Phased Patch Scheduling. During the Exploration phase (Iterations 1–25), the search prioritizes Capability patches, which address structural limitations that cannot be resolved through prompt edits alone. For example, at Iteration 11, AutoSaddler introduces a batched API, `get_crime_rates_batch`, that consolidates multiple serial queries and substantially reduces step-budget consumption. At Iteration 13, AutoSaddler identifies an execution stall in the environment's polling logic and resolves it by implementing a notification-based exit strategy.

After these capability-level improvements stabilize the harness, the Refinement phase (Iterations 26–50) shifts the search toward Steering patches that fine-tune agent behavior. For instance, at Iteration 27, AutoSaddler replaces a prohibitive hook with a constructive response template, recovering cases

that had previously failed due to over-correction and reaching the peak dev-set accuracy of 72.3%. Overall, the trajectory suggests that Structured Patching and Phased Patch Scheduling first expand the harness's functional capacity and then refine its decision-making behavior, yielding a cumulative $24.6\,pp$ improvement.

M Additional Analysis for RQ3

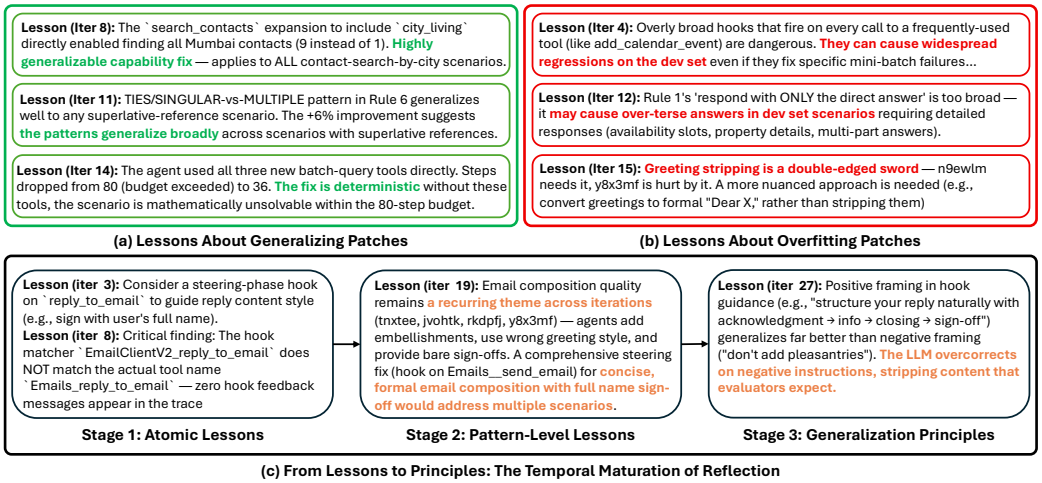


Figure 12: Reflection extracts lessons from (a) patches that generalize and (b) patches that overfit despite identical mini-batch outcomes. (c) The lineage of `reply_to_email`-related failures illustrates how atomic lessons gradually evolve into a generalization principle.

We further investigate the mechanism underlying the regression gap between AutoSaddler and the "*w/o Generalization-Aware Selection*" ablation setting by examining the lessons extracted by reflection at each iteration (Figure 12). Two key observations emerge. First, panels (a) and (b) demonstrate that the reflection session, by jointly analyzing dev-set score fluctuations and patch contents, accurately differentiates between generalizing and overfitting patches. For example, reflection correctly endorses patches covering coherent scenario classes—such as a search-field expansion for city-based queries (Iter 8) or a tie-breaking rule for superlative references (Iter 11). Conversely, it successfully flags overfitting patches whose side effects spill over into unintended scenarios, such as a calendar hook firing indiscriminately on all event creations (Iter 4) or a conciseness rule damaging scenarios requiring structured detail (Iter 12). These accumulated judgments provide the crucial filtering signal consumed by the generalization-aware selection. Second, panel (c) traces a single problem—email reply quality—across four Iterations and shows that reflection's knowledge of this problem changes form: from scenario-specific diagnostics (Iter 3 identifies the need for a reply hook; Iter 8 catches a matcher bug), to a pattern recognized across four scenarios (Iter 19), and finally to a scenario-independent principle on how steering text should be framed (Iter 27). What matures is not the problem itself but reflection's level of abstraction over it—evidence of cumulative learning rather than repeated rediscovery.

N Catalog of Discovered Patches on GAIA2

Table 16 presents representative patches between the *base harness* (the seed harness prior to optimization) and the *final harness* $\hat{\theta}_{\mathrm{AS}}$ that AutoSaddler produces after training on GAIA2. Patches are grouped by category and sub-type following the taxonomy in Table 1; inserted text is highlighted in teal, and italicized base-harness entries indicate that the corresponding component or feature did not exist before the patch.

Table 16: Representative patches discovered by AutoSaddler on GAIA2, producing the final harness $\hat{\theta}_{\mathrm{AS}}$ from the base harness. C = Capability, S = Steering. Inserted text is highlighted in teal.

