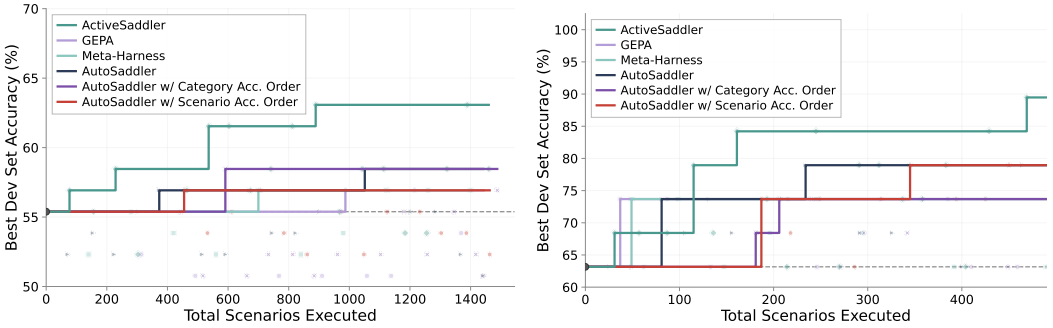
ACTIVESADDLER: AUTOMATED CURRICULUM LEARNING FOR AGENT HARNESS OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Automated harness optimization can substantially improve LLM agents by iteratively updating their prompts, tool interfaces, and control logic from execution feedback. However, existing methods primarily optimize *how* the harness is updated while largely fixing *which training scenarios* generate the feedback that drives those updates. As the harness evolves, the scenarios most useful for further optimization can change, suggesting that the training curriculum itself should adapt alongside the harness. We formulate this missing dimension of harness optimization as an automated curriculum learning problem and introduce **ActiveSaddler**. ActiveSaddler models the evolving curriculum as a non-stationary bandit with dynamically instantiated optimization targets. It abstracts recurring failures into reusable failure-pattern arms, estimates the potential learning progress from further targeting each pattern, and adaptively balances revisiting known weaknesses with exploring unseen scenarios for new ones. Optimization outcomes continually update both the set of discovered failure patterns and their priorities, allowing the curriculum to co-evolve with the harness. Experiments on GAIA2 and Terminal-Bench 2.0 show that ActiveSaddler consistently discovers stronger harnesses, improving test Pass@1 by **4.4** and **7.5** percentage points over the same harness optimizer using a scenario order fixed before optimization, respectively. Ablations further show that these gains depend on dynamically constructing optimization targets, estimating their evolving utility, and balancing continued optimization with new failure discovery. Together, these results establish automated curriculum learning as a new crucial optimization dimension for harness optimization.



(a) GAIA2. (b) Terminal-Bench 2.0.

Figure 1: **Optimization efficiency across benchmarks.** Best-so-far development accuracy versus cumulative task-agent rollouts for ActiveSaddler and baseline methods.

1 INTRODUCTION

Large Language Models (LLMs) have become increasingly capable of powering autonomous agents, yet reliable performance on multi-step and long-horizon tasks still depends heavily on the external *harness* surrounding the model. A harness specifies how the agent is prompted, which tools it can use, and how its execution is controlled, monitored, and corrected. While well-designed harnesses can substantially improve agent robustness (Zhou et al., 2026; Anthropic, 2025; OpenAI, 2026;

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

LangChain, 2026), their effectiveness can degrade as models, task distributions, or deployment environments change. This has motivated automated harness optimization methods that execute training scenarios, diagnose deficiencies from the resulting execution traces, and patch prompts, tool interfaces, or runtime control logic (Lee et al., 2026; Park et al., 2026).

These methods thus focus primarily on *how* to update the harness, whereas the training scenarios used to generate feedback are usually predetermined, either as the full training set or as mini-batches scheduled before optimization begins. As the harness evolves, however, the usefulness of these training scenarios can change, creating two limitations. First, unresolved failures may benefit from further optimization, whereas repaired ones may require little additional attention. A fixed schedule cannot respond to this shift and may therefore move on from unresolved failures too early or spend rollouts on scenarios with little remaining value. Indeed, under fixed-order optimization, many failures observed during training remain unresolved by the final optimized harness (Figure 3b).

Second, the relative value of revisiting known failures versus evaluating unseen scenarios also changes over optimization. Revisiting is more valuable when actionable unresolved failures remain, whereas exploring unseen scenarios becomes more valuable when few worthwhile targets remain among the observed failures. A fixed schedule cannot adapt this balance. These limitations are particularly costly under a limited rollout budget, where optimization opportunities must be allocated selectively. This perspective is consistent with prior work on curriculum and environment design, which shows that learning can benefit from adapting *what* data, tasks, or environments are presented to the learner (Clune, 2019; Wang et al., 2019; Graves et al., 2017; Matiisen et al., 2017). This suggests a complementary dimension for harness optimization: as the harness evolves, the curriculum can also adapt *which training scenarios* are used to drive subsequent updates.

In this work, we formulate scenario selection for budgeted offline harness optimization as an automated curriculum-learning problem and introduce **ActiveSaddler**, which *proactively* adapts the optimization curriculum as the harness evolves. We model curriculum selection as a non-stationary bandit whose arms are instantiated online from observed failures: each *failure-pattern arm* represents a potentially repairable harness deficiency shared across failed trajectories. At each iteration, ActiveSaddler either evaluates unseen scenarios to discover new failure patterns or revisits an existing arm, prioritizing arms by their estimated potential for further improvement. The resulting optimization outcomes update both the priorities of existing arms and the set of discovered patterns. Through this feedback loop, ActiveSaddler co-evolves the curriculum with the harness to allocate a fixed rollout budget toward more valuable optimization opportunities.

We evaluate ActiveSaddler on GAIA2 (Froger et al., 2026) and Terminal-Bench 2.0 (Merrill et al., 2026) against existing harness optimizers and fixed curriculum baselines. Across both benchmarks, ActiveSaddler achieves the strongest final harness performance, improving over the same harness optimizer with a scenario order fixed before optimization by **4.4** and **7.5** percentage points, respectively. Our ablation analyses reveal three design principles for automated curriculum learning in harness optimization: i) **failure-pattern arms** provide an effective optimization abstraction: category-level arms can conflate distinct failures, whereas scenario-level arms fragment recurring failures across individual scenarios; ii) **adaptive arm prioritization** is crucial because the value of revisiting a failure changes as the harness evolves; iii) **adaptive exploration** is necessary because the relative value of revisiting observed failures versus evaluating unseen scenarios changes over optimization.

Our main contributions are summarized as follows:

- We formulate scenario selection for offline harness optimization as an automated curriculum-learning problem, where training scenarios are selected adaptively as the harness evolves.
- We introduce ActiveSaddler, which combines failure-pattern arms, adaptive arm prioritization, and adaptive exploration to construct this evolving curriculum.
- We show consistent gains on GAIA2 and Terminal-Bench 2.0 over existing harness optimizers and fixed curricula under the same rollout budget.

2 Related Work

Automatic Harness Optimization. Recent work extends automatic optimization beyond prompts to the broader agent *harness*, including tools, memory, skills, and runtime control logic (Zhou et al.,

2026). Despite implementation differences, these methods generally execute the current harness, use rollout feedback to propose updates, and evaluate candidate harnesses. Broadly, automatic harness optimization (AHO) spans search-based and learning-based approaches. Search-based methods explore alternative harness implementations or configurations through evolutionary or Bayesian search (Lee et al., 2026; Lin et al., 2026; Zhang et al., 2026; Sengupta & Wang, 2026). Learning-based methods instead treat external harness state as the object being learned, repeatedly updating it from rollout feedback on training mini-batches, analogous to model training but without updating model weights (Park et al., 2026; Yang et al., 2026b). GEPA (Agrawal et al., 2026) lies between these styles, coupling reflective updates from training mini-batches with evolutionary Pareto search over validation performance. Recent work further extends AHO by changing the optimization signal, such as recalibrating historical experience or replacing labeled validation with self-preference (Guo et al., 2026; Pan et al., 2026), and by jointly optimizing harnesses and model weights (Hebbar et al., 2026; Chen et al., 2026; Kim et al., 2026). Orthogonally, offline approaches optimize the harness before deployment, often using training data and held-out validation (Park et al., 2026; Yang et al., 2026b), whereas online or test-time approaches continue adapting during sequential interaction or evaluation (Karten et al., 2026; Wei et al., 2026; Nie et al., 2026). We focus on offline, learning-based AHO under a standard train/dev/test protocol, where ActiveSaddler leaves the harness-update mechanism unchanged and instead adapts *which training scenarios* provide evidence for each update.

Automated Curriculum Learning. Curriculum learning studies how training data, tasks, or environments are scheduled to improve learning (Bengio et al., 2009). Early work prioritized learning opportunities according to competence or learning progress (Oudeyer et al., 2007; Lopes & Oudeyer, 2012), while subsequent automated curricula formalized this choice through non-stationary bandits, teacher-student task selection, and progress-driven environment sampling (Graves et al., 2017; Matiisen et al., 2017; Portelas et al., 2019). These methods estimate task utility from observable learner-side signals, such as loss reduction, competence progress, or TD error (Graves et al., 2017; Matiisen et al., 2017; Jiang et al., 2021). ActiveSaddler adopts the same principle that training opportunities should be prioritized according to their evolving value to the learner. Harness optimization, however, makes both the curriculum target and its progress implicit. A training scenario does not specify a fixed harness weakness: the weakness becomes apparent only after execution, the same scenario may expose different failures as the harness changes, and the same weakness may recur across different scenarios. Moreover, a diagnosed weakness has no direct loss or other target-level learning signal indicating whether it remains unresolved or worth another intervention; this must instead be inferred from the current harness, prior optimization attempts, and execution history. ActiveSaddler therefore constructs failure-pattern arms from diagnosed failures, continually re-estimates their optimization value, and allocates rollouts between revisiting known weaknesses and exploring unseen scenarios for new ones.

Adaptive Task Selection for Agent Optimization. Recent work has begun to adapt task allocation at different stages of agent optimization. For model-weight optimization, CoEvolve (Yang et al., 2026a) uses forgetting and uncertainty observed in agent rollouts to generate new training tasks targeting behaviors the current agent still struggles with. In harness optimization, Task-CoEvolve (Miyai et al., 2026) adaptively selects validation tasks that best distinguish candidate harnesses, while HarnessLens (Xu et al., 2026) selects verification tasks relevant to each proposed harness modification. These methods adapt task selection on the evaluation side of harness optimization, after candidate modifications have been proposed. In contrast, ActiveSaddler adapts the training scenarios that generate evidence for harness updates throughout optimization, repeatedly deciding which known weaknesses to revisit and when to execute unseen scenarios to discover new ones.

3 PRELIMINARIES

Harness Optimization. Let H_θ denote an agent harness parameterized by $\theta \in \Theta$, where θ may include prompts, tool interfaces, and runtime control logic. For a task scenario (x, y^*) drawn from a task distribution $\mathcal{T}$, where x denotes the task input and y^* its reference answer, executing the agent with H_θ produces an execution trace τ and a final answer $\hat{y}$ according to $(\tau, \hat{y}) \sim P_\theta(\cdot \mid x)$, where P_θ denotes the stochastic execution distribution induced by the model and harness. Let $\mu(\hat{y}, y^*)$ be a task-level evaluation metric. The population performance of a harness is

$$J(\theta) = \mathbb{E}_{(x,y^*)\sim\mathcal{T}}\mathbb{E}_{(\tau,\hat{y})\sim P_\theta(\cdot|x)}\big[\mu(\hat{y}, y^*)\big]. \tag{1}$$

For a finite dataset D, we define the empirical harness performance as

$$\widehat{J}_D(\theta) = \frac{1}{|D|} \sum_{(x_i, y_i^*) \in D} \mu(\hat{y}_i, y_i^*), \qquad (\tau_i, \hat{y}_i) \sim P_\theta(\cdot \mid x_i). \tag{2}$$

Following the learning-based offline harness-optimization setting of AutoSaddler (Park et al., 2026), we assume fixed and mutually disjoint training, development, and test sets, D_{train}, D_{dev}, and D_{test}. At iteration t, let θ_t denote the current harness parameters and write $H_t := H_{\theta_t}$. Given a selected set of training scenarios $X_t \subseteq D_{\text{train}}$, executing H_t produces a collection of execution records

$$\mathcal{E}_t = \{(x, y^*, \tau, \hat{y}) \mid (x, y^*) \in X_t,\ (\tau, \hat{y}) \sim P_{\theta_t}(\cdot \mid x)\}. \tag{3}$$

A harness optimizer $\mathcal{O}$ uses these records to diagnose deficiencies in the current harness, construct and evaluate candidate updates, and return the harness retained for the next iteration:

$$\theta_{t+1} \sim \mathcal{O}\big(\theta_t, \mathcal{E}_t, D_{\text{dev}}\big), \tag{4}$$

where D_{dev} supports candidate selection, regression checks, and generalization checks during optimization, while D_{test} is reserved for final evaluation. Thus, X_t determines *which evidence* is collected, while $\mathcal{O}$ determines *how it is used* to update the harness.

Automated Curriculum Learning for Harness Optimization. Automated curriculum learning adapts the selection of training data to the evolving state of a learner (Graves et al., 2017). In harness optimization, we regard the current harness as the evolving learner and formulate the selection of X_t as a curriculum-learning problem. Let $\mathcal{I}_{<t}$ denote the information available before the curriculum decision at iteration t, including the current harness and the execution and update history accumulated over previous iterations. A curriculum policy π selects the next training-scenario set according to

$$X_t \sim \pi(\cdot \mid \mathcal{I}_{<t}), \qquad X_t \subseteq D_{\text{train}}. \tag{5}$$

The selected scenarios are executed under H_t, and the resulting harness-optimization step provides feedback for subsequent curriculum decisions. We hold the optimizer $\mathcal{O}$ and its candidate-evaluation procedure fixed and study how the curriculum policy adapts as the harness evolves.

Let $\hat{\theta}_\pi$ denote the final harness selected by development-set performance from the optimization trajectory induced by policy π. Under a fixed rollout budget shared across curriculum policies, the automated curriculum-learning objective is

$$\pi^* \in \arg\max_{\pi} \mathbb{E}\Big[J(\hat{\theta}_\pi)\Big]. \tag{6}$$

After optimization, we report the performance of the selected harness on the test set as $\widehat{J}_{D_{\text{test}}}(\hat{\theta}_\pi)$.