C/S	Component	Base Harness	Final Harness ($\hat{\theta}_{\mathrm{AS}}$)
Prompt Patch	*(modifications to `default_agent/prompts/system_prompt.py`)*		
	Prompt Rule Modification		
S	Rule 1 (Communication)	1. COMMUNICATION: Only message the user when completely done or if the task is impossible.	1. COMMUNICATION: Only message the user when completely done or if the task is impossible. When the user asks a factual question expecting a specific answer (a city name, a number, a person's name, etc.), respond with ONLY the direct answer — no extra calculations, context, or formatting.
S	Rule 6 (Ambiguity)	6. AMBIGUITY: Execute all clear and unambiguous parts of a request immediately. When you encounter ambiguities, contradictions, or impossible elements, finish unambiguous subtasks and then stop and explicitly ask the user for clarification before proceeding with those specific parts.	6. AMBIGUITY: Execute all clear and unambiguous parts … ask the user for clarification before proceeding with those specific parts. A subtask is CLEAR if all its required tool parameters (recipients, content, IDs, paths) can be fully determined — even if it semantically relates to an ambiguous part. Common ambiguity patterns that REQUIRE clarification: (a) TIES — when the task references 'the item with the highest/lowest X' but multiple items are tied at that value; (b) SINGULAR vs MULTIPLE — when the task uses singular language ('the event', 'the property') but multiple items match the criteria; (c) PER-GROUP vs PER-ITEM — when applying an action 'for each day/group' at a time derived from a singular source, but multiple sources exist within that group.
	Prompt Rule Addition		
S	Rule 7 (Task Decomposition)	*(rule does not exist)*	7. TASK DECOMPOSITION: When the user's request contains multiple ordered steps (signaled by words like "then", "after that", "next", "afterwards"), treat each step as a separate subtask. Complete each subtask fully before moving to the next. If a later subtask has ambiguities but earlier subtasks are clear, execute all clear subtasks first, then ask about the ambiguous ones. CRITICAL: Ordering words create HARD DEPENDENCY BOUNDARIES — subtasks after the boundary MUST NOT be executed until all preceding subtasks are completed, even if the later subtask's parameters are fully determinable.
Tool Patch			
	New Tool Addition		
C	`CityApp.get_crime_rates_batch`	*(tool does not exist)*	`get_crime_rates_batch(zip_codes: list[str]) -> dict` Batch crime-rate lookup over many zip codes in a single call; replaces N serial queries that exhausted the step budget.
C	`EmailClientApp.get_email_addresses_in_folder`	*(tool does not exist)*	`get_email_addresses_in_folder(folder_name: str = "INBOX") -> dict` Returns all unique sender/recipient addresses in a folder for fast contact discovery.

(continued on next page)

(Table 16 continued from previous page)

C/S	Component	Base Harness	Final Harness ($\hat{\theta}_{\mathrm{AS}}$)
C	`MessagingApp.get_all_participants`	*(tool does not exist)*	`get_all_participants() -> dict` Returns all unique participant names across conversations, excluding "Me".
C	`MessagingAppV2.get_all_participants`	*(tool does not exist)*	`get_all_participants() -> dict` V2 analog: returns unique participant IDs, excluding the current user.
C	`CabApp.search_ride_history`	*(tool does not exist; only paginated `get_ride_history` was available)*	`search_ride_history(date: str | None = None, start_location_keyword: str | None = None, end_location_keyword: str | None = None) -> dict` Searches the full ride history with optional filters in one call.
	Implementation Fix		
C	`ContactsApp.search_contacts`	Searches `first_name`, `last_name`, `phone_number`, `email` only.	Additionally searches `city_living`, `country`, `nationality`, `job`, `address`, enabling city- and role-based contact lookup.
Middleware Patch			
	PreToolUse Hook		
S	`Emails__reply_to_email` (reminder text in `hook.json`)	"Do not add unnecessary pleasantries or filler text." (prohibitive: triggers over-correction — agent strips content evaluators expect.)	"Structure your reply naturally: (1) brief acknowledgment, (2) provide the core information, (3) optional closing phrase, (4) sign off with 'Best,\n[Full Name]'." (constructive: provides a positive structural template.)
	Agent Loop Logic Change		
C	`are_simulation_main.py` (agent loop)	The benchmark environment does not notify the agent of cab status changes (e.g., delays). Without a mechanism to advance simulated time between turns, the agent loop stalls indefinitely, unable to reach the point where it could proactively discover these state changes.	AutoSaddler resolves the stall by inserting an inter-turn time-advancement call: when no messages are pending, the loop invokes `wait_for_notification(timeout=300)` to fast-forward simulated time by up to 300 seconds. If no notification arrives, the timeout itself wakes the agent, prompting it to proactively poll for state changes.
C	`_resolve_fs_path` in `execute_tool_call()` on `*__mkdir` (`path`), `*__mv` (`path2`)	`path="benchmarks/sims3"` (no automatic resolution: `mkdir` silently creates a new `benchmarks/sims3/` at the filesystem root, despite an existing `Documents/benchmarks/` elsewhere.)	`path="Documents/benchmarks/sims3"` (automatic path resolution: locates existing parent directories with the same basename via filesystem search before `mkdir`/`mv` is invoked.)

O Catalog of Discovered Patches on Terminal-Bench 2.0

Table 17 presents representative patches between the *base harness* (the seed harness prior to optimization) and the *final harness* $\hat{\theta}_{\mathrm{AS}}$ that AutoSaddler produces after training on Terminal-Bench 2.0. Patches are grouped by category and sub-type following the taxonomy in Table 1; inserted text is highlighted in teal, and italicized base-harness entries indicate that the corresponding component or feature did not exist before the patch.

Table 17: Representative patches discovered by AutoSaddler on Terminal-Bench 2.0, producing the final harness $\hat{\theta}_{\mathrm{AS}}$ from the base harness. C = Capability, S = Steering. Inserted text is highlighted in teal.

C/S	Component	Base Harness	Final Harness ($\hat{\theta}_{\mathrm{AS}}$)
Prompt Patch	*(modifications to the system prompt template* `prompt-templates/verify-before-complete.txt`*)*		
	Prompt Rule Addition		
S	Verification before completion	*(no verification guidance; the base prompt ends immediately after the format specification)*	CRITICAL – Verification before completion: You MUST verify your solution works before setting `"task_complete": true`. When setting up services, test endpoints. When creating files, verify contents. Verify file permissions for all service users. Perform a dry run of automated hooks. If verification fails, fix and verify again before completing.
S	Different-input testing & build-artifact clean-up	*(rule does not exist)*	For scripts or programs, test with a DIFFERENT input than the provided example to confirm generalizability. After testing, clean up the output directory: remove any build artifacts (compiled binaries, `.o` files), temporary files, or intermediate outputs that the task did NOT ask you to create. Verifiers often check that output directories contain ONLY the expected files.
S	Library preference	*(rule does not exist)*	When writing scripts, check the terminal output for pre-installed packages and prefer established libraries over fragile custom implementations.
S	HTML sanitization	*(rule does not exist)*	For HTML sanitization / XSS filtering, ALWAYS use BeautifulSoup (not regex). Decompose dangerous tags (`script`, `noscript`, `iframe`, `object`, `embed`, `frame`, `style`) and do a string replacement to strip `javascript:` from entire output as final safety pass.
Tool Patch			
	Argument Modification		
C	`_limit_output_length` (`max_bytes` parameter)	`_limit_output_length(output, max_bytes=10000)` Default 10 KB limit truncates large file contents or command outputs, causing the agent to miss critical information.	`_limit_output_length(output, max_bytes=30000)` 30 KB limit provides more room while keeping context manageable.
	Implementation Fix		
C	`_get_prompt_template_path` (prompt template selection)	Returns the **default** Terminus-2 prompt template path. The default prompt contains only JSON format specification without domain-specific guidance.	Returns path to `verify-before-complete.txt` Custom prompt template that includes verification guidance, library-preference rules, and coding best practices alongside the standard format specification.
Middleware Patch			
	PreToolUse Hook		

(continued on next page)

(Table 17 continued from previous page)

C/S	Component	Base Harness	Final Harness ($\hat{\theta}_{\mathrm{AS}}$)
S	`_get_completion_confirmation_message` (completion confirmation)	"Current terminal state:\n{output}\n \n Are you sure you want to mark the task as complete? This will trigger your solution to be graded and you won't be able to make any further corrections." (generic confirmation; agent often skips verification)	"Current terminal state:\n{output}\n \n STOP – You must verify your solution before confirming completion.\n For service tasks: test endpoints, verify permissions.\n For coding tasks: test with a DIFFERENT input, verify output structure.\n Clean up: remove build artifacts the task did NOT request." (structured verification checklist with task-type-specific guidance)
Agent Loop Logic Change			
C	`run()` (environment discovery)	Agent starts with no knowledge of what packages are pre-installed in the Docker container. Frequently chooses fragile custom implementations (e.g., regex for HTML) over established libraries that are already available.	Before the agent loop begins, `pip list --format=columns` is executed in the container so the initial terminal state shows all pre-installed Python packages. The agent can then choose appropriate libraries (e.g., BeautifulSoup over regex) from the start.

P Instructions Used In AutoSaddler

To ensure reproducible and principled optimization, AutoSaddler employs a multi-session orchestration design where each session is guided by a specialized prompt template. These instructions are designed to operationalize the core pillars of our framework: in-depth debugging, structured harness intervention, and generalization-aware selection.

The **Diagnosis-Patch Session** (Figure 13) initiates the optimization loop by grounding the agent in the current mini-batch failures. Unlike simple reflection-based approaches, this prompt enforces a rigorous "diagnose-before-patch" workflow. It provides the agent with access to raw execution traces alongside the harness codebase and mandates the use of specialized skills (e.g., `history-analysis`, `diagnose`) to identify the underlying root causes. Furthermore, it implements our phased scheduling by restricting the agent to either *capability* or *steering* patches, ensuring that interventions are targeted and architecturally sound.

The **Reflection Session** (Figure 14) focuses on causal attribution and the extraction of durable lessons. The prompt instructs the agent to perform a comparative analysis of pre- and post-patch execution traces for every scenario in the mini-batch. A key feature of this session is the mandatory "stochasticity check", where the agent must distinguish between true causal fixes/regressions and stochastic artifacts caused by LLM non-determinism. By categorizing outcomes into four states (*fixed, regressed, still-failing, still-passing*) and recording detailed evidence in the EvoDAG, this session ensures that only verified insights drive the evolution process.

Finally, the **Evolution Session** (Figure 15) serves as the framework's meta-optimizer. It provides the agent with a global view of the EvoDAG history, including accumulated lessons, performance metrics across development sets, and code diffs from prior iterations. The prompt guides the agent to synthesize the next harness candidate not just by incremental editing, but by potentially merging successful components from different lineages or reverting regressive updates. This session ensures that the resulting harness base is stable, verified for syntax and logic, and optimized for broad generalization across the task distribution.

Diagnosis-Patch Prompt

```
## Mandatory Skills
You MUST read and follow the SKILL.md for each skill listed below.  Do NOT skip or
summarize any skill -- execute the full procedure described in each one.  These skills
are installed at '.claude/skills/<n>/SKILL.md' in the current worktree.
| Order | Skill | When | Why |
|----|----|---|---|
```

(continued)

```
| 1 | `history-analysis` | Before any other work (Step 1) | Structured analysis of the full evolution history -- identifies relevant lessons, prior attempts, proven strategies, and regression-prone areas |
| 2 | `diagnose` | Before applying any patch (Step 2) | Root-cause analysis from agent traces and codebase -- pinpoints exactly why each failing scenario fails |
| 3 | `capability-patch` or `steering-patch` | When applying patches (Step 3) | Phase-appropriate patch methodology -- `capability-patch` during capability phase, `steering-patch` during steering phase. Follow the skill matching the current phase |
| 4 | `patch-verification` | After applying all patches (Step 4) | Crash-safety verification -- syntax, imports, docstrings, hooks, logic. A crashing patch is worse than no patch |

> Enforcement: Every step above is mandatory and sequential. Do NOT apply patches without first completing diagnosis. Do NOT finish the session without running the full `patch-verification` procedure. Do NOT skip `history-analysis` -- it prevents repeating known failures.

## Goal

Fix failing scenarios in this mini-batch by diagnosing root causes from execution traces and the agent codebase, then applying targeted code patches -- without regressing scenarios that already pass.

Each scenario is a task the agent must solve. The initial evaluation on the mini-batch has already run the agent on each scenario and recorded its execution trace: every tool call, every response, and every reasoning step. Your job is to read these traces alongside the agent codebase, identify why the agent failed, determine what code change would fix the root cause, and apply it.

The optimization objective is to maximize performance across the full task distribution -- not just this mini-batch. Patches should address general behavioral patterns, not hardcode scenario-specific answers. A patch that fixes one scenario by adding a general rule will also help similar unseen scenarios; a patch that hardcodes a specific answer helps only that one scenario and may harm others.

Acceptance: A patch is accepted if the sum of re-evaluation scores exceeds the sum of initial scores. Even fixing a single scenario is enough. However, regressions (passing → failing) are recorded as bad patterns -- avoid them.

## Context

- Iteration: {iteration}
- Candidate: C{candidate_idx}
- Current worktree: `{worktree_path}`
- Base parent: C{base_parent_idx} (worktree: `{parent_worktree}`)
{cherry_pick_parents_section}- Phase: {phase}
- Initial evaluation output: `{before_output_dir}`

## Current Mini-Batch ({num_scenarios} scenarios)

{mini_batch_listing}

### Initial Scores (pass rate: {before_pass_rate})

{before_scores_listing}

## Patch Types for This Phase

{patch_types_section}

## Workflow

### 1. Analyze history

MANDATORY first step. Run the `history-analysis` skill to build a structured understanding of the full evolution history. Do NOT skip this step or truncate `evo-dag show history` with `head`/`tail`. The skill provides the methodology for reading the complete history without truncation. Focus on:

- Relevant lessons (good/bad patterns) for the current mini-batch
- Prior attempts on the failing scenarios -- what worked and what failed
- Proven patch strategies and regression-prone areas

### 2. Diagnose failing scenarios

For each failing scenario, use the `diagnose` skill to identify the root cause. The key is to find why the agent made the wrong decision, not just what went wrong. Diagnosis requires reading both the execution trace and the agent codebase.

Trace files (output dir: `{before_output_dir}`): See CLAUDE.md's Iteration Output Structure section for the exact file layout. Key files to read:
```