4 Proposed Method: ActiveSaddler

We now present ActiveSaddler, an iterative curriculum-learning framework for harness optimization. We first introduce the problem formulation and overall workflow, then describe key modules in detail.

Problem Formulation and Design Rationale. Harness optimization operates under a limited rollout budget, requiring the curriculum to decide at each iteration which training target should receive the next optimization opportunity. This naturally leads to a sequential resource-allocation problem, which we formulate as a *multi-armed bandit*: each arm represents an optimization target, and selecting an arm directs the next round of optimization toward that target.

To instantiate these optimization targets, we define each arm around a *harness weakness* rather than an individual failed execution, since a patch typically addresses an underlying deficiency that may recur across multiple scenarios and traces. Because such weaknesses are not directly observable, we infer them through failure diagnosis and group failures attributed to the same deficiency into a common *failure-pattern arm*. This lets related failures contribute evidence toward a shared optimization target.

This formulation introduces two practical challenges. First, the arm set is not known in advance: as the harness evolves and executions vary stochastically, new failure patterns may emerge. We therefore

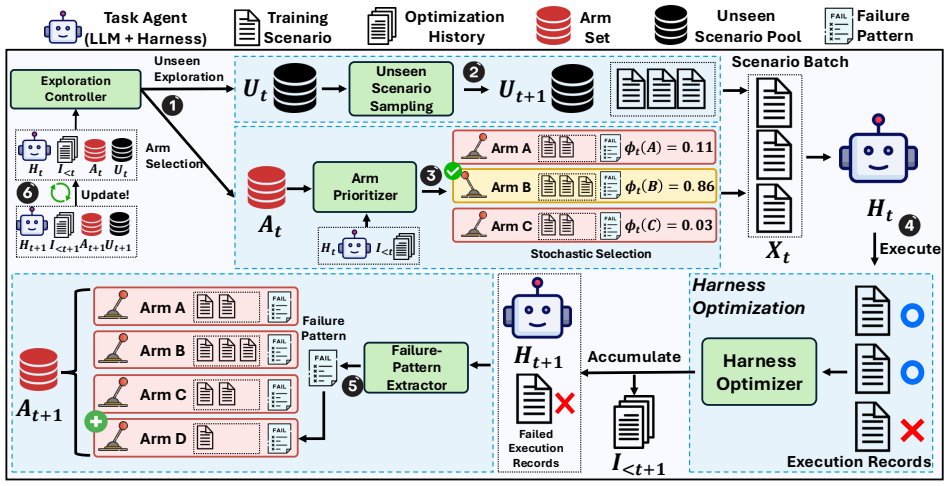


Figure 2: **Overview of ActiveSaddler**. At each iteration, the curriculum chooses between exploring unseen scenarios and revisiting an instantiated arm, optimizes the harness on the selected scenario batch, and updates its state based on the resulting execution and optimization outcomes.

construct the arm set during optimization and expand it as new failure patterns are diagnosed. Second, each iteration typically executes only a subset of the training set (Park et al., 2026; Agrawal et al., 2026), leaving some scenarios and their potential failure patterns unobserved. The curriculum must therefore balance *exploitation*, by prioritizing among known failure-pattern arms, with *exploration*, by executing unobserved scenarios that may reveal new weaknesses.

These challenges motivate the three core components of ActiveSaddler: a **Failure-Pattern Extractor**, which identifies recurring harness weaknesses and organizes them into failure-pattern arms; an **Arm Prioritizer**, which selects among known arms for further optimization; and an **Exploration Controller**, which decides when to explore unseen scenarios that may reveal new weaknesses. We next describe how these components interact within the iterative harness-optimization loop.

Workflow Overview. As illustrated in Figure 2, each iteration t begins with the current harness H_t, accumulated optimization history $\mathcal{I}_{<t}$, instantiated arm set $\mathcal{A}_t$, and unseen scenario pool U_t. The Exploration Controller first decides whether the next optimization step should explore previously unseen scenarios or continue optimizing a known failure-pattern arm (❶). If exploration is selected, ActiveSaddler constructs the next scenario batch X_t from U_t (❷); otherwise, the Arm Prioritizer selects an arm $a_t \in \mathcal{A}_t$, and X_t is constructed from the scenarios associated with that arm (❸).

The selected scenarios are then executed under H_t, and the resulting execution records $\mathcal{E}_t$ are passed to the harness optimizer $\mathcal{O}$ to produce the updated harness H_{t+1} (❹). From failed executions, the Failure-Pattern Extractor identifies recurring harness weaknesses, matches them to existing failure-pattern arms or instantiates new ones, and updates the arm set for subsequent iterations (❺). The resulting H_{t+1}, optimization history $\mathcal{I}_{<t+1}$, arm set $\mathcal{A}_{t+1}$, and unseen scenario pool U_{t+1} define the state for the next curriculum decision (❻). In this way, the training curriculum co-evolves with the harness as its unresolved weaknesses change over optimization. The full ActiveSaddler algorithm and its instantiation with AutoSaddler are in Appendix B.

Failure-Pattern Extraction. The Failure-Pattern Extractor converts failure evidence produced during each optimization step into reusable failure-pattern arms. Let $\mathcal{F}_t$ denote the failed execution records made available by the underlying harness optimizer at iteration t, and let $\delta_t(e)$ denote the associated diagnostic evidence for each $e \in \mathcal{F}_t$. The exact composition of $\mathcal{F}_t$ depends on the optimizer. In our AutoSaddler instantiation, it includes both failures diagnosed before patching and failures that remain or newly emerge during post-patch evaluation.

The extractor operates in two stages: *symptom extraction* and *pattern normalization*. It first independently abstracts each failure into a candidate symptom, $c_t(e) = g_{\text{ext}}(e, \delta_t(e))$, without consulting the

existing arm set. After all candidates are extracted, they are normalized against the current arms:

$$\mathcal{A}_{t+1} = g_{\text{norm}}(\mathcal{A}_t, \{c_t(e)\}_{e \in \mathcal{F}_t}), \tag{7}$$

where g_{norm} links each candidate to an existing atomic pattern, composes multiple existing patterns when needed, or instantiates a new arm for a previously unseen failure pattern.

Each arm $a \in \mathcal{A}_{t+1}$ maintains a pattern description, a supporting scenario set $S_{t+1}(a) \subseteq D_{\text{train}}$, and the associated harness, execution, and diagnostic evidence under which the pattern was observed. When an arm is selected, its supporting scenarios form the pool for the next optimization batch, subject to the per-iteration execution limit. Detailed extraction instructions are in Appendix K.

Arm Prioritization. When the Exploration Controller chooses to continue optimization over the instantiated arm set, the Arm Prioritizer determines which $a \in \mathcal{A}_t$ should receive the next optimization attempt. Ideally, this decision would be based on the expected population-performance change induced by selecting arm a. However, this population objective is not directly observable during optimization; we therefore use an LLM to estimate a learning-progress score

$$\phi_t(a) = g_{\text{select}}(a, \mathcal{I}_{<t}) \in [0, 1], \tag{8}$$

where g_{select} denotes the LLM-based arm-scoring procedure and a larger $\phi_t(a)$ indicates greater estimated benefit from allocating the next optimization step to arm a. To estimate this score, the LLM is instructed to jointly consider four factors: whether the failure pattern remains active under the current harness (*severity*), whether it appears addressable through a harness intervention (*fixability*), how broadly a successful repair may generalize (*breadth*), and the risk that such an intervention introduces regressions (*side-effect risk*). The scores are recomputed from the latest $\mathcal{I}_{<t}$ at each arm-selection step, allowing arm priorities to adapt as the harness evolves.

The resulting scores define a stochastic selection distribution over the current arm set:

$$q_t(a) = \frac{\exp(\phi_t(a)/\tau_{\text{sel}})}{\sum_{a' \in \mathcal{A}_t} \exp(\phi_t(a')/\tau_{\text{sel}})}, \qquad a \in \mathcal{A}_t, \tag{9}$$

where $\tau_{\text{sel}} > 0$ controls the stochasticity of arm selection. The curriculum then samples

$$a_t \sim \text{Categorical}(q_t) \tag{10}$$

and constructs the next scenario batch from the selected arm, $X_t \subseteq S_t(a_t)$. This stochastic selection rule favors arms with greater estimated learning progress while preserving the possibility of revisiting lower-scored arms as their utility changes over optimization. Detailed scoring instructions and definitions of the four factors are in Appendix K.

Exploration Control. The Exploration Controller determines whether the next optimization step should exploit a previously discovered failure-pattern arm or explore unseen scenarios that may reveal new failure patterns. Given the current harness H_t, instantiated arm set $\mathcal{A}_t$, unseen scenario pool U_t, and optimization history $\mathcal{I}_{<t}$, we instantiate an agent session for a binary decision:

$$d_t = g_{\text{explore}}(H_t, \mathcal{A}_t, U_t, \mathcal{I}_{<t}) \in \{\texttt{unseen}, \texttt{arm}\}. \tag{11}$$

The policy makes this decision based on the current optimization context, weighing whether the known arms contain a worthwhile unresolved optimization target against exploring the remaining unseen scenarios for previously undiscovered weaknesses. If $d_t = \texttt{unseen}$, i.e., a DRAW decision, ActiveSaddler samples $X_t \subseteq U_t$ without replacement and removes the selected scenarios from the unseen pool. If $d_t = \texttt{arm}$, i.e., a PULL decision, Arm Prioritizer selects $a_t \in \mathcal{A}_t$, and the next scenario batch is constructed as $X_t \subseteq S_t(a_t)$. When $\mathcal{A}_t = \varnothing$ and $U_t \neq \varnothing$, the policy selects `unseen`; when $U_t = \varnothing$ and $\mathcal{A}_t \neq \varnothing$, it selects `arm`. Detailed decision criteria and instructions are in Appendix K.

To complement the regression and generalization checks of the underlying harness optimizer, ActiveSaddler also guards against regressions on previously successful training scenarios. Once all offline training scenarios have been explored and every arm has since been visited, prior successes that did not instantiate an arm become eligible for exploration again, analogous to additional epochs over seen examples in conventional offline learning. If such a scenario now fails, the Failure-Pattern Extractor processes it normally, enabling regression rediscovery without re-evaluating all prior successes after every harness update.

Table 1: Test Pass@1 on GAIA2 and Terminal-Bench 2.0 (mean ± std. over three test-time executions). Parentheses show task counts; bold indicates the best result.

GAIA2 (Pass@1)

Harness (Type)	Test-Split (300)
Default Agent (manual)	53.6 ± 1.1
GEPA	54.2 ± 2.2
Meta-Harness	54.2 ± 1.2
AutoSaddler	55.4 ± 1.2
AutoSaddler w/ Category Acc. Order	55.9 ± 1.3
AutoSaddler w/ Scenario Acc. Order	55.7 ± 1.2
ActiveSaddler	**59.8 ± 1.0**
w/o Failure Pattern Based Arm (Category Arm)	56.8 ± 0.8
w/o Failure Pattern Based Arm (Scenario Arm)	56.2 ± 0.2
w/o Arm Prioritizer	55.3 ± 2.1
w/o Exploration Controller	55.3 ± 0.9

Terminal-Bench 2.0 (Pass@1)

Harness (Type)	Test-Split (40)
Terminus 2 (manual)	64.2 ± 2.9
Terminus-KIRA (manual)	69.2 ± 3.8
GEPA	65.8 ± 5.2
Meta-Harness	66.7 ± 5.2
AutoSaddler	72.5 ± 0.0
AutoSaddler w/ Category Acc. Order	70.8 ± 1.4
AutoSaddler w/ Scenario Acc. Order	73.3 ± 1.4
ActiveSaddler	**80.0 ± 2.5**
w/o Failure Pattern Based Arm (Category Arm)	69.2 ± 2.9
w/o Failure Pattern Based Arm (Scenario Arm)	73.3 ± 3.8
w/o Arm Prioritizer	72.5 ± 2.5
w/o Exploration Controller	71.7 ± 1.4

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Benchmarks and Base Harness Systems. We evaluate ActiveSaddler on two diverse benchmarks: GAIA2 (Froger et al., 2026) and Terminal-Bench 2.0 (TB2) (Merrill et al., 2026). GAIA2 evaluates general-purpose assistant capabilities in a simulated smartphone environment spanning 10 distinct user environments, referred to as *Universes*; we use its default ReAct-based agent as the base harness. TB2 comprises 89 realistic tasks across domains including system administration, machine learning, and cybersecurity, and introduces Terminus 2, which we adopt as the base harness.

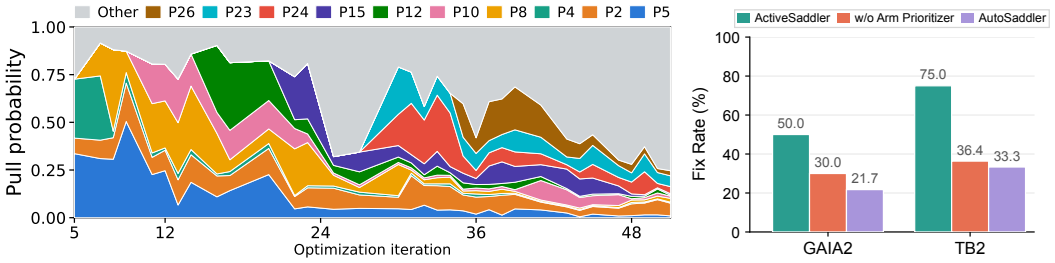
Implementation and Baseline Methods. We instantiate ActiveSaddler on AutoSaddler (Park et al., 2026) by adding three curriculum components: the *Failure-Pattern Extractor*, *Arm Prioritizer*, and *Exploration Controller* (Algorithm 2). These components use the GitHub Copilot SDK (GC-SDK) (GitHub, 2026) and a shared pattern registry for dynamically instantiated failure-pattern arms; agent instructions and registry details are provided in Appendix K and Appendix A. We compare ActiveSaddler against GEPA (Agrawal et al., 2026), Meta-Harness (Lee et al., 2026), and AutoSaddler (Park et al., 2026), which optimize prompts or harnesses without an explicit curriculum. We also construct fixed easy-to-hard curricula (Bengio et al., 2009; Zaremba & Sutskever, 2014; Wu & Tian, 2017) by ranking categories or scenarios by empirical accuracy from five evaluations of the base harness on the training set. For reference, we additionally report Terminus-KIRA (KRAFTON AI & Ludo Robotics, 2026), a manually engineered TB2 harness.

Experimental Protocol and Data Splits. Unless otherwise noted, both the task agent and optimizer use `gpt-5.5`, with reasoning effort set to `medium` for the task agent and `xhigh` for GC-SDK optimizer sessions. Following AutoSaddler (Park et al., 2026), we use out-of-distribution train/dev/test splits with disjoint task groups to evaluate harness generalization. Detailed train/dev/test splits and hyperparameters are provided in Appendix A. We evaluate each optimized harness over three test-time executions and report mean *Pass@1* with standard deviation.

5.2 MAIN RESULTS

As shown in Table 1, ActiveSaddler achieves the strongest test performance on both GAIA2 and TB2, reaching 59.8% and 80.0% Pass@1, respectively. Under the same AutoSaddler optimizer, ActiveSaddler improves over a fixed randomly shuffled scenario order by **+4.4 pp** on GAIA2 and **+7.5 pp** on TB2. It also outperforms difficulty-based fixed curricula by **+3.9 pp** and **+4.1 pp** over category- and scenario-level ordering on GAIA2, and by **+9.2 pp** and **+6.7 pp** on TB2, respectively. Together, these results show that adapting scenario selection during optimization improves harness quality over fixed curricula under the same optimizer.

We further examine these gains through an independent optimization run, a different offline optimizer, and alternative curriculum strategies. In an independent GAIA2 run, ActiveSaddler again outperforms all fixed-order baselines, showing robustness to optimization stochasticity (Appendix C). Applying ActiveSaddler to GEPA improves its average Pass@1 from 54.2% to 57.2% (**+3.0 pp**), showing that the benefit of adaptive training-scenario selection is not limited to AutoSaddler (Appendix E). We



(a) Evolution of pull-probability mass (GAIA2). (b) Failure-to-success conversion rate.

Figure 3: **Effect of adaptive arm scoring.** Adaptive scoring dynamically reallocates optimization priority and yields more effective repair of observed failures.

also compare against non-LLM alternatives: EMA-based failure-persistence scoring and count-based UCB-AIR achieve results **3.1 pp** and **4.2 pp** below ActiveSaddler, respectively (Appendix D).

As shown in Figure 1, ActiveSaddler achieves stronger development performance with fewer task-agent rollouts on both benchmarks. We further analyze whether this efficiency persists after accounting for optimizer-side LLM overhead by combining it with task-agent execution cost. Although ActiveSaddler incurs additional optimizer overhead, it achieves a substantially better end-to-end cost–accuracy trade-off on both benchmarks. On GAIA2, ActiveSaddler reaches 58.5% development accuracy at \$298, whereas AutoSaddler and category ordering require \$1,360 and \$673, respectively, to reach the same accuracy, and scenario ordering reaches only 56.9% despite spending \$1,698. On TB2, ActiveSaddler reaches 78.9% at \$128, while AutoSaddler and scenario ordering require \$220 and \$363, respectively, to reach the same accuracy, and category ordering reaches only 73.7% despite spending \$548. Thus, the additional cost of curriculum adaptation is offset by more effective allocation of expensive task-agent rollouts; detailed cost analyses are provided in Appendix F.

5.3 Ablation Studies and Analysis

RQ1: How do failure-pattern arms enable optimization to track and retarget weaknesses as the harness evolves? We first examine how the choice of arm representation affects optimization. Specifically, we replace the failure-pattern arms in ActiveSaddler with category- or scenario-level arms while keeping the curriculum procedure unchanged. As shown in Table 1, category and scenario arms reduce test Pass@1 by **3.0 pp** and **3.6 pp** on GAIA2, and by **10.8 pp** and **6.7 pp** on TB2, respectively. These results show that arm representation is critical to harness quality.

Analyses in Appendix G further support the benefit of failure-pattern arms. The case studies show that failure-pattern arms preserve unresolved weaknesses as optimization targets, allowing them to be revisited and unsuccessful patches to be iteratively refined. In contrast, category arms are too coarse, grouping distinct weaknesses together and allowing successes on some scenarios to mask unresolved weaknesses in others. Scenario arms are too fine-grained, separating the same weakness across individual scenarios and making it harder to revisit under a limited budget. Consistent with these cases, failure-pattern arms more frequently sample scenarios on which the current harness still fails, exceeding category and scenario arms by **16.5 pp** and **35.3 pp** on GAIA2, and by **29.9 pp** and **23.8 pp** on TB2, respectively. The discovered failure pattern arms are listed in Appendix H.

RQ2: How does adaptive arm scoring prioritize weaknesses as the harness evolves? We examine whether prioritizing failure-pattern arms by their estimated learning potential improves optimization. Specifically, we compare ActiveSaddler with a *w/o Arm Prioritizer* variant that assigns the same score to every arm while keeping the remaining curriculum procedure unchanged. As shown in Table 1, removing adaptive arm scoring reduces test Pass@1 by **4.5 pp** on GAIA2 and **7.5 pp** on TB2, showing the value of prioritizing by learning potential under limited rollouts.

To understand why adaptive arm scoring improves optimization, Figure 3 examines both how optimization priorities evolve and how effectively observed failures are repaired. As shown in (a), pull-probability mass continually shifts across failure-pattern arms rather than remaining concentrated on a fixed subset, indicating that the curriculum reallocates optimization budget as the harness and

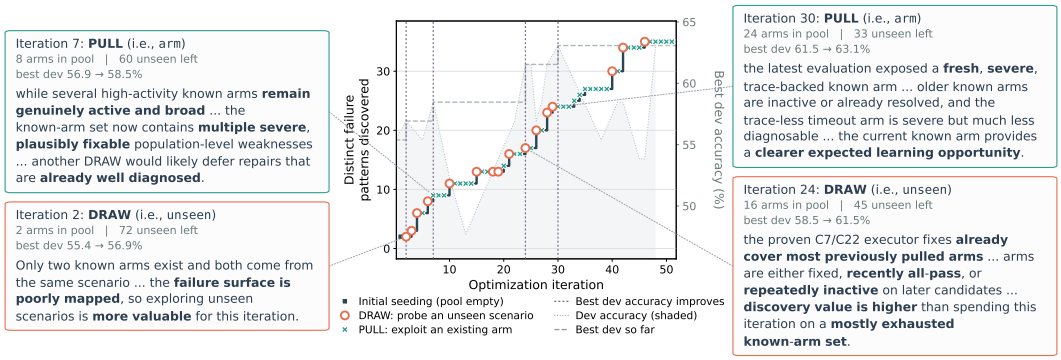


Figure 4: **Adaptive exploration over the optimization trajectory.** The policy chooses DRAW when discovery is more valuable and PULL when promising repair opportunities remain.

accumulated evidence evolve. In (b), we measure the fraction of scenarios observed to fail during optimization that are subsequently converted to success by the optimized harness. ActiveSaddler achieves the highest conversion rate, followed by *w/o Arm Prioritizer* and AutoSaddler. Further analyses in Appendix I provide representative case studies of resolved and recurring weaknesses, examine probability redistribution across the full arm pool, and analyze stochastic selection as new arms are discovered. Together, these results show that adaptive arm scoring improves optimization by reprioritizing weaknesses as their learning potential changes.

RQ3: How does adaptive exploration balance the continued repair of known weaknesses with the discovery of new ones? We examine whether adapting the timing of unseen-scenario exploration improves optimization over a fixed schedule. Specifically, we compare ActiveSaddler with a *w/o Exploration Controller* variant that explores unseen scenarios every five iterations while keeping the remaining curriculum procedure unchanged. As shown in Table 1, replacing adaptive exploration with this fixed schedule reduces test Pass@1 by **4.5 pp** on GAIA2 and **8.3 pp** on TB2. These results show that when to explore is critical under a limited optimization budget.

Figure 4 illustrates how exploration decisions adapt to the evolving optimization state on GAIA2. Representative decisions show that the policy selects DRAW when known arms offer limited remaining learning value and PULL when actionable weaknesses remain. This behavior also appears globally: the DRAW rate decreases from 44% in the first half, with relatively few instantiated arms, to 20% in the second half, with a larger arm pool. To quantify this behavior, we compare how many arms remain unresolved according to their most recent observation at PULL and DRAW decisions. As detailed in Appendix J, ActiveSaddler has 2.0× and 6.9× as many unresolved arms at PULL as at DRAW decisions on GAIA2 and TB2, respectively. In contrast, the fixed schedule yields ratios close to one (0.8× and 1.3×), indicating that its PULL and DRAW decisions are largely insensitive to how many unresolved arms remain. Together, these results show that adaptive exploration favors repair when worthwhile unresolved weaknesses remain and discovery when few such weaknesses remain among the known arms.

6 CONCLUSION

We introduced ActiveSaddler, an automated curriculum-learning approach for harness optimization that adapts which training scenarios generate the execution feedback used for each harness update. Rather than fixing scenario selection before optimization, ActiveSaddler formulates curriculum construction as a non-stationary bandit that dynamically instantiates failure-pattern arms, prioritizes them by their evolving learning potential, and balances revisiting known weaknesses with exploring unseen scenarios. Across GAIA2 and Terminal-Bench 2.0, ActiveSaddler improves test Pass@1 by **4.4** and **7.5** percentage points over the same harness optimizer with a fixed scenario order, respectively, while also outperforming existing harness optimizers and fixed curricula. Our ablations further show that these gains rely on failure-pattern arms, adaptive arm prioritization, and adaptive exploration. Overall, these results establish automated curriculum learning as a complementary dimension of harness optimization: performance depends not only on how feedback is used for updates, but also on which training scenarios generate that feedback.

AI USE STATEMENT

We used generative AI tools for writing assistance, including revising sentences and paragraphs to improve clarity, readability, and overall presentation. We also used generative AI tools for retrieval and discovery, including identifying potentially relevant related work and supporting literature searches. All AI-assisted content was carefully reviewed and verified by the authors, including the wording of the manuscript, technical claims, numerical results, and citations. The authors take full responsibility for the final manuscript.

REPRODUCIBILITY STATEMENT

We provide the implementation details, prompts, data splits, and evaluation protocol needed to reproduce our experiments. Section 5.1 specifies the benchmark and base-harness configurations, baseline methods, model settings, construction of the fixed curricula, and test-time evaluation protocol. Appendix A further describes the Pattern Registry and its `pattern` CLI interface, as well as the data-split construction and task counts for both benchmarks. Algorithm 1 provides the complete optimization procedure, including curriculum decisions, and development-set-based harness selection. Finally, Appendix K provides the full session prompt templates used for failure-pattern extraction, arm scoring, and unseen-scenario exploration. All source code and related artifacts for ActiveSaddler, including the complete `SKILL.md` procedures referenced by the curriculum prompts in Appendix K, are available at `anonymous.4open.science/r/ActiveSaddler-920D`.

REFERENCES

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 Carl Vondrick, Bharath Hariharan, Colin Raffel, Lerrel Pinto, Diyi Yang, and Aleksandra Faust (eds.), *The Fourteenth International Conference on Learning Representations, ICLR 2026, Rio de Janeiro, Brazil, April 23-27, 2026*. proceedings.iclr.cc, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/0e9e708b6f48e14fd0ac29e167413f76-Abstract-Conference.html.

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

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In Andrea Pohoreckyj Danyluk, Léon Bottou, and Michael L. Littman (eds.), *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, volume 382 of *ACM International Conference Proceeding Series*, pp. 41–48. ACM, 2009. doi: 10.1145/1553374.1553380. URL https://doi.org/10.1145/1553374.1553380.

Zhengyu Chen, Teng Xiao, Huaisheng Zhu, Yige Yuan, Luan Zhang, and Jingang Wang. Co-harness: Co-evolving harnesses and model weights for LLM agents. *CoRR*, abs/2607.22688, 2026. doi: 10.48550/ARXIV.2607.22688. URL https://doi.org/10.48550/arXiv.2607.22688.

Jeff Clune. Ai-gas: Ai-generating algorithms, an alternate paradigm for producing general artificial intelligence. *CoRR*, abs/1905.10985, 2019. URL http://arxiv.org/abs/1905.10985.

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. doi: 10.48550/ARXIV.2602.11964. URL https://doi.org/10.48550/arXiv.2602.11964.

GitHub. Github copilot sdk. https://docs.github.com/en/copilot/how-tos/copilot-sdk, 2026. Accessed 2026-09-23.

Alex Graves, Marc G. Bellemare, Jacob Menick, Rémi Munos, and Koray Kavukcuoglu. Automated curriculum learning for neural networks. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1311–1320. PMLR, 2017. URL http://proceedings.mlr.press/v70/graves17a.html.

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. *CoRR*, abs/2607.26722, 2026. doi: 10.48550/ARXIV.2607.26722. URL https://doi.org/10.48550/arXiv.2607.26722.

Prannay Hebbar, Yogendra Manawat, Samuel Verboomen, Alesia Ivanova, Selvam Palanimalai, Kunal Bhatia, and Vignesh Baskaran. SIA: self improving AI with harness & weight updates. *CoRR*, abs/2605.27276, 2026. doi: 10.48550/ARXIV.2605.27276. URL https://doi.org/10.48550/arXiv.2605.27276.

Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized level replay. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pp. 4940–4950. PMLR, 2021. URL http://proceedings.mlr.press/v139/jiang21b.html.

Seth Karten, Joel Zhang, Tersoo Upaa Jr, Ruirong Feng, Wenzhe Li, Chengshuai Shi, Chi Jin, and Kiran Vodrahalli. Continual harness: Online adaptation for self-improving foundation agents. *CoRR*, abs/2605.09998, 2026. doi: 10.48550/ARXIV.2605.09998. URL https://doi.org/10.48550/arXiv.2605.09998.

Haechan Kim, Yoonho Lee, Gisang Lee, Chelsea Finn, and Kangwook Lee. Whale: A simple recipe for joint harness-weight optimization. *arXiv preprint arXiv:2609.00196*, 2026.

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

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

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *CoRR*, abs/2603.28052, 2026. doi: 10.48550/ARXIV.2603.28052. URL https://doi.org/10.48550/arXiv.2603.28052.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Xuanjing Huang, Hang Yan, Zhenhua Han, and Tao Gui. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *CoRR*, abs/2604.25850, 2026. doi: 10.48550/ARXIV.2604.25850. URL https://doi.org/10.48550/arXiv.2604.25850.

Manuel Lopes and Pierre-Yves Oudeyer. The strategic student approach for life-long exploration and learning. In *2012 IEEE International Conference on Development and Learning and Epigenetic Robotics, ICDL-EPIROB 2012, San Diego, CA, USA, November 7-9, 2012*, pp. 1–8. IEEE, 2012. doi: 10.1109/DEVLRN.2012.6400807. URL https://doi.org/10.1109/DevLrn.2012.6400807.