(continued)

```
- Evaluation rationale:  Per-scenario scores and judge rationale -- the evaluator's
summary of what was expected vs what the agent produced.
- Agent execution traces:  Per-scenario full tool call, response, and reasoning traces --
where you find the exact point where the agent diverged from correct behavior.
Agent codebase:  current worktree `{worktree_path}`
The `diagnose` skill provides the full methodology -- follow it for each failing
scenario.
### 3.  Apply patches
Based on your diagnosis, use the phase-appropriate patch skill:
- Capability phase (`capability-patch`):  Expand what the agent CAN DO -- new tool
methods, parameter additions, implementation fixes, infrastructure changes.  These
unlock scenarios that are unreachable through prompt tuning alone.
- Steering phase (`steering-patch`):  Refine HOW the agent behaves -- prompt rules, tool
description corrections, PreToolUse hooks.  These fine-tune the agent's use of existing
capabilities.
Patch guidelines:
- Target the root cause, not the symptom.  If the agent sends a wrong email subject, the
fix is a general rule about deriving subjects from user wording -- not hardcoding the
correct subject for one scenario.
- Keep patches generalizable.  Rules should be abstract:  no scenario IDs, no specific
names, no hardcoded answers.  A good patch helps all scenarios that share the same root
cause pattern.
- Align agent-facing text with capability changes.  When you add or modify tools,
parameters, or infrastructure code, also update the system prompt, tool docstrings,
and/or hooks so the agent is aware of the changes.  Check for conflicting rules.
- Learn from patch history.  Check `evo-dag show history` -- it contains diffs,
per-scenario results, reflections, and accumulated lessons.  Use it to understand:
which patches generalized well to the dev set, which caused regressions, which
root-cause patterns were effectively resolved, and which approaches repeatedly failed.
Build on proven strategies and avoid repeating known bad patterns.
- Prefer replacement over addition when modifying prompts.  The system prompt has a
finite attention budget -- adding rules without removing or merging existing ones
dilutes their impact.
### 4.  Verify patches
After applying all patches, you MUST run the `patch-verification` skill to ensure the
patched codebase has no runtime errors.  A patch that crashes at runtime is worse than
no patch -- verification is mandatory before committing.  Do NOT skip this step under
any circumstances.
### 5.  Write reasoning and record intent
After applying all patches, write your reasoning to `proposer_reasoning.md` in the
working directory root.  For each targeted scenario, include:  task description,
expected vs actual behavior, trace analysis (which step went wrong), root cause, and
resolution strategy (generalizable rule).  This sharpens the `-diagnosis` that
propagates to future iterations via `evo-dag show history`.  Then record intent:
    ```bash
    evo-dag update-intent \
      -target-scenarios "id1,id2" \
      -diagnosis "Root cause analysis of why target scenarios fail" \
      -approach "Brief description of the patch approach" \
      -files-changed "f1.py,f2.py" \
      -change-summary "What was changed and why"
    ```
CRITICAL: You MUST run `evo-dag update-intent` before finishing this session. Failure
to do so will result in incomplete iteration records and degrade the quality of future
iterations' patch history analysis.
What Happens Next
After this session, the outer loop re-evaluates the patched worktree on the same
mini-batch, computes initial/re-evaluation per-scenario impacts (fixed, regressed,
still_failing, still_passing), and records the patch verdict. If the patch is accepted
(re-evaluation score > initial score), the candidate is also evaluated on the full
```

(continued)

```
dev-set to measure generalizability.  Then Session 2 (Reflection) runs, where you
analyze the results and record structured reflections for each scenario.
```

Figure 13: Diagnosis-Patch Session Prompt: agent receives the current mini-batch's failing scenarios with execution traces, diagnoses root causes via the agent codebase, and applies targeted, generalizable patches to the existing harness.

Reflection Session Prompt

```
## Mandatory Skills
You MUST read and follow the SKILL.md for each skill listed below.  Do NOT skip or
summarize any skill -- execute the full procedure described in each one.  These skills
are installed at `.claude/skills/<n>/SKILL.md` in the current worktree.
| Order | Skill | When | Why |
|----|----|---|---|
| 1 | `history-analysis` | Before any other work (Step 1) | Structured analysis of the
full evolution history -- provides historical context for each scenario, pattern
evolution, and dev score attribution needed for meaningful reflections |
| 2 | `diagnose` | For every regressed AND fixed scenario (Step 3) | Causal attribution
-- determines whether each state change (PASS↔FAIL) was truly caused by the patch or is
a stochastic artifact from LLM non-determinism.  Accurate classification is critical to
avoid polluting lessons with false signal |
> Enforcement:  Do NOT attribute any state change to "stochastic noise" or "LLM
non-determinism" without running the full `diagnose` skill procedure.  Every
fixed/regressed classification must cite specific evidence from the diagnosis.
## Goal
Analyze the initial/re-evaluation results for every scenario in the mini-batch and
record structured reflections that future iterations can learn from.
Your reflections feed directly into `evo-dag show history` -- the primary knowledge
store that future iterations consult when diagnosing failures, choosing patch
strategies, and avoiding past mistakes.  The quality of your reflections determines how
effectively the pipeline learns.  Be specific:  reference exact tool calls, code paths,
and behavioral patterns.  Vague reflections waste learning potential.
## Context
- Iteration:  {iteration}
- Candidate:  C{candidate_idx}
- Current worktree:  `{worktree_path}`
- Base parent:  C{base_parent_idx} (worktree:  `{parent_worktree}`)
- Phase:  {phase}
- Initial evaluation output:  `{before_output_dir}`
- Re-evaluation output:  `{train_after_cycle_dir}`
## Session 1 Reasoning
The following is the proposer's diagnosis and patch rationale from Session 1.  Use this
to understand the intent behind the patch -- what root causes were identified, what the
patch was designed to fix, and what behavioral changes were expected.
{proposer_reasoning}
## Results Summary
{results_summary}
## Per-Scenario Details
{per_scenario_details}
{generalization_section}
## The Four Outcomes
| Before | After | Status | Key Question |
|----|----|----|-------|
| FAIL | PASS | `fixed` | What root cause did the patch resolve?  How?  |
| PASS | FAIL | `regressed` | What did the patch break?  Why?  |
| FAIL | FAIL | `still_failing` | Why was the patch insufficient?  What to try next?  |
| PASS | PASS | `still_passing` | Did the patch interact with this scenario at all?  |
## Workflow
```