Tambet Matiisen, Avital Oliver, Taco Cohen, and John Schulman. Teacher-student curriculum learning. *CoRR*, abs/1707.00183, 2017. URL http://arxiv.org/abs/1707.00183.

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. doi: 10.48550/ARXIV.2601.11868. URL https://doi.org/10.48550/arXiv.2601.11868.

Atsuyuki Miyai, Kiyoharu Aizawa, and Toshihiko Yamasaki. Task-coevolve: Efficient harness optimization via adaptive validation task selection. *CoRR*, abs/2608.20169, 2026. doi: 10.48550/ARXIV.2608.20169. URL https://doi.org/10.48550/arXiv.2608.20169.

Jun Nie, Yonggang Zhang, Jun Song, Qianshu Cai, Dahai Yu, Yike Guo, Xinmei Tian, and Bo Han. TTHE: test-time harness evolution. *CoRR*, abs/2607.08124, 2026. doi: 10.48550/ARXIV.2607.08124. URL https://doi.org/10.48550/arXiv.2607.08124.

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

Pierre-Yves Oudeyer, Frédéric Kaplan, and Verena Vanessa Hafner. Intrinsic motivation systems for autonomous mental development. *IEEE Trans. Evol. Comput.*, 11(2):265–286, 2007. doi: 10.1109/TEVC.2006.890271. URL https://doi.org/10.1109/TEVC.2006.890271.

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. *CoRR*, abs/2606.05922, 2026. doi: 10.48550/ARXIV.2606.05922. URL https://doi.org/10.48550/arXiv.2606.05922.

Sungho Park, Wonjoong Kim, Rongyuan Tan, Jue Zhang, Wook-Shin Han, Pengfei Gao, Chanyoung Park, Yongqiang Yao, Rao Fu, Elsie Nallipogu, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Autosaddler: Automatic harness optimization with durable updates from agent execution traces. *CoRR*, abs/2608.23041, 2026. doi: 10.48550/ARXIV.2608.23041. URL https://doi.org/10.48550/arXiv.2608.23041.

Rémy Portelas, Cédric Colas, Katja Hofmann, and Pierre-Yves Oudeyer. Teacher algorithms for curriculum learning of deep RL in continuously parameterized environments. In Leslie Pack Kaelbling, Danica Kragic, and Komei Sugiura (eds.), *3rd Annual Conference on Robot Learning, CoRL 2019, Osaka, Japan, October 30 - November 1, 2019, Proceedings*, volume 100 of *Proceedings of Machine Learning Research*, pp. 835–853. PMLR, 2019. URL http://proceedings.mlr.press/v100/portelas20a.html.

Biswa Sengupta and Jinhua Wang. HARBOR: automated harness optimization. *CoRR*, abs/2604.20938, 2026. doi: 10.48550/ARXIV.2604.20938. URL https://doi.org/10.48550/arXiv.2604.20938.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O. Stanley. Paired open-ended trailblazer (POET): endlessly generating increasingly complex and diverse learning environments and their solutions. *CoRR*, abs/1901.01753, 2019. URL http://arxiv.org/abs/1901.01753.

Wenyi Wang, Piotr Piekos, Nanbo Li, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley-gödel machine: Human-level coding agent development by an approximation of the optimal self-improving machine. *CoRR*, abs/2510.21614, 2025. doi: 10.48550/ARXIV.2510.21614. URL https://doi.org/10.48550/arXiv.2510.21614.

Yizao Wang, Jean-Yves Audibert, and Rémi Munos. Algorithms for infinitely many-armed bandits. In Daphne Koller, Dale Schuurmans, Yoshua Bengio, and Léon Bottou (eds.), *Advances in Neural Information Processing Systems 21, Proceedings of the Twenty-Second Annual Conference on Neural*

Information Processing Systems, Vancouver, British Columbia, Canada, December 8-11, 2008, pp. 1729–1736. Curran Associates, Inc., 2008. URL https://proceedings.neurips.cc/paper/2008/hash/49ae49a23f67c759bf4fc791ba842aa2-Abstract.html.

Tianxin Wei, Zhan Shi, Minhua Lin, Bing He, Zewen Liu, Yisi Sang, Yuanchen Bei, Xuying Ning, Jiaru Zou, Ting-Wei Li, Xiao Lin, Yanjun Zhao, Chi Wang, Benoît Dumoulin, Dakuo Wang, Jingrui He, and Hanqing Lu. Evo-harness: Context-to-harness skill compilation for self-evolving agents. *CoRR*, abs/2608.15071, 2026. doi: 10.48550/ARXIV.2608.15071. URL https://doi.org/10.48550/arXiv.2608.15071.

Yuxin Wu and Yuandong Tian. Training agent for first-person shooter game with actor-critic curriculum learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=Hk3mPK5gg.

Jinghan Xu, Yikai Zhang, Aili Chen, Weiyuan Li, Jiaqing Liang, and Deqing Yang. Verify smarter, evolve further: Efficient harness evolution through behavior-aware verification. *CoRR*, abs/2608.27311, 2026. doi: 10.48550/ARXIV.2608.27311. URL https://doi.org/10.48550/arXiv.2608.27311.

Shidong Yang, Ziyu Ma, Tongwen Huang, Yiming Hu, Yong Wang, and Xiangxiang Chu. Coevolve: Training LLM agents via agent-data mutual evolution. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2026, San Diego, California, United States, July 2-7, 2026*, pp. 23015–23036. Association for Computational Linguistics, 2026a. doi: 10.18653/V1/2026.ACL-LONG.1055. URL https://doi.org/10.18653/v1/2026.acl-long.1055.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. Skillopt: Executive strategy for self-evolving agent skills. *CoRR*, abs/2605.23904, 2026b. doi: 10.48550/ARXIV.2605.23904. URL https://doi.org/10.48550/arXiv.2605.23904.

Wojciech Zaremba and Ilya Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014. URL http://arxiv.org/abs/1410.4615.

Luan Zhang, Ruochen Zhou, Dandan Song, Zhengyu Chen, Yuhang Tian, Jun Yang, Huipeng Ma, Chenhao Li, Guangyuan Feng, Xudong Li, Yizhou Jin, and Yan Xu. Harnesscompass: Guiding automatic harness evolution toward generalizable and effective agent harnesses. *CoRR*, abs/2608.01918, 2026. doi: 10.48550/ARXIV.2608.01918. URL https://doi.org/10.48550/arXiv.2608.01918.

Chenyu Zhou, Huacan Chai, Wenteng Chen, Zihan Guo, Rong Shan, Yuanyi Song, Tianyi Xu, Yingxuan Yang, Aofan Yu, Weiming Zhang, Congming Zheng, Jiachen Zhu, Zeyu Zheng, Zhuosheng Zhang, Xingyu Lou, Changwang Zhang, Zhihui Fu, Jun Wang, Weiwen Liu, Jianghao Lin, and Weinan Zhang. Externalization in LLM agents: A unified review of memory, skills, protocols and harness engineering. *CoRR*, abs/2604.08224, 2026. doi: 10.48550/ARXIV.2604.08224. URL https://doi.org/10.48550/arXiv.2604.08224.

A Additional Details on Experiment Setup

Implementation of Pattern Registry. The Pattern Registry stores each failure pattern, i.e., each arm, as a symptom-level label with its tagged *(harness, trace, scenario)* tuples, root-cause evidence, activity observations, and learning-progress scores. The registry can be implemented straightforwardly, but providing effective agent access to it is more challenging: as arms accumulate observation histories and evidence across iterations, serializing the registry into the prompt is neither compact nor selective. We therefore introduce the `pattern` command-line interface (CLI) to facilitate interaction between the GC-SDK and the Pattern Registry. The interface provides on-demand access to scoped views of the arm population, including the canonical pattern table, per-pattern evidence, score rankings, and the complete pull history of an arm with its patch intents; it is also the only channel through which the agent registers arms, tags tuples, and records scores and decisions. Because sessions differ in what they are permitted to do, the outer loop installs the CLI with a per-session capability manifest determined by the sampling strategy, and commands outside it are refused. The full command set is summarized in Table 2.

Table 2: Commands exposed by the `pattern` CLI, grouped into read operations for scoped views of the arm population and write operations for per-session registry updates. Each session receives only the subset of commands in its capability manifest.

Command	Purpose	When to use
Read operations		
`pattern list`	Canonical arm table: score, observation history, scenario count, last observation, label	Start of session; orienting to the full arm population
`pattern show <id>`	Pattern details: label, score, tagged tuples, per-scenario root causes, observations	Inspecting a specific arm before rating or patching
`pattern score [-top-k <n>]`	Top-k arms ranked by current score	Quick triage of the highest-priority arms
`pattern scenarios -pattern-id <id>`	Scenarios tagged to an arm, with co-occurring patterns	Checking an arm's coverage and overlap
`pattern history [-pattern-id <ids>] [-last-k <K>]`	Complete per-arm pull history: patched attempts, all-pass skips, and failed attempts, with diagnoses, patch intents, dev-set impact, and per-scenario reflections	Before scoring or re-selecting an arm; assessing what prior repairs already attempted
Write operations		
`pattern register -label "..."`	Create a new arm from a symptom-level label; returns its pattern ID	Pattern Extraction Session, for a newly observed failure type
`pattern tag -pattern-id <ids> -harness <idx> -trace <dir> -scenario <sid> [-root-cause "..."]`	Bind a *(harness, trace, scenario)* tuple to one or more patterns and record its root-cause evidence	Pattern Extraction Session, after normalizing a candidate against existing arms
`pattern rate -pattern-id <id> -severity -fixability -breadth -side-effect [-rationale]`	Record the cardinal learning-progress score for the current iteration from the severity, fixability, breadth, and side-effect axes	Arm Scoring Session, once per arm; an unrated arm defaults to the lowest priority
`pattern decide -action pull|draw [-rationale]`	Record the exploration decision: exploit a known arm (`pull`) or explore the unseen-scenario pool for a new failure type (`draw`)	Unseen-Scenario Exploration Session, exactly once per iteration

Data Splits across Benchmarks. To evaluate whether optimized harnesses generalize beyond the tasks seen during optimization, we use the same train, development, and test splits as AutoSaddler (Park et al., 2026) for each benchmark, as summarized in Table 3. 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. 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 3: Data splits across the two benchmarks. For GAIA2, we evaluate generalization under distribution shift by partitioning the benchmark such that train, development, and test sets contain tasks from disjoint `Universes` (personas), rather than using 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
Terminal-Bench 2.0	Random[†]	Train	—	30
		Dev	—	19
		Test	—	40

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

Hyperparameters. We set the shared rollout budget to $K = 10\,(|\mathcal{D}_{\text{train}}| + |\mathcal{D}_{\text{dev}}|)$, corresponding to 1,400 rollouts on GAIA2 and 490 on Terminal-Bench 2.0. We choose this budget to match ten full passes over the Meta-Harness training set: following the configuration used in AutoSaddler (Park et al., 2026), Meta-Harness treats $\mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{dev}}$ as its training set, and Meta-Harness (Lee et al., 2026) runs optimization for 10 iterations. For all methods, we limit each optimization iteration to at most three scenarios, matching the mini-batch size used in AutoSaddler (Park et al., 2026). We set the arm-selection temperature to $\tau_{\text{sel}} = 0.15$.

B Algorithm of ActiveSaddler

General Algorithm. Algorithm 1 operationalizes the curriculum policy defined in Sections 3 and 4. At each iteration, the **Exploration Controller** first decides whether to explore scenarios from the current exploration pool or continue optimization of an instantiated failure-pattern arm. If an arm is chosen, the **Arm Prioritizer** scores the current arms and samples the next optimization target. After the fixed harness optimizer $\mathcal{O}$ updates the harness, the **Failure-Pattern Extractor** converts the resulting failure evidence into matched, composed, or newly instantiated failure-pattern arms. Thus, the curriculum determines which training scenarios form X_t, while $\mathcal{O}$ determines how the resulting execution evidence is translated into harness updates and uses D_{dev} for candidate selection, regression checks, and generalization checks.

The regression-rediscovery rule in Section 4 is handled within the Exploration Controller. Once all training scenarios have been explored and every arm has since been visited, previously successful scenarios that did not instantiate an arm become eligible for exploration again. This changes only the eligibility of scenarios for the `unseen` branch and does not introduce an additional curriculum action. Following AutoSaddler (Park et al., 2026), all task-level executions incurred during optimization, including those used by the underlying optimizer to evaluate candidate harnesses, count toward the fixed rollout budget K, and the final harness is selected based on development-set performance.