(continued)

```
### 1. Analyze history

MANDATORY first step. Run the ‘history-analysis‘ skill to build a structured
understanding of the full evolution history. Do NOT skip this step or truncate ‘evo-dag
show history‘ with ‘head‘/‘tail‘. The skill provides the methodology for reading the
complete history without truncation. Focus on:

- Historical context for each scenario in the mini-batch

- Pattern evolution -- emerging patterns not yet captured in lessons

- Dev score attribution -- which patches improved/hurt dev accuracy and why

This context is essential for writing meaningful reflections -- you need to know what
was tried before to explain why this iteration’s results differ, and to write
‘-prevention-or-next‘ guidance that adds new information rather than repeating existing
lessons.

### 2. Read traces (by priority)

Before traces are in ‘{before_output_dir}‘. After traces are in
‘{train_after_cycle_dir}‘. See CLAUDE.md’s Iteration Output Structure section for the
exact file layout within each output directory.

Compare before and after traces to understand what the patch changed in the agent’s
behavior.

Priority order:

1. Regressed -- always read both before and after traces. You must understand what
behavior was correct before and what the patch broke.

2. Still-failing (targeted) -- read to check for partial progress. Did the failure
point shift? Is the agent closer to correct behavior?

3. Fixed -- skim the after trace to confirm the patch resolved the root cause as
intended, not through a lucky side effect.

4. Still-passing -- skip unless the scenario uses the same tools or components you
modified.

### 3. Diagnose state-changed scenarios (MANDATORY)

For every regressed AND fixed scenario, run the full ‘diagnose‘ skill to determine
whether the state change was truly caused by the patch or is a stochastic artifact
(caused by LLM non-determinism).

LLM non-determinism means the same agent code can produce different tool-call sequences
across runs. This affects both directions:

- A regressed scenario (PASS→FAIL) may have nothing to do with the patch -- the agent
simply took a different reasoning path.

- A fixed scenario (FAIL→PASS) may not be a real fix -- the agent may have succeeded by
luck, not because of the patch. Recording this as a true fix pollutes ‘good_patterns‘
with false signal.

Accurate causal attribution is critical: false regressions accumulate noise in
‘bad_patterns‘, and false fixes accumulate noise in ‘good_patterns‘. Both degrade the
quality of lessons for future iterations.

How to distinguish true vs. stochastic state changes:

1. Read the before trace: Identify the exact tool-call sequence and reasoning steps
that led to the before result.

2. Read the after trace: Identify where the agent’s behavior diverged.

3. Check the code diff (‘evo-dag show edge {base_parent_idx} {candidate_idx}‘): Does
the diff touch any code path, prompt text, hook, or tool that the scenario exercises?
If the diff is completely unrelated to the scenario’s tool usage and divergence point,
it is likely stochastic.

4. Check the divergence point: Is the behavioral change at a step that the patch
modified (true causation) or at an unrelated step where the agent simply made a
different LLM-driven choice (stochastic)?

Classification criteria:

- True (regression or fix): The diff modifies code/prompt/hook that the scenario
directly exercises, AND the after trace shows the agent behaving differently at the
modified point in a way that explains the outcome change.

- Stochastic: The diff does NOT touch anything the scenario exercises, OR the agent’s
divergence point is unrelated to the patch (e.g., different search query phrasing,
different email wording).

- Uncertain: The diff touches a shared component but the causal link is unclear.
Record as uncertain with specific evidence.
```

(continued)

```
Do NOT attribute any state change to "LLM non-determinism" or "stochastic noise" without
performing the above analysis.  Every classification must cite:
- The specific divergence point in the traces
- Whether the diff touches the relevant code path
- The evidence for or against a causal link
Record the classification in `-explanation` and `-prevention-or-next`.  For true
regressions, explain what the patch broke.  For stochastic regressions, note the
evidence so future iterations don't over-correct.  For true fixes, explain the causal
chain from patch to fix.  For stochastic fixes, note the evidence so future iterations
don't over-rely on the patch strategy.
### 4.  Inspect code changes
Run `evo-dag show edge {base_parent_idx} {candidate_idx}` to see the code diff, files
changed, and per-scenario impacts.  Cross-reference the diff with the trace behavior to
understand causality.
For deeper inspection, read source files directly in the current worktree
`{worktree_path}`.
### 5.  Record reflections
Use `evo-dag update-reflection` for every scenario in the mini-batch.
Fields:
- `-root-cause`:  The underlying reason the scenario fails (independent of the patch).
What is the agent doing wrong and why?
- `-explanation`:  For `fixed` -- how the patch resolved it.  For `still_failing` -- why
the patch did NOT work (what was insufficient).  For `regressed` -- what the patch broke
and how.  For `still_passing` -- why the patch did not interfere (only for scenarios
using modified components).
- `-prevention-or-next`:  What to try next, or what to avoid.  This is critical for all
non-passing outcomes -- it creates the lessons that future iterations rely on.
- `-generalization-note`:  How this scenario's outcome relates to development set
accuracy.  Did the patch generalize beyond the mini-batch?  Record when dev scores are
available.
Per outcome:
`fixed` -- You MUST have completed the diagnosis step (step 3) before recording this
reflection.  Classify as true fix or stochastic artifact.
    ```bash
    # True fix (patch caused the success):
    evo-dag update-reflection \
      -node {candidate_idx} \
      -scenario "<id>" -status "fixed" \
      -root-cause "Agent called X with wrong param Y because docstring said Z." \
      -explanation "TRUE FIX: Patch corrected docstring to clarify Y. Agent now calls
correctly at step 4 -- directly caused by diff in tools/email.py L42." \
      -prevention-or-next "For similar tool-misuse scenarios, check docstring accuracy
first."
    # Stochastic artifact (not caused by the patch):
    evo-dag update-reflection \
      -node {candidate_idx} \
      -scenario "<id>" -status "fixed" \
      -root-cause "Agent happened to choose correct search query in after trace." \
      -explanation "STOCHASTIC: Diff only touches calendar tool.  This scenario uses
search tool exclusively.  Agent succeeded because it picked a better search query by
chance at step 2 -- unrelated to patch." \
      -prevention-or-next "Not a reliable fix -- scenario may fail again.  Root cause
(weak search strategy) still needs addressing."
    ```
`regressed` (highest priority) -- You MUST have completed the diagnosis step (step 3)
before recording this reflection. Classify as true regression or stochastic artifact
with evidence.
    ```bash
    # True regression (patch caused the failure):
```
```

```
    evo-dag update-reflection \
      -node {candidate_idx} \
      -scenario "<id>" -status "regressed" \
      -root-cause "This scenario relied on optional param Y in tool X." \
      -explanation "TRUE REGRESSION: Docstring change removed note about Y being
optional.  Agent stopped passing Y. Divergence at step 5 where agent no longer passes Y
-- directly caused by diff in tools/email.py L42." \
      -prevention-or-next "When modifying docstrings, preserve param optionality
annotations."
    # Stochastic artifact (not caused by the patch):
    evo-dag update-reflection \
      -node {candidate_idx} \
      -scenario "<id>" -status "regressed" \
      -root-cause "Agent used different search query phrasing in after trace." \
      -explanation "STOCHASTIC: Diff only touches calendar tool docstring.  This
scenario uses email tool exclusively.  Agent diverged at step 3 with different query
wording -- unrelated to patch." \
      -prevention-or-next "No action needed -- stochastic noise.  Do not over-correct
for this scenario."
    ‘‘‘
‘still_failing‘ -- Explain why the patch was insufficient.  What should the next
iteration try instead?
    ‘‘‘bash
    evo-dag update-reflection \
      -node {candidate_idx} \
      -scenario "<id>" -status "still_failing" \
      -root-cause "Agent uses absolute dates instead of relative dates in messages." \
      -explanation "Prompt rule ’use relative dates’ was too weak -- agent still used
’Oct 22’.  Partial progress:  bullet formatting was fixed." \
      -prevention-or-next "Stronger rule needed.  Consider PreToolUse hook on
send_message for just-in-time reminder."
    ‘‘‘
‘still_passing‘ -- For scenarios that use the same tools or components you modified,
explain which specific patch change could have caused interference and why it didn’t.
This is valuable signal for understanding patch safety.
    ‘‘‘bash
    evo-dag update-reflection \
      -node {candidate_idx} \
      -scenario "<id>" -status "still_passing" \
      -explanation "Uses send_message but passes because it sends user-provided verbatim
content.  Our rule only affects agent-composed messages."
    ‘‘‘
For scenarios unrelated to the patch, use the batch command:
    ‘‘‘bash
    evo-dag update-reflection -node {candidate_idx} -batch-still-passing "id1,id2,..."
    ‘‘‘
{generalization_workflow_step}
## Causal Attribution Checklist
Before finishing, verify each regressed AND fixed scenario’s reflection:
1.  Classified:  Explicitly labeled as TRUE (REGRESSION/FIX) or STOCHASTIC
2.  Evidence-based:  Cites the specific divergence point in traces
3.  Diff-linked:  States whether the diff touches the relevant code path
4.  Actionable:  True regressions have prevention guidance; true fixes explain the
causal chain; stochastic cases explicitly state the evidence and implications
## Reflection Quality Checklist
Before finishing, verify each reflection meets these criteria:
1.  Specific:  References exact tool calls, steps, or code paths
2.  Causal:  Explains WHY the outcome occurred, not just WHAT happened
```

(continued)

```
3. Actionable: `-prevention-or-next` gives clear guidance for future iterations -- not
generic advice like "try harder"
4. Distinct: Each reflection adds unique information (no copy-paste templates across
scenarios)
## Constraints
- Record a reflection for every scenario in the mini-batch.
- Use `evo-dag update-reflection` -- do NOT write reflections to files.
- Do NOT modify source code during reflection -- Session 2 is analysis only.
- Do NOT read raw traces from development set output directories -- only aggregate dev
accuracy is available for generalization analysis.
- Do NOT read raw traces from development set dirs (`_val_` or `seed_val_` output dirs)
-- only aggregate accuracy is available.
CRITICAL: You MUST run `evo-dag update-reflection` for every scenario in the mini-batch
before finishing this session. Use individual calls for fixed/regressed/still_failing
scenarios and `-batch-still-passing` for unaffected passing scenarios. Missing
reflections degrade the quality of accumulated lessons for future iterations.
```

Figure 14: Reflection Session Prompt: agent analyzes per-scenario initial vs. re-evaluation outcomes (`fixed` / `regressed` / `still_failing` / `still_passing`), distinguishes true causal effects of the patch from stochastic LLM non-determinism, and records structured reflections that feed back into `evo-dag show history` for future iterations.

Evolution Session Prompt