Algorithm 1 ActiveSaddler for automated curriculum learning in harness optimization

Require: Initial harness parameters θ_0, training set D_{train}, development set D_{dev}, rollout budget K, fixed harness optimizer $\mathcal{O}$, arm-selection temperature τ_{sel}, and procedures $g_{\text{explore}}, g_{\text{select}}, g_{\text{ext}}, g_{\text{norm}}$

$H_0 \leftarrow H_{\theta_0}$
Initialize optimization history $\mathcal{I}_{<0}$ with H_0 and arm set $\mathcal{A}_0 \leftarrow \varnothing$
$U_0 \leftarrow D_{\text{train}}$; $t \leftarrow 0$
while rollout budget remains **do**
Exploration Controller
  if $U_t = \varnothing$, all scenarios in D_{train} have been explored, and every instantiated arm has since been visited **then**
    Add to U_t previously successful scenarios that did not instantiate a failure-pattern arm
  end if
  if $\mathcal{A}_t = \varnothing$ and $U_t \neq \varnothing$ **then**
    $d_t \leftarrow$ `unseen`
  else if $U_t = \varnothing$ and $\mathcal{A}_t \neq \varnothing$ **then**
    $d_t \leftarrow$ `arm`
  else if $\mathcal{A}_t \neq \varnothing$ and $U_t \neq \varnothing$ **then**
    $d_t \leftarrow g_{\text{explore}}(H_t, \mathcal{A}_t, U_t, \mathcal{I}_{<t})$
  else
    break
  end if
  if $d_t =$ `unseen` **then**
    Sample $X_t \subseteq U_t$ without replacement, subject to the per-iteration execution limit
    $U_t \leftarrow U_t \setminus X_t$
  else
Arm Prioritizer
    for all $a \in \mathcal{A}_t$ **do**
      $\phi_t(a) \leftarrow g_{\text{select}}(a, \mathcal{I}_{<t})$
    end for
    $q_t(a) \leftarrow \dfrac{\exp(\phi_t(a)/\tau_{\text{sel}})}{\sum_{a' \in \mathcal{A}_t} \exp(\phi_t(a')/\tau_{\text{sel}})}$ for all $a \in \mathcal{A}_t$
    $a_t \sim \text{Categorical}(q_t)$
    Construct $X_t \subseteq S_t(a_t)$, subject to the per-iteration execution limit
  end if
Harness Optimizer
  $\mathcal{E}_t \leftarrow \text{EXECUTE}(H_t, X_t)$
  $\theta_{t+1} \sim \mathcal{O}(\theta_t, \mathcal{E}_t, D_{\text{dev}})$; $H_{t+1} \leftarrow H_{\theta_{t+1}}$
  Obtain failed execution records $\mathcal{F}_t$ and diagnostic evidence $\{\delta_t(e)\}_{e \in \mathcal{F}_t}$ made available by $\mathcal{O}$
Failure-Pattern Extractor
  for all $e \in \mathcal{F}_t$ **do**
    $c_t(e) \leftarrow g_{\text{ext}}(e, \delta_t(e))$ ▷ Symptom extraction
  end for
  $\mathcal{A}_{t+1} \leftarrow g_{\text{norm}}(\mathcal{A}_t, \{c_t(e)\}_{e \in \mathcal{F}_t})$ ▷ Pattern normalization
  Update the supporting scenario sets and associated harness, execution, and diagnostic evidence for matched, composed, or newly instantiated arms
Curriculum-State Update
  $U_{t+1} \leftarrow U_t$
  Update $\mathcal{I}_{<t+1}$ with H_{t+1}, X_t, the execution and optimizer outcomes, and the curriculum decision
  $t \leftarrow t + 1$
end while
Select $\hat{\theta}_\pi$ by development-set performance
return $H_{\hat{\theta}_\pi}$

AutoSaddler Instantiation. Algorithm 2 instantiates the same ActiveSaddler curriculum on AutoSaddler. The three curriculum components, the **Exploration Controller**, **Arm Prioritizer**, and **Failure-Pattern Extractor**, are unchanged from Algorithm 1. We repeat the full procedure for line-by-line comparison, with only the AutoSaddler-specific optimizer operations highlighted in blue.

The AutoSaddler-specific difference lies in how the selected execution records are converted into a harness update and which failures are exposed to the Failure-Pattern Extractor. After executing H_t on X_t, the Diagnosis–Patch Session diagnoses the observed failures and derives a structured patch, yielding a candidate harness H'_t. The candidate is then executed on the same scenario batch

Algorithm 2 ActiveSaddler instantiated on AutoSaddler (AutoSaddler-specific operations highlighted in blue)

Require: Initial harness parameters θ_0, training set D_{train}, development set D_{dev}, rollout budget K, AutoSaddler as the fixed harness optimizer $\mathcal{O}$, arm-selection temperature τ_{sel}, and procedures $g_{\text{explore}}, g_{\text{select}}, g_{\text{ext}}, g_{\text{norm}}$

$H_0 \leftarrow H_{\theta_0}$
Initialize optimization history $\mathcal{I}_{<0}$ with H_0 and arm set $\mathcal{A}_0 \leftarrow \varnothing$
$U_0 \leftarrow D_{\text{train}}$; $t \leftarrow 0$
while rollout budget remains **do**
Exploration Controller
  if $U_t = \varnothing$, all scenarios in D_{train} have been explored, and every instantiated arm has since been visited **then**
    Add to U_t previously successful scenarios that did not instantiate a failure-pattern arm
  end if
  if $\mathcal{A}_t = \varnothing$ and $U_t \neq \varnothing$ **then**
    $d_t \leftarrow$ `unseen`
  else if $U_t = \varnothing$ and $\mathcal{A}_t \neq \varnothing$ **then**
    $d_t \leftarrow$ `arm`
  else if $\mathcal{A}_t \neq \varnothing$ and $U_t \neq \varnothing$ **then**
    $d_t \leftarrow g_{\text{explore}}(H_t, \mathcal{A}_t, U_t, \mathcal{I}_{<t})$
  else
    break
  end if
  if $d_t =$ `unseen` **then**
    Sample $X_t \subseteq U_t$ without replacement, subject to the per-iteration execution limit
    $U_t \leftarrow U_t \setminus X_t$
  else
Arm Prioritizer
    for all $a \in \mathcal{A}_t$ **do**
      $\phi_t(a) \leftarrow g_{\text{select}}(a, \mathcal{I}_{<t})$
    end for
    $q_t(a) \leftarrow \dfrac{\exp(\phi_t(a)/\tau_{\text{sel}})}{\sum_{a' \in \mathcal{A}_t} \exp(\phi_t(a')/\tau_{\text{sel}})}$ for all $a \in \mathcal{A}_t$
    $a_t \sim \text{Categorical}(q_t)$
    Construct $X_t \subseteq S_t(a_t)$, subject to the per-iteration execution limit
  end if
AutoSaddler Optimizer
  $\mathcal{E}_t \leftarrow \textsc{Execute}(H_t, X_t)$
  if $\textsc{Failed}(\mathcal{E}_t) \neq \varnothing$ **then**
    Diagnose failures in $\mathcal{E}_t$ and derive a structured patch, yielding candidate harness H'_t
    $\mathcal{E}_t^{\text{post}} \leftarrow \textsc{Execute}(H'_t, X_t)$
    If H'_t improves over H_t on X_t, additionally evaluate H'_t on D_{dev}
    Reflect on the pre-/post-patch outcomes; record *fixed*, *regressed*, *still-failing*, and *still-passing* cases; and update EvoDAG
    $H_{t+1} \leftarrow \textsc{Evolve}(\text{EvoDAG})$
    $\mathcal{F}_t \leftarrow \textsc{Failed}(\mathcal{E}_t) \cup \textsc{Failed}(\mathcal{E}_t^{\text{post}})$, with diagnostic evidence $\{\delta_t(e)\}_{e \in \mathcal{F}_t}$ from the Diagnosis–Patch and Reflection sessions
  else
    $H_{t+1} \leftarrow H_t$; $\mathcal{F}_t \leftarrow \varnothing$
  end if
Failure-Pattern Extractor
  for all $e \in \mathcal{F}_t$ **do**
    $c_t(e) \leftarrow g_{\text{ext}}(e, \delta_t(e))$ ▷ Symptom extraction
  end for
  $\mathcal{A}_{t+1} \leftarrow g_{\text{norm}}(\mathcal{A}_t, \{c_t(e)\}_{e \in \mathcal{F}_t})$ ▷ Pattern normalization
  Update the supporting scenario sets and associated harness, execution, and diagnostic evidence for matched, composed, or newly instantiated arms
Curriculum-State Update
  $U_{t+1} \leftarrow U_t$
  Update $\mathcal{I}_{<t+1}$ with H_{t+1}, X_t, the execution and optimizer outcomes, and the curriculum decision
  $t \leftarrow t + 1$
end while
Select $\hat{\theta}_\pi$ by development-set performance
return $H_{\hat{\theta}_\pi}$

X_t and, if it improves over H_t on this batch, is additionally evaluated on D_{dev}. The Reflection Session compares the pre- and post-patch executions, records *fixed*, *regressed*, *still-failing*, and *still-passing* cases, and updates EvoDAG. The Evolution Session then synthesizes H_{t+1} from the updated EvoDAG. For this instantiation, $\mathcal{F}_t$ contains failures diagnosed before patching together with failures that remain or newly emerge during post-patch evaluation, with diagnostic evidence supplied by the Diagnosis–Patch and Reflection sessions. The same Failure-Pattern Extractor then processes $\mathcal{F}_t$ exactly as in the general algorithm.

C Robustness to Optimization Stochasticity

Table 4: **Robustness to optimization stochasticity.** Pass@1 on three held-out GAIA2 universes for harnesses obtained from two independent optimization runs of AutoSaddler, its difficulty-based fixed curricula, and ActiveSaddler. Performance on each universe is reported as mean ± standard deviation over three test-time executions, and the average is computed across all test scenarios.

Harness (Type)	GAIA2 Universe (Pass@1)			Avg.
	21 (107)	22 (112)	27 (81)	
Default Agent (manual)	54.5 ± 0.5	50.0 ± 1.5	57.2 ± 1.4	53.6 ± 1.1
AutoSaddler (Run 1)	57.0 ± 0.9	51.5 ± 2.2	58.8 ± 0.7	55.4 ± 1.2
AutoSaddler (Run 2)	57.6 ± 1.4	53.0 ± 1.0	60.1 ± 2.9	56.6 ± 1.0
AutoSaddler w/ Category Acc. Order (Run 1)	56.1 ± 3.4	53.9 ± 1.0	58.4 ± 1.9	55.9 ± 1.3
AutoSaddler w/ Category Acc. Order (Run 2)	57.0 ± 3.4	53.6 ± 0.9	58.4 ± 0.7	56.1 ± 1.0
AutoSaddler w/ Scenario Acc. Order (Run 1)	57.0 ± 1.9	53.3 ± 1.0	57.2 ± 3.8	55.7 ± 1.2
AutoSaddler w/ Scenario Acc. Order (Run 2)	56.1 ± 3.7	52.7 ± 2.4	60.1 ± 1.9	55.9 ± 0.7
ActiveSaddler (Run 1)	60.4 ± 1.4	57.7 ± 2.6	61.7 ± 1.2	59.8 ± 1.0
ActiveSaddler (Run 2)	**62.0 ± 4.4**	**60.4 ± 1.9**	**62.6 ± 2.6**	**61.6 ± 2.9**

To assess robustness to optimization stochasticity, we conduct two independent GAIA2 optimization runs for AutoSaddler, its category- and scenario-level difficulty-based fixed curricula, and ActiveSaddler. Each resulting harness is evaluated on three held-out universes (Universes 21, 22, and 27; 300 scenarios in total) with three repeated test-time executions. Both runs of ActiveSaddler outperform all fixed-order baselines on each held-out universe. Across the three universes, ActiveSaddler achieves 59.8% and 61.6% Pass@1 in Runs 1 and 2, exceeding the fixed-order baselines by 3.9 to 4.4 and 5.0 to 5.7 percentage points, respectively. These consistent gains across independent optimization runs and held-out universes indicate that the improvement of ActiveSaddler is not specific to a single optimization trajectory.

D Alternative Arm Prioritization and Exploration Strategies

Table 5: **Comparison with alternative curriculum strategies.** Pass@1 on three held-out GAIA2 universes when replacing either the Arm Prioritizer or Exploration Controller in ActiveSaddler with an alternative strategy while keeping the remaining curriculum components unchanged. Performance on each universe is reported as mean ± standard deviation over three test-time executions.

Method	GAIA2 Universe (Pass@1)			Avg.
	21 (107)	22 (112)	27 (81)	
Default Agent (manual)	54.5 ± 0.5	50.0 ± 1.5	57.2 ± 1.4	53.6 ± 1.1
AutoSaddler	57.0 ± 0.9	51.5 ± 2.2	58.8 ± 0.7	55.4 ± 1.2
ActiveSaddler	**60.4 ± 1.4**	**57.7 ± 2.6**	**61.7 ± 1.2**	**59.8 ± 1.0**
ActiveSaddler w/ EMA Scoring	57.9 ± 2.5	53.0 ± 3.4	60.1 ± 0.7	56.7 ± 1.5
ActiveSaddler w/ UCB-AIR Controller	57.9 ± 3.4	50.9 ± 1.5	58.8 ± 3.1	55.6 ± 1.4
w/o Arm Prioritizer	57.3 ± 1.4	51.2 ± 3.1	58.4 ± 4.3	55.3 ± 2.1
w/o Exploration Controller	56.4 ± 2.2	51.5 ± 0.5	59.3 ± 3.3	55.3 ± 0.9

To examine alternative design choices for the two curriculum decisions in ActiveSaddler, we replace the Arm Prioritizer and Exploration Controller, one at a time, with simple non-LLM alternatives while

keeping the Failure-Pattern Extractor, arm-selection distribution, and underlying harness optimizer unchanged.

First, **ActiveSaddler w/ EMA Scoring** replaces the LLM-based learning-progress estimate $\phi_t(a) = g_{\text{select}}(a, \mathcal{I}_{<t})$ with an exponential moving-average (EMA) score, following the EMA-based curriculum scoring used by TSCL (Matiisen et al., 2017). In our setting, the score tracks the residual failure rate of each arm after patching:

$$\phi_t(a) = \bar{X}_t(a), \qquad \bar{X}_t(a) = \begin{cases} (1-\eta)\bar{X}_{t-1}(a) + \eta\,\text{active}_t(a), & \text{if } a \text{ is observed at } t, \\ \bar{X}_{t-1}(a), & \text{otherwise,} \end{cases}$$

where $\text{active}_t(a)$ is the fraction of evaluated scenarios associated with a that still exhibit the same failure pattern after patching. Thus, recurring failures that remain unresolved receive higher scores and are prioritized for further optimization. Following TSCL (Matiisen et al., 2017), we set $\eta = 0.1$. Newly instantiated arms are initialized with $\bar{X}_t(a) = 1$.

Second, **ActiveSaddler w/ UCB-AIR Controller** retains the LLM-based Arm Prioritizer but replaces the Exploration Controller $d_t = g_{\text{explore}}(H_t, \mathcal{A}_t, U_t, \mathcal{I}_{<t})$ with a count-based arm-creation schedule derived from UCB-AIR for infinitely many-armed bandits (Wang et al., 2008):

$$d_t = \begin{cases} \texttt{unseen}, & N_t^{\alpha} \geq |\mathcal{A}_t|, \\ \texttt{arm}, & \text{otherwise,} \end{cases}$$

where N_t is the cumulative number of scenario executions so far. We set $\alpha = 0.6$, following the UCB-AIR configuration adopted by HGM (Wang et al., 2025). This rule periodically explores unseen scenarios as optimization proceeds, using only the number of previous executions and instantiated arms rather than the current optimization state.

Table 5 reports the results. EMA scoring achieves 56.7% average Pass@1, improving by 1.4 percentage points over removing the Arm Prioritizer (55.3%), showing that even a simple failure-persistence signal provides useful prioritization. However, it remains 3.1 points below the 59.8% achieved by ActiveSaddler. By comparison, the UCB-AIR controller achieves 55.6%, only 0.3 points above the fixed exploration schedule (55.3%) and 4.2 points below ActiveSaddler. Together, these results show that, among the tested alternatives, the LLM-based modules are more effective for the two key curriculum decisions: which known failure pattern to revisit and when to explore unseen scenarios for new ones.