```
## Mandatory Skills
You MUST read and follow the SKILL.md for each skill listed below. Do NOT skip or
summarize any skill -- execute the full procedure described in each one. These skills
are installed at `.claude/skills/<n>/SKILL.md` in the current worktree.
| Order | Skill | When | Why |
|----|----|---|---|
| 1 | `history-analysis` | Before any other work (Step 1) | Structured analysis of the
full evolution history -- classifies every patch as revert vs. preserve to inform
candidate selection |
| 2 | `patch-verification` | After any codebase change (Step 5) | Verifies the prepared
base codebase has no runtime errors -- a broken base causes every subsequent session to
fail |
> Enforcement: If you make any change to the worktree (rsync, cherry-pick, revert, or
code edit), you MUST run the `patch-verification` skill's full procedure before
finishing. Skipping verification is a critical failure.
## Goal
Prepare the best possible base codebase for this iteration's patch. The optimization
objective is to maximize performance across the task distribution, with development set
(dev) accuracy as the proxy.
Your selection must be data-driven, based on dev scores, regression analysis, and
cross-candidate code comparison -- not on attachment to accumulated patches.
Anti-Sunk-Cost Principle: The number of iterations invested in a lineage is NOT a
reason to continue it. A long lineage with declining dev scores is a signal to switch,
not to persist. Evaluate each candidate by its measured performance and the quality of
its code, not by how many patches led to it.
## Context
- Iteration: {iteration}
- Phase: {phase}
- Current worktree: `{worktree_path}` (fork of C{base_parent_idx})
- Base parent: C{base_parent_idx} (worktree: `{parent_worktree}`)
## Candidate Performance (sorted by dev score)
{candidate_table}
Note: Dev scores are only available for candidates that passed mini-batch acceptance.
"pending" means the candidate was not evaluated on the dev set. Use patch history, raw
traces, and codebase inspection to assess these.
```

(continued)

```
## DAG Topology
{dag_topology}
## Selection Strategy
Choose the base candidate by weighing dev scores, regression history, score trajectory,
and code quality together.  Dev scores are noisy -- small differences may be evaluation
variance, so corroborate with regression counts and code inspection.  Do not stay on a
lineage just because many patches were accumulated; patches only have value if
performance improved.
Options:
- Continue from latest:  When the lineage is near its peak and stable.  Revert any
regressions and cherry-pick fixes from other candidates.
- Switch parent:  When dev score has dropped well below the best candidate and the
trajectory is not recovering.  Cherry-pick verified good patches from the old lineage.
- Combine candidates:  When different candidates have complementary strengths in
non-overlapping areas.
Key rules:
- Regressions identified here must be reverted in THIS session.  Do NOT defer to Session
1 -- Session 1 only addresses the current mini-batch and will not fix prior regressions.
- When cherry-picking, prefer diff-level precision over copying entire files.  Use
‘evo-dag show edge‘ to see exact diffs and apply only the relevant changes.  Whole-file
copies via rsync bring unintended changes.
- Compare top candidates’ codebases directly (‘diff -rq‘) to find cherry-pick
opportunities that aren’t visible from scores alone.
## Workflow
### 1.  Analyze history
MANDATORY first step.  Run the ‘history-analysis‘ skill to build a structured
understanding of the full evolution history.  Do NOT skip this step or truncate ‘evo-dag
show history‘ with ‘head‘/‘tail‘.  The skill provides the methodology for reading the
complete history without truncation.
The purpose of this analysis is to classify every patch in the history into two
categories that directly inform candidate selection:
#### Patches to revert (caused severe regressions)
- Which patches introduced regressions?  Identify the exact iteration, candidate, and
code diff that caused each regression.
- How severe is each regression?  (How many scenarios regressed?  Did dev score drop?)
- Is the regression still present in the current lineage, or was it already fixed by a
later patch?
- What files were modified -- so you know exactly what to revert or undo.
#### Patches to preserve (drove performance gains)
- Which patches produced the largest improvements in dev score?
- Which patches fixed scenarios that stayed fixed in subsequent iterations (durable
fixes vs.  fragile ones)?
- Which patches generalized well beyond their target mini-batch?
- What files and patterns were involved -- so you know what to protect when combining
candidates.
This classification directly drives your selection decision:
- Continue from latest:  Revert the harmful diffs while keeping the beneficial ones.
- Switch parent:  If too many regressions have accumulated, switch to the best candidate
and cherry-pick the preserved patches.
- Combine candidates:  If different candidates have complementary preserved patches,
combine them.
### 2.  Investigate candidates
Use the following sources to decide which candidate(s) to build on:
1.  History analysis output:  The structured summary from Step 1 -- lineage trajectory,
lessons, and cross-candidate strengths.
2.  Specific candidate:  ‘evo-dag show node <idx>‘ -- scores, patch intent, verdict,
output directories.
3.  Raw traces:  Read evaluation output and trace files in the iteration output
directories listed by ‘evo-dag show node‘ or ‘evo-dag show current-batch‘.  See
CLAUDE.md’s Iteration Output Structure section for the file layout.
```

(continued)

4. Candidate codebases: Read source files directly in other candidates' worktrees to understand what changed and whether a fix is worth porting. Worktree paths are shown in the candidate table and ‘evo-dag show node‘.

5. Cross-candidate code comparison (MANDATORY for top candidates): For the top 2-3 candidates by dev score, directly compare their codebases to identify divergent files and cherry-pick opportunities:

    ‘‘‘bash

    diff -rq {session_root}/worktrees/<candidate_A_dir>/
{session_root}/worktrees/<candidate_B_dir>/ | grep -v '.git'

    ‘‘‘

   Then read the divergent files in both worktrees to understand which candidate's version is better for each file. This comparison reveals cherry-pick opportunities that are invisible from score data alone.

3. Revert regressions (MANDATORY if continuing from latest)

If you chose to continue from the latest candidate, you MUST revert all identified regressions in THIS session. Do NOT defer regression fixes to Session 1 -- Session 1 only addresses the current mini-batch and will not fix prior regressions.

For each regression identified in Step 1:

1. Locate the exact diff that caused the regression using ‘evo-dag show edge <parent> <child>‘ for the iteration that introduced it.

2. Compare with pre-regression code: Read the same file in the pre-regression candidate's worktree to see what the code looked like before the harmful change.

3. Revert only the harmful lines: Do not revert the entire patch if it also contains beneficial changes. Surgically revert only the lines that caused the regression.

4. Preserve beneficial changes: If a patch contains both good and bad changes, keep the good parts. Check whether specific scenarios were fixed by specific lines in the diff to determine what to preserve.

After reverting, run the ‘patch-verification‘ skill to ensure no runtime errors were introduced.

4. Prepare the base codebase

Each candidate has a base parent (the candidate it was forked from) and optionally cherry-pick parents (other candidates whose code was referenced, copied, or combined). The base parent is set automatically by the outer loop. Cherry-pick parents are what you record here.

Your worktree starts as a fork of C{base_parent_idx} (base parent).

Switching base entirely

If you decided to switch base, copy the new base's entire codebase:

    ‘‘‘bash

    rsync -a -exclude='.git' {session_root}/worktrees/<candidate_dir>/ ./

    ‘‘‘

Then cherry-pick good patches from the old lineage using the diff-level method below.

Diff-level cherry-pick (preferred over file-level copy)

When porting specific fixes from another candidate, use diff-level precision rather than copying entire files. Copying entire files brings unintended changes from the source candidate.

1. Identify the exact diff: Use ‘evo-dag show edge <parent> <child>‘ to see the exact code changes that constituted the fix you want to port.

2. Compare files: Read the specific file in both the source candidate's worktree and your current worktree. Use ‘diff‘ to see differences:

    ‘‘‘bash

    diff {session_root}/worktrees/<source_candidate_dir>/<file> ./<file>

    ‘‘‘

3. Apply only the relevant changes: Manually apply only the lines from the diff that correspond to the fix. Do not copy unrelated changes that the source candidate may have accumulated.

4. Verify: After each cherry-pick, run the ‘patch-verification‘ skill.

5. Verify changes

If you made any changes to the worktree (rsync, cherry-pick, revert, etc.), you MUST run the ‘patch-verification‘ skill to ensure the codebase has no runtime errors. A broken base codebase will cause every subsequent session to fail -- verification is mandatory.

(continued)