E Applying ActiveSaddler to GEPA

Table 6: **Applying ActiveSaddler to GEPA.** Pass@1 on three held-out GAIA2 universes for the default agent, GEPA, and GEPA augmented with the ActiveSaddler curriculum. Performance on each universe is reported as mean ± standard deviation over three test-time executions. The average is computed across all test scenarios in the three universes.

Method	GAIA2 Universe (Pass@1)			Avg.
	21 (107)	22 (112)	27 (81)	
Default Agent (manual)	54.5 ± 0.5	50.0 ± 1.5	57.2 ± 1.4	53.6 ± 1.1
GEPA	56.4 ± 1.4	50.0 ± 2.4	57.2 ± 3.6	54.2 ± 2.2
GEPA w/ ActiveSaddler	**58.6 ± 0.5**	**54.2 ± 1.0**	**59.7 ± 0.7**	**57.2 ± 0.4**

Our main experiments instantiate ActiveSaddler on AutoSaddler. To examine whether its curriculum benefit extends beyond AutoSaddler, we additionally apply ActiveSaddler to GEPA, another offline optimizer. At each iteration, ActiveSaddler selects the training-scenario batch provided to GEPA, while GEPA's underlying prompt-update and candidate-selection procedure remains unchanged. We follow the same GAIA2 train/development/test split and test-time evaluation protocol as in the main experiments.

Table 7: **Optimizer-side cost per generated patch.** Runtime, monetary cost, LLM calls, and token usage are averaged per generated patch.

GAIA2

Method	Generated	Rejected	Accepted	Time (s)	Cost ($)	LLM Calls	Output Tokens	Cache-Read Input Tokens
AutoSaddler	32	14	18	943	7.87	95.4	57,751	5,956,240
w/ Category Acc. Order	20	7	13	889	6.71	91.7	59,945	5,717,811
w/ Scenario Acc. Order	30	18	12	1,142	7.75	100.3	61,791	6,087,970
ActiveSaddler	37	18	19	1,581	11.44	142.2	89,525	8,704,858

Terminal-Bench 2.0

Method	Generated	Rejected	Accepted	Time (s)	Cost ($)	LLM Calls	Output Tokens	Cache-Read Input Tokens
AutoSaddler	26	12	14	977	6.77	79.2	47,389	4,513,280
w/ Category Acc. Order	20	12	8	801	6.71	80.2	49,387	4,641,920
w/ Scenario Acc. Order	14	5	9	875	7.21	82.9	52,445	4,796,855
ActiveSaddler	35	17	18	1,438	11.37	121.5	75,257	6,873,936

Table 8: **Average task-agent evaluation cost per rollout.**

GAIA2

Method	LLM Calls	Input Tokens	Output Tokens	Time (s)
AutoSaddler	18.9	467,996	13,678	231.6
w/ Category Acc. Order	16.5	381,948	11,629	201.4
w/ Scenario Acc. Order	16.4	410,934	11,948	199.8
ActiveSaddler	17.5	410,960	12,340	212.0

Terminal-Bench 2.0

Method	LLM Calls	Input Tokens	Output Tokens	Time (s)
AutoSaddler	11.8	204,258	10,689	305.4
w/ Category Acc. Order	14.8	331,167	12,124	365.5
w/ Scenario Acc. Order	14.9	321,888	12,198	369.6
ActiveSaddler	13.4	307,070	11,553	322.6

Table 6 reports the results. Applying ActiveSaddler improves GEPA on all three held-out universes, from 56.4% to 58.6% on Universe 21, from 50.0% to 54.2% on Universe 22, and from 57.2% to 59.7% on Universe 27. Aggregated over the three universes, GEPA w/ ActiveSaddler achieves 57.2% Pass@1, compared with 54.2% for GEPA, an improvement of 3.0 percentage points. These results show that the benefit of adapting which training scenarios drive optimization also extends beyond the AutoSaddler instantiation used in our main experiments.

F End-to-End Optimization Cost Characterization

We profile optimizer-side computation and task-agent execution separately, and then combine them to characterize end-to-end optimization cost. Optimizer-side cost includes AutoSaddler's diagnosis, patch generation, reflection, and candidate selection; for ActiveSaddler, it additionally includes the curriculum-layer operations for failure-pattern extraction, arm scoring, and unseen-scenario exploration. This decomposition allows us to examine how the additional computation introduced by the curriculum layer affects overall optimization efficiency.

Cost Breakdown. We first examine optimizer-side and task-agent costs separately to understand where the end-to-end optimization cost arises. Table 7 reports optimizer-side resource usage per generated patch, including AutoSaddler's diagnosis, patch generation, reflection, and candidate selection; for ActiveSaddler, it additionally includes the curriculum-layer operations for failure-pattern extraction, arm scoring, and unseen-scenario exploration. On GAIA2, ActiveSaddler costs $11.44 per generated patch, compared with $7.87 for AutoSaddler, $6.71 for category ordering, and $7.75 for scenario ordering. On Terminal-Bench 2.0, the corresponding costs are $11.37, $6.77, $6.71, and $7.21. In contrast, Table 8 shows that the average cost of an individual task-agent rollout

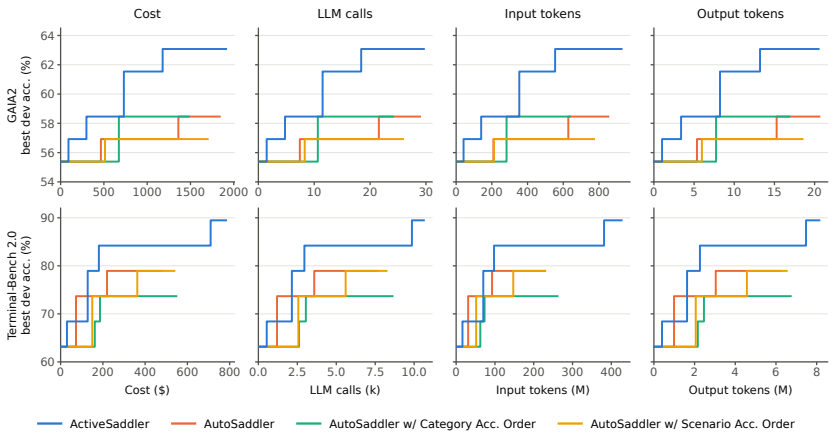


Figure 5: **End-to-end optimization efficiency.** Best-so-far development accuracy against cumulative monetary cost, LLM calls, input tokens, and output tokens on GAIA2 (top) and Terminal-Bench 2.0 (bottom). Cumulative usage includes both optimizer-side and task-agent computation.

is broadly comparable across methods on each benchmark. This suggests that differences in total task-agent cost are driven primarily by how rollouts are allocated over the optimization trajectory rather than by substantially cheaper individual executions.

End-to-End Cost Efficiency. We next combine optimizer-side and task-agent usage along each optimization trajectory and measure the best development accuracy achieved at each cumulative resource level. For the two accuracy-ordered AutoSaddler variants, the five full training-set evaluations used to construct the ordering are counted toward both the shared rollout budget and the cumulative optimization cost. Figure 5 reports cumulative monetary cost, LLM calls, input tokens, and output tokens, with every method truncated at the shared rollout budget.

After accounting for both optimizer-side and task-agent computation, ActiveSaddler achieves a better end-to-end cost–accuracy trade-off on both benchmarks. On GAIA2, ActiveSaddler reaches 58.5% development accuracy at \$298, 61.5% at \$730, and 63.1% at \$1,178. In contrast, AutoSaddler reaches 58.5% only at \$1,360 and remains at that accuracy after \$1,838, category ordering reaches 58.5% at \$673 and remains there after \$1,479, and scenario ordering does not exceed 56.9% after \$1,698. On Terminal-Bench 2.0, ActiveSaddler reaches 78.9% at \$128 and 84.2% at \$181, whereas AutoSaddler and scenario ordering require \$220 and \$363, respectively, to reach their peak of 78.9%, and category ordering does not exceed 73.7% after \$548. The same qualitative advantage holds for cumulative LLM calls and token usage. Overall, these results show that, despite its additional optimizer-side overhead, ActiveSaddler achieves better end-to-end cost efficiency by allocating task-agent rollouts to more useful optimization targets.

G Additional Analysis for RQ1

Case Studies. Figure 6 illustrates how retargeting the same failure-pattern arm enables iterative refinement of an initially unsuccessful repair. In this task, the agent should email two friends to confirm their attendance and, if either responds positively within four minutes, immediately reply that the party starts at 10 PM. The agent completed the interaction but produced an affirmative event-start reply that omitted the required closing and user-name signature. This failure instantiated a failure-pattern arm capturing that affirmative event-start replies were not canonicalized to the required acknowledgement structure. The first repair at Iteration 3 added an executor hook that detects such replies and appends the required closing and real-name signature. However, the hook was designed to trigger only when the model-generated reply was a single-line draft. Because the model instead produced a multiline reply, the hook did not trigger and the scenario remained unsuccessful. Because the weakness remained explicitly represented as a failure-pattern arm, it was retargeted at Iteration 5.

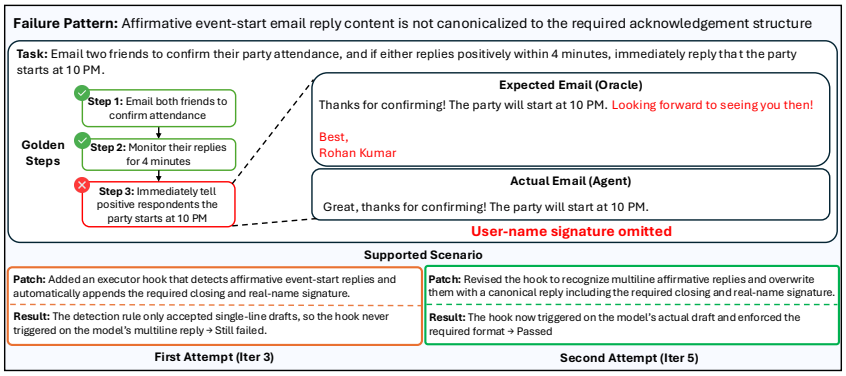


Figure 6: Case study on `tnxtee` (GAIA2), where a failure-pattern arm instantiated from the initial failure enables an unsuccessful repair to be retargeted and refined in a subsequent iteration.

The second repair extended the trigger condition to multiline affirmative replies, allowing the hook to correctly append the required closing and real-name signature; the scenario then passed.

In the category-arm variant, `tnxtee` was grouped with 10 other scenarios under the much broader `category-time` arm. After the first sampled time batch passed, the arm's severity dropped sharply from 0.62 to 0.12, and its median rank was only fourth among the five category arms. Although the arm was selected three times, `tnxtee` itself was never sampled. Thus, successes on other time-related scenarios reduced the estimated value of the entire category before this still-active weakness was observed, preventing the optimizer from obtaining an opportunity to repair it.

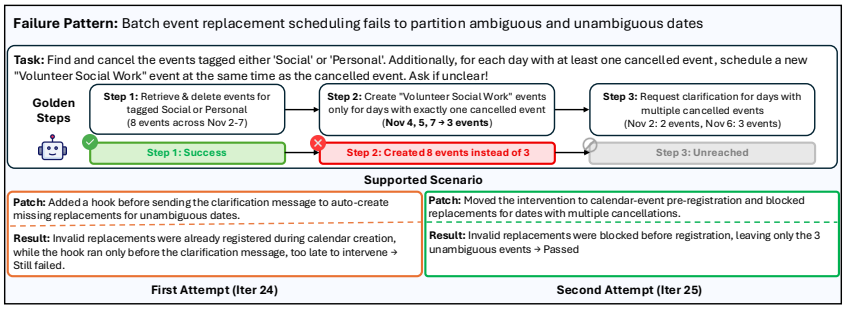


Figure 7: Case study on `4bytkx` (GAIA2), where retargeting the same failure-pattern arm corrects an executor hook that was initially inserted too late in the execution flow.

Figure 7 illustrates a complementary form of repair refinement, where the first repair addresses the correct weakness but places the executor hook too late in the execution flow. In this task, the agent should cancel all events tagged `Social` or `Personal`, create replacement events only for dates with a single cancelled event, and request clarification for dates with multiple cancellations. Instead, it created replacement events for all eight cancelled events, including dates whose replacement times were ambiguous. This failure instantiated a failure-pattern arm capturing that batch event replacement failed to distinguish ambiguous from unambiguous dates. The first repair at Iteration 24 added an executor hook immediately before the clarification message so that valid replacements could be created while ambiguous dates were left for clarification. However, by the time this hook ran, the invalid replacement events had already been registered during calendar creation. Because the same weakness remained represented as a failure-pattern arm, it was retargeted at Iteration 25. The second repair moved the executor hook to calendar-event pre-registration, allowing invalid replacements to be blocked before creation while retaining the three valid replacements; the scenario then passed.

In the scenario-arm variant, `4bytkx` was also successfully repaired at Iteration 31, but this repair was later lost as subsequent harness updates accumulated. The scenario-level representation maintained 75 independent arms, more than twice the maximum of 35 failure-pattern arms, making individual scenarios less likely to be revisited under the same optimization budget. Moreover, a scenario

arm identifies an executable instance but does not explicitly represent which weakness or repair in the current harness that scenario corresponds to. After the repair disappeared, `4bytkx` was not re-executed for more than ten iterations, leaving the scorer without fresh evidence about its status under the changed harness. It instead concluded that "**This arm was recently fixed**," causing its severity to drop from 0.80 to 0.15 and its rank from 6th to 52nd. This deprioritization further reduced the likelihood of revisiting the scenario, and it was not selected again before optimization ended.

Together, these cases expose complementary limitations of predefined arm definitions. Category arms are too coarse: heterogeneous weaknesses share a single optimization state, so successes on unrelated scenarios can obscure a weakness that remains unresolved. Scenario arms are precise in identifying executable instances, but their optimization state is fragmented across many arms and does not explicitly encode the relationship between a scenario, its underlying weakness, and the repairs retained in the current harness. Under a limited optimization budget, this makes frequent re-observation difficult and can leave arm scores based on stale evidence as the harness evolves. Failure-pattern arms instead represent observed harness weaknesses as explicit optimization targets, providing a granularity that is specific enough to isolate repair opportunities while compact enough to keep their status trackable. This allows unsuccessful or subsequently lost repairs to remain available for retargeting and further refinement.

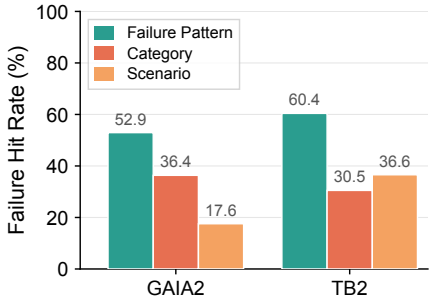


Figure 8: **Failure hit rate by arm definition.** Fraction of optimization iterations whose sampled mini-batch contains at least one scenario with an unresolved failure. Budgets are 1,400 rollouts for GAIA2 and 490 for TB2.

Failure Hit Rate. Figure 8 measures how often each arm definition directs an optimization iteration toward a scenario whose failure remains unresolved. A scenario is considered *unresolved* if it most recently failed or if a repair that previously made it pass is no longer retained in the current harness. An iteration is a *failure hit* when its sampled mini-batch contains at least one such scenario, and the *failure hit rate* is the fraction of iterations that satisfy this condition. We use equal budgets of 1,400 rollouts on GAIA2 and 490 rollouts on TB2.

On GAIA2, failure-pattern arms achieve a failure hit rate of **52.9%**, compared with 36.4% for category arms and 17.6% for scenario arms. On TB2, failure-pattern arms achieve **60.4%**, compared with 30.5% for category arms and 36.6% for scenario arms. Failure-pattern arms are induced from observed failures, so their pulls directly target weaknesses the harness has already exhibited. Category arms instead mix multiple scenarios under one label, while scenario arms spread optimization across the full training set. As a result, failure-pattern arms devote a larger share of the optimization budget to weaknesses that remain unresolved as the harness evolves.

H DISCOVERED FAILURE PATTERNS

Table 9: Failure patterns induced by ActiveSaddler on GAIA2

ID	Iter.	Induced Failure Pattern	Supporting Scenarios
1	1	No-attachment path defaults serialize as null instead of the oracle's empty path string	8hgfug, aa11lh, tnxtee
2	1	Over-specific event notification wording diverges from concise oracle-style message content	8hgfug
3	3	Affirmative event-start email reply content is not canonicalized to the required acknowledgement structure	tnxtee
4	4	Multi-ambiguity user clarifications omit explicit questions for all unresolved decision dimensions	2qw4nk, qrxrry
5	4	Time-sensitive repeated actions drift against deadlines and stop before the required count is complete	71j6lf
6	4	Lookup tools reject raw identifiers that downstream action tools accept, causing resolution loops in deadline-bound tasks	71j6lf
7	6	Partial filesystem discovery causes incomplete multi-file deletion for filename-prefix tasks	xcv49b
8	6	Incomplete source enumeration before cross-app aggregation produces wrong numeric answers	anc8y4, bv0gwh, m1momj, mp5r62
9	7	Relative same-time-tomorrow reschedules anchor to the referenced event date instead of the task receipt date	2qw4nk
10	10	Ambiguous contact-dependent state changes execute before identity clarification	dpbafp, rkdpfj
11	10	Duplicate first-name contact clarification uses downstream app matches instead of source contact ambiguity	dpbafp, rkdpfj
12	15	Missing target event replacement ambiguity produces over-broad clarification messages	b6v3y8
13	15	Explicit separate-email requests are collapsed into one multi-recipient email event	b6v3y8
14	20	Property-viewing confirmation emails omit canonical subject and full property address details	rkdpfj
15	21	Picky-recipient gift shopping tasks add cart items before resolving ambiguous product variants	kao12j
16	21	Direct personal gift-notification messages omit required recipient first-name greeting before event registration	kao12j
17	24	Batch event replacement scheduling fails to partition ambiguous and unambiguous dates	4bytkx
18	26	Ambiguous existing calendar event references are guessed and mutated instead of deferred for clarification	lxe62a
19	26	Email composition drops one purpose from compound thank-you-and-materials requests	lxe62a
20	26	Unresolved ambiguity in one subtask blocks independent required actions	ja2axd
21	28	Unrequested optional calendar metadata on indirectly described events	8c9am2
22	28	Premature notification waiting blocks required user update at multi-phase task boundaries	8c9am2, fqdxfe
23	28	Trace-less execution timeout in broad multi-step workflows	3rljy6
24	29	Follow-up availability notifications trigger unrequested reply confirmations in multi-phase workflows	fqdxfe
25	33	Notification-summary reliance misses authoritative same-thread follow-up state before conditional downstream actions	fqdxfe
26	34	Premature downstream actions before required checkpoint notifications in conditional workflows	fqdxfe
27	35	Global shopping option-term narrowing undercounts per-product variant ambiguity in clarification prompts	kao12j
28	40	Duplicate-contact clarification canonicalization drops full-name options and task-specific action context	dpbafp
29	40	Filtered save-only selection is misread as permission to remove existing saved items	dpbafp
30	40	Stale contact addresses are trusted for location-dependent service booking despite relocation uncertainty	r4pl5k
31	42	Correct cab-conflict handling leaves replacement ride writes with non-canonical location and exact-time arguments	096dyu
32	42	Phase-blind write canonicalization corrupts preliminary multi-phase scheduling outreach before reply context exists	i6hy8j
33	42	Schedule proposal selection ignores same-category calendar commitments when applying availability constraints	i6hy8j
34	42	Descriptor-based contact lookup lacks profile-field coverage and triggers fragile downstream message composition	n9ewlm
35	46	Timed reply-gathering tasks proceed before the collection window is complete	tf35kd

Table 10: Failure patterns induced by ActiveSaddler on Terminal-Bench 2.0

ID	Iter.	Induced Failure Pattern	Supporting Scenarios
1	1	Premature completion on constrained file edits without rule-aware final validation	`overfull-hbox`
2	1	Long-running probabilistic computation lacks staged dependency setup, diagnostic runs, and progress-aware output validation	`mcmc-sampling-stan`
3	2	Security sanitizer tasks finalize after weak smoke tests without adversarial and clean-preservation validation	`filter-js-from-html`
4	4	Binary extraction solutions finalize after superficial output smoke tests without independent address-space convention validation	`extract-elf`
5	4	Agent self-validation leaves stale runtime artifacts that change verifier timing and hide required fresh-run output	`make-doom-for-mips`
6	4	Completion-only validation gates do not help long build/debug tasks that time out before attempting finalization	`make-doom-for-mips`
7	6	Missing role-by-role exact source-component validation for multi-part generated artifacts	`protein-assembly`
8	9	Superficial VM/frame validators accept nonsemantic dummy outputs as success	`make-doom-for-mips`, `make-mips-interpreter`
9	10	Unbounded manual ML artifact creation lacks executable training and metric validation gates	`train-fasttext`
10	10	VM/frame validators conflate runtime display geometry with saved artifact geometry	`make-doom-for-mips`, `make-mips-interpreter`
11	12	Temporal leaderboard queries use current live data with date filters instead of reconstructing historical snapshots	`mteb-leaderboard`
12	16	Missing bounded decoder and completion gate for text embedded in structured geometry artifacts	`gcode-to-text`
13	23	Oversized inline setup payloads exceed exec argument limits before the task loop	`bn-fit-modify`, `constraints-scheduling`, `gcode-to-text`, `polyglot-c-py`
14	23	Transient validation artifacts left in verifier-checked output directories violate clean artifact contracts	`polyglot-c-py`
15	41	Unsafe top-level shell exits in persistent terminal sessions cause harness-level crashes before recovery	`make-doom-for-mips`

Table 9 and Table 10 present the failure-pattern arms discovered during the GAIA2 and Terminal-Bench 2.0 optimization runs, respectively. These patterns are induced from diagnosed execution failures during optimization rather than predefined by a failure taxonomy. For each pattern, we report its discovery iteration and the set of supporting scenarios. The induced representation exhibits a many-to-many relationship between scenarios and failure patterns. Some patterns consolidate the same weakness across multiple scenarios (e.g., GAIA2 P8 and TB2 P13 each span four scenarios/tasks), while a single scenario can expose multiple distinct patterns (e.g., `dpbafp` in GAIA2 and `make-doom-for-mips` in TB2). Appendix K details the two-stage extraction and normalization procedure. The complete `SKILL.md` files for `symptom-extract` and `symptom-normalize` are provided in the anonymous repository linked in the Reproducibility Statement.

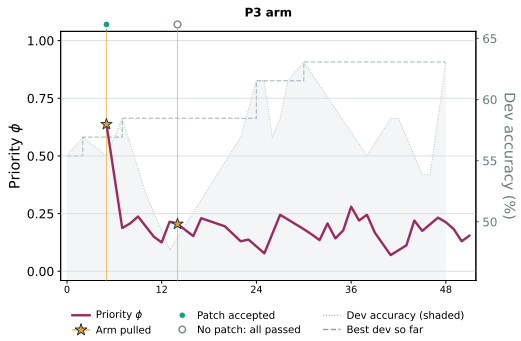


Figure 9: **Case study of a resolved weakness (P3 of GAIA2).** After an accepted patch at Iteration 5, the arm's priority drops sharply. A later pull finds all supporting scenarios successful and produces no further patch, after which the arm remains at low priority.

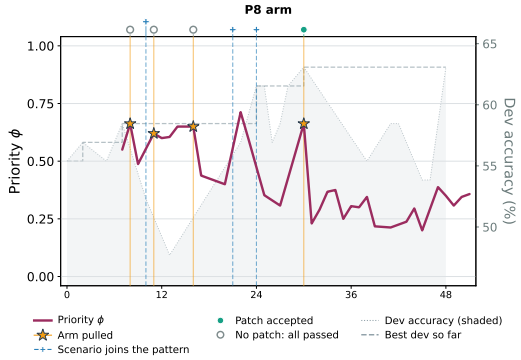


Figure 10: **Case study of a recurring weakness (P8 of GAIA2).** Early pulls produce no patch because the associated scenarios succeed when re-executed, yet the arm remains highly prioritized. The same failure pattern later appears in additional scenarios, providing new evidence that the weakness remains active; a subsequent pull produces an accepted patch. Cross markers indicate scenarios newly associated with P8.

I Additional Analysis for RQ2

Case Studies. To illustrate how LLM-based arm scoring responds to evolving optimization evidence, Figure 9 and Figure 10 show two representative failure-pattern arms with contrasting trajectories. For P3, the first pull at Iteration 5 produces an accepted patch, after which the arm's priority drops sharply. A later revisit finds that all supporting scenarios pass and requires no additional patch, providing further evidence that the corresponding weakness has been resolved. The arm consequently remains at low priority, allowing optimization budget to shift toward other weaknesses.

P8 exhibits the opposite behavior. Its first three pulls produce no patch because the associated scenarios pass when re-executed, but these transient successes provide limited evidence that the underlying weakness has been resolved: agent executions are stochastic, and no patch directly addressing the failure pattern has yet been applied. Accordingly, the scorer continues to assign substantial priority to P8. The same failure pattern subsequently appears in additional scenarios (Figure 10), strengthening the evidence that the weakness remains relevant beyond its originally observed cases. P8 is eventually selected again, and the resulting accepted patch increases the best development-set accuracy from 61.5% to 63.1%.

Together, these cases illustrate how arm scoring distinguishes evidence of resolution from evidence of continued relevance. A successfully repaired weakness is deprioritized, whereas a recurring weakness can retain or regain priority as new evidence accumulates. The following analyses examine how this evidence-dependent reprioritization manifests across the full arm pool.

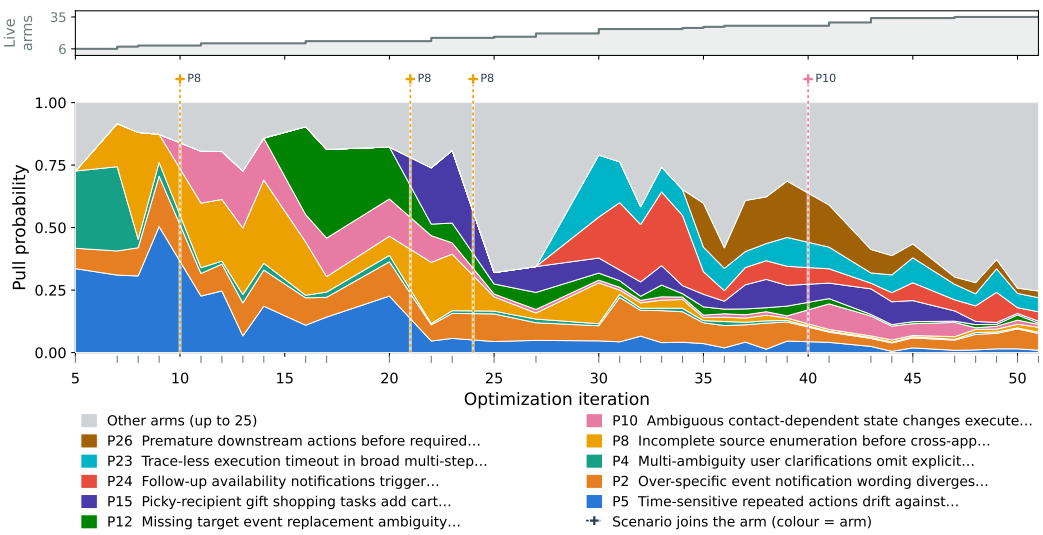


Figure 11: **Evolution of pull-probability mass across failure-pattern arms.** The ten arms with the highest average pull probability over the run are shown individually, while the remaining 25 arms are aggregated into *Other arms*. The upper panel shows the number of instantiated arms, which grows from 6 to 35 as new failure patterns are discovered. Pull probabilities are shown only at the 34 iterations where arm selection is performed.

Evolution of Arm Priorities. To further examine the curriculum-level dynamics summarized in Figure 3a, we analyze the pull-probability trajectories of all failure-pattern arms throughout the GAIA2 optimization run. Arm scores and pull probabilities are recomputed only when the curriculum selects an instantiated arm for optimization. Accordingly, Figure 11 covers the 34 arm-selection iterations between Iterations 1 and 51; iterations without arm selection are omitted. These decisions are made over a continually evolving target space: the number of instantiated failure-pattern arms grows from 6 to 35 as previously unseen failure patterns are discovered. Thus, the curriculum must continually reassess priorities while both the harness and the set of observed weaknesses evolve.