```
### 6.  Record selection
After preparing the base code, record your selection with ‘evo-dag update-selection‘.
List all candidates you referenced -- whether you switched base entirely, cherry-picked
specific files, or just used their code as reference for a re-implementation:
    ‘‘‘bash
    evo-dag update-selection \
      -parent-candidates "<idx1>,<idx2>,..." \
      -reasoning "<what you took from each and why>"
    ‘‘‘
CRITICAL: You MUST run ‘evo-dag update-selection‘ before finishing this session.
Failure to do so will result in lost selection history that future iterations need to
understand the DAG lineage.
## What Happens Next
After this session, the outer loop runs Session 1 (Diagnose + Patch):  the agent
receives the current mini-batch’s failing scenarios with execution traces, diagnoses
root causes, and applies targeted code patches to the worktree you prepared here.
```

Figure 15: Evolution Session Prompt: data-driven base codebase selection from prior candidates, including history analysis, regression reversion, and diff-level cherry-picking.

Q Data Licenses

The licenses for the datasets used in this paper are as follows:

- GAIA2: cc-by-4.0
- SWE-Bench Pro: MIT license
- Terminal-Bench 2.0: Apache-2.0 license

All datasets were used strictly for research purposes and were not utilized in any non-research contexts, particularly for commercial applications.

R Limitations and Future Work

AutoSaddler currently formulates harness optimization as a supervised learning problem, assuming access to a training set of tasks paired with gold answers and a task-level success metric (e.g., pass/fail) that provides a clear signal for each rollout. While this assumption aligns with standard agent benchmarks, real-world deployment settings may lack such ground-truth outcomes, and obtaining them at scale can be prohibitively expensive. A natural extension is therefore to relax this supervision requirement and study unsupervised or weakly supervised harness optimization, in which failure signals are derived from intrinsic cues such as trace-level inconsistencies, tool-call errors, or self-consistency across rollouts rather than external labels. Another direction is to synthesize training data by bootstrapping from agent environments, such as each user's `Universe` in GAIA2. Finally, our current scope is limited to stateless, independent tasks; extending AutoSaddler to stateful settings, integrating memory and skill curation, and validating the approach across a broader set of LLM families remain important directions for future work.

In terms of real-world deployment, we want to stress the importance of strong governance and human-in-the-loop in the early development stage, namely, automated harness modifications should undergo human review or additional security validation before deployment to production systems.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and section 1 explicitly state three core contributions.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The limitations and future work are clearly discussed in Appendix R.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper does not include any theoretical results or proofs.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Appendix B provides comprehensive details on the experimental setup, including the optimization budgets, precise data splits across the two benchmarks (GAIA2 and Terminal-Bench 2.0), structural adaptations made to run baselines (GEPA and Meta-Harness), and the specific evaluation infrastructures and models (e.g., Llama-3.3-70B-Instruct, Docker-based testing) used to run and evaluate the task agents.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We will submit the anonymized code as part of the supplementary material.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Appendix B provides comprehensive details on the experimental setup, including the optimization budgets, precise data splits across the two benchmarks (GAIA2 and Terminal-Bench 2.0), structural adaptations made to run baselines (GEPA and Meta-Harness), and the specific evaluation infrastructures and models (e.g., Llama-3.3-70B-Instruct, Docker-based testing) used to run and evaluate the task agents.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: As detailed in Section 5.2, the main evaluations across the benchmarks were performed over three independent runs. To capture the variance across these overall runs, the results reported in Tables 2 and 3 present the mean Pass@1 success rates along with their corresponding standard deviations.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: As detailed in Section 5.2 and Appendix H, the paper explicitly analyzes the computational cost required for the optimization process. Specifically, Figures 1 and 5 report the compute efficiency by quantifying the exact number of "task executions" and "leveraged traces" needed for the proposed method and baselines to reach their peak dev accuracies across the benchmarks. This detailed breakdown of task executions serve as a direct proxy for the inference costs and computational resources required to reproduce the optimization trajectories.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The study adheres to the NeurIPS Code of Ethics and does not involve any sensitive or human-related data.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [N/A]

Justification: The work focuses on the harness optimization and does not involve direct societal applications; hence, broader societal impacts are not discussed.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: Since the work does not involve releasing models or data with high misuse potential, no specific safeguards are necessary.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: We listed the licenses and terms of use for all external datasets in Appendix Q.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.

- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The optimization harness introduced in this paper is a new asset. In Appendix N and Appendix O, we present representative patches applied to the base harnesses of GAIA2 and Terminal-Bench 2.0, respectively. We will also provide full documentation in the supplementary material.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This paper does not involve any crowdsourcing or research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: The research presented in this paper does not involve human subjects or study participants. Therefore, IRB approval, risk disclosure, and related ethical reviews are not applicable.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: As detailed in our methodology Section 4, LLMs are a core, fundamental component of the proposed AutoSaddler framework. Specifically, LLMs are utilized to conduct in-depth diagnosis of agent failure traces and to generate structured code patches for harness optimization. Additionally, all task agents evaluated across our benchmark experiments are intrinsically LLM-driven, with specific method details provided in Appendix B.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.