We first examine how probability mass is distributed according to the latest observed outcomes of each arm's supporting scenarios. As optimization proceeds, previously observed weaknesses may be addressed such that the scenarios supporting an arm subsequently succeed when re-evaluated. At Iteration 51, 24 of the 35 arms (68.6%) have had all of their supporting scenarios re-evaluated after association with the arm, with every such scenario succeeding in its latest valid evaluation. Despite comprising more than two thirds of the arm pool, these arms account for only 26.2% of the total pull-probability mass. For reference, under uniform allocation, they would receive 68.6% of the probability mass, making their observed share only 0.38× this uniform baseline. The other 11 arms include at least one supporting scenario without a subsequent successful evaluation and together receive 73.8% of the probability mass, compared with 31.4% under uniform allocation, or 2.35× their uniform share. These results indicate that, as optimization progresses and previously observed weaknesses are successfully addressed on their supporting scenarios, their corresponding arms tend to receive progressively less pull probability.

However, arm priorities can also increase when new evidence indicates that a previously observed weakness remains active. The cross markers for P8 and P10 in Figure 11 indicate iterations at which a failure pattern already represented by an existing arm is observed again in an additional scenario, which is then added to that arm's supporting scenario set. For P8, these support-set expansions are followed by an average increase of 4.8 percentage points in pull probability, while the corresponding P10 event is followed by a 9.4 percentage-point increase. These cases show that observing the same failure pattern in additional scenarios can increase the priority of an existing arm by providing new evidence that the corresponding weakness remains relevant.

Taken together, these observations show that the priority assigned to a failure-pattern arm changes with the evidence accumulated about the corresponding weakness. Arms receive less probability as



Figure 12: **Pull-probability trajectories for all failure-pattern arms.** Rows show all 35 discovered arms ordered by instantiation time, and columns correspond to the 34 iterations in which arm selection is performed. Cell intensity denotes pull probability, orange markers indicate the arm sampled at each iteration, and gray cells denote iterations before an arm entered the pool.

their supporting scenarios subsequently succeed, while new observations of the same failure pattern across additional scenarios can increase their priority again. This evidence-dependent prioritization explains the continual redistribution of probability mass across failure-pattern arms observed in Figure 11.

Stochastic Selection over the Expanding Arm Pool. Figure 12 complements the aggregate probability-mass view in Figure 11 by showing the pull probability and realized selection of every failure-pattern arm at each arm-selection step. Across 34 pulls, the sampler selects 19 of the 35 discovered failure patterns. The selected arm has a median priority rank of third, with 62% of pulls drawn from the top three arms, while selections extend as far as rank 20. Thus, the stochastic selection rule strongly favors arms assigned high current priority without reducing the curriculum to greedy selection of the highest-scoring arm. Lower-ranked arms retain opportunities to be sampled as their relative priorities change over optimization.

The gray staircase also makes explicit that these selections are made over an arm pool that expands throughout the run. Consequently, an arm's priority is always relative to the set of weaknesses known at that point in optimization rather than to a fixed collection of targets defined in advance. As new failure patterns enter the pool and the evidence associated with existing arms changes, the set of arms receiving substantial probability and the arms actually selected both change over time. Together with the probability-mass analysis in Figure 11, this shows that LLM-based arm scoring produces a focused but non-greedy curriculum: most optimization opportunities are directed toward currently high-priority weaknesses, while stochastic selection preserves the possibility of targeting other arms as their relative optimization value evolves.

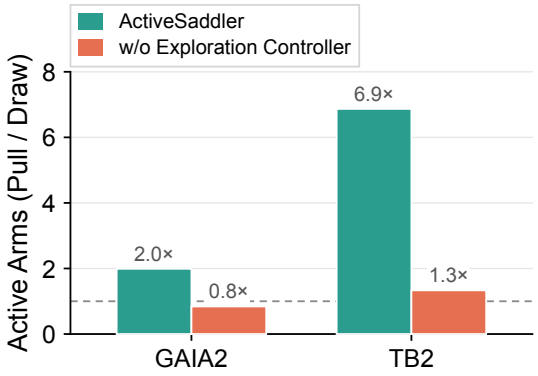


Figure 13: **Dependence of exploration decisions on unresolved weaknesses.** Ratio of the average number of unresolved arms at PULL to DRAW decisions. Values above one indicate more unresolved weaknesses at PULL, while values near one indicate little difference between the two decision types.

J ADDITIONAL ANALYSIS FOR RQ3

Dependence of Exploration Decisions on Unresolved Weaknesses. To quantify whether exploration decisions respond to unresolved weaknesses, we compare how many arms remain unresolved at PULL and DRAW decisions. An arm is considered unresolved when its associated weakness remains unsolved under the current harness. We report the ratio of the average number of unresolved arms at PULL to that at DRAW: values above one indicate that PULL tends to occur when more unresolved weaknesses remain, whereas values near one indicate little difference between the two decision types.

As shown in Figure 13, ActiveSaddler has 2.0× as many unresolved arms at PULL as at DRAW decisions on GAIA2 (1.62 vs. 0.81) and 6.9× as many on Terminal-Bench 2.0 (0.76 vs. 0.11). In contrast, the fixed exploration schedule yields ratios close to one: 0.8× on GAIA2 (0.21 vs. 0.25) and 1.3× on Terminal-Bench 2.0 (1.14 vs. 0.86), indicating that its PULL and DRAW decisions are largely insensitive to how many unresolved arms remain. Thus, ActiveSaddler tends to continue repair when more unresolved weaknesses remain and to explore when fewer remain, whereas the fixed schedule shows little such dependence on the current optimization state.

K INSTRUCTIONS USED IN ACTIVESADDLER

To make the curriculum decisions in ActiveSaddler reproducible and interpretable, we provide the prompt templates used by the three curriculum-specific LLM components introduced in the main method: the Failure-Pattern Extractor, Arm Prioritizer, and Exploration Controller. Each prompt provides the optimization context required for its corresponding decision and constrains the agent to a focused analysis procedure.

The **Failure Pattern Extraction Prompt** (Figure 14) converts failed execution evidence into reusable failure-pattern arms. It first abstracts each failure independently into a candidate symptom without consulting the existing arm set, and then normalizes the extracted candidates by linking them to existing atomic patterns, composing multiple patterns when needed, or instantiating new arms.

The **Arm Scoring Prompt** (Figure 15) implements the Arm Prioritizer by estimating a learning-progress score $\phi_t(a)$ for every instantiated failure-pattern arm. Using the latest optimization history, the agent assesses severity, fixability, breadth, and side-effect risk. The resulting scores are used by the stochastic arm-selection rule to prioritize the next optimization target.

The **Unseen Scenario Exploration Prompt** (Figure 16) implements the Exploration Controller. Given the current optimization context, the agent decides whether to continue optimizing a known failure-pattern arm or explore scenarios that may reveal previously undiscovered weaknesses, based on whether a worthwhile unresolved target remains among the known arms and the remaining value of further exploration.

Failure Pattern Extraction 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/<name>/SKILL.md`
in the current worktree.
| Order | Skill | When | Why |
|-----|-----|----|---|
| 1 | `symptom-extract` | For each failed scenario | Symptom extraction from
diagnosis/reflection results -- generates candidate symptoms |
| 2 | `symptom-normalize` | After extracting all candidates | 3-way
normalization against existing patterns -- links, composes, or creates
patterns to maintain atomic pattern integrity |
> Enforcement:  You MUST process every failed scenario.  Do NOT skip the
extraction step for any failed scenario.  During extraction (Skill 1:
`symptom-extract`), do NOT examine existing patterns; consult them only during
normalization (Skill 2:  `symptom-normalize`).  This separation is required to
avoid anchoring bias.
## Goal
Extract failure patterns from diagnosis and reflection results, then tag
each failed (harness, trace, scenario) tuple with the identified pattern(s).
Failure patterns are symptom-level abstractions -- more abstract than
individual root causes, but specific enough to distinguish different failure
types.  They are the arms of the curriculum sampler.
At each outer-loop iteration, each arm (i.e., failure pattern) is assigned a
learning-progress score estimating the expected improvement in the current
harness's population performance that would result from selecting the arm and
successfully addressing its corresponding failure pattern in the next
optimization step.
## Context
- Iteration:  {iteration}
- Candidate:  C{candidate_idx}
- Current worktree:  `{worktree_path}`
- Session root:  `{session_root}`
- Pre-patch trace dir:  `{before_output_dir}`
- Post-patch trace dir:  `{after_output_dir}`
## Failed Scenarios to Process
### Pre-patch failures (from diagnosis)
These scenarios failed BEFORE any patch was applied.  Each entry provides the
batch-level diagnosis-and-patching step diagnosis and detailed reasoning-file
location from before
patching, followed by the reviewed underlying root cause and post-hoc
analysis of how the patch affected that cause.
{pre_patch_failures}
### Post-patch failures (from reflection)
These scenarios STILL FAILED or REGRESSED after the patch was applied.
Each entry provides the status change, the batch-level patch approach,
and the reviewed root cause and post-hoc patch-effect explanation.  For
`still_failing`, the root cause is the remaining or revised failure cause; for
`regressed`, it is the newly observed regression cause or failure mechanism.
{post_patch_failures}
## Workflow
### 1.  Extract candidate symptoms
For each failed scenario listed above:
1.  For pre-patch failures, review the diagnosis/reasoning and reflection
step's reviewed cause and patch-effect analysis.  For post-patch failures,
also
```

(continued)

```
use the status change and patch approach to distinguish a remaining
cause from a newly introduced regression mechanism.
2. Follow the `symptom-extract` skill to generate a candidate symptom
label. Do NOT look at existing patterns during this step.
3. Record every candidate symptom and its associated root cause in
`symptom_candidates.md` at the working directory root.
Complete `symptom_candidates.md` for all failures before moving to
normalization.
### 2. Normalize against existing patterns
After all candidate symptoms are extracted:
1. Read `symptom_candidates.md` from the working directory root.
2. Run `pattern list` to retrieve all existing patterns.
3. For each candidate symptom in the file, follow the `symptom-normalize`
skill to
perform a 3-way judgment:
- (a) Same: the candidate matches an existing pattern → link to it
- (b) Composition: the candidate is a combination of existing atomic
patterns → tag with multiple existing patterns
- (c) New: genuinely novel failure type → register as a new pattern
### 3. Tag tuples
For each resolved pattern assignment:
1. Register new patterns if needed:
"'
pattern register -label "DESCRIPTIVE SYMPTOM LABEL"
"'
2. Tag the (harness, trace, scenario) tuple:
"'
pattern tag -pattern-id <id> -harness {candidate_idx} -trace <dir> -scenario
<sid> -root-cause "..."
"'
For compositions, repeat `-pattern-id` for each atomic pattern.
> IMPORTANT: Always use `-harness {candidate_idx}` (the current
> candidate) for ALL tagging commands -- both pre-patch and post-patch
> failures. The `-trace` directory distinguishes before/after. The
> harness identifies which iteration this observation belongs to.
> Note: The failure-pattern extraction step only needs to execute `pattern
register` and `pattern tag`. The agent does not need to calculate or record
observation values separately. For each previously known pattern, the outer
loop automatically calls `PatternRegistry.observe()` with the scenarios
associated with that pattern and evaluated in the mini-batch, the subset still
tagged after patching, and the resulting activity fraction.
## Important Notes
- Root cause is preserved as evidence, not used directly as the pattern.
The pattern label should be an abstraction that generalizes across
multiple scenarios.
- Patterns should be atomic -- a single patch should be able to fix one
pattern. Complex failures are expressed as compositions (multi-tagging).
- Do NOT over-merge: if two failures have different root causes, they
should be different patterns even if they look superficially similar.
- Do NOT over-split: if the same behavioral issue manifests with minor
surface variations, it's the same pattern.
```

Figure 14: Failure Pattern Extraction Prompt: the agent independently abstracts failed execution evidence into candidate symptoms and then normalizes them against the existing arm set by linking to existing atomic patterns, composing multiple patterns when needed, or instantiating new failure-pattern arms.

Arm Scoring 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/<name>/SKILL.md`
in the current worktree.
| Order | Skill | When | Why |
|-----|-----|----|---|
| 1 | `history-analysis` | Before any scoring (Step 1) | Understand the
Prepared harness and evolution history |
| 2 | `progress-scoring` | For every candidate pattern (Steps 3-5) | Estimate
each arm's cardinal learning-progress score |
## Goal
For every candidate failure-pattern arm, estimate a cardinal
learning-progress score φ ∈ [0, 1] representing the expected improvement in
the prepared harness's population performance from allocating the next
optimization step to that arm.  The estimate should reflect the likely outcome
of executing the arm's supporting scenarios and attempting to address the
harness weakness represented by the pattern, including both the potential
performance gain and the probability of obtaining a broadly effective,
dev-preserving update.
Base each score on the prepared harness, the arm's complete execution and
optimization history, prior dev-set outcomes, and the available evidence about
the pattern's severity, fixability, breadth, and risk of side effects.  Record
every cardinal score with `pattern rate`.
This is a read-only analysis session:  do NOT modify the codebase.  Only
inspect the harness, failure patterns, supporting scenarios, and optimization
history, then record the scores.
## Context
- Iteration:  {iteration}
- Prepared harness:  C{prepared_candidate_idx}
- Prepared worktree:  `{prepared_worktree_path}`
- Prepared commit:  `{prepared_commit}`
- Provisional parent:  C{provisional_parent_idx}
(`{provisional_parent_commit}`)
- Selection JSON: `{selection_session_json_path}`
- Session root:  `{session_root}`
### Prepared Harness Summary
{prepared_harness_summary}
## Candidate Failure Patterns (arms to score)
Score every pattern below.
{candidate_patterns}
### Candidate Table Columns
- `Pattern`:  Stable failure-pattern ID used by the `pattern` and `evo-dag`
CLIs.
- `Activity`:  Rested EMA of the raw observations, seeded at `1.00`.  This is
a
mechanical severity reference, not the Agent learning-progress score φ.
- `Observations`:  Complete per-iteration observation history.  Each entry
uses
`iteration:tagged/evaluated (active=...), tagged=[...], evaluated=[...]`.
`evaluated` lists the scenarios associated with this pattern that were
evaluated in that iteration's mini-batch, and `tagged` lists the subset that
was still tagged with the pattern after patching.  `active` is the
corresponding
```

(continued)

`tagged/evaluated` fraction. For legacy observations whose scenario IDs were not recorded, the counts appear as `?/¿ and both lists as `(not recorded)`; `(none)` means the arm has not been selected again since the pattern was most recently observed in a scenario, so no subsequent activity observation has been recorded.
- `#Scen`: Number of unique scenarios currently tagged with the pattern.
- `LastObs`: Most recent iteration in which the arm was selected and an activity observation was recorded, or `0` if the arm has not been selected again since the pattern was most recently observed in a scenario.
- `Label`: Symptom-level failure-pattern description.
- `Scenarios`: Complete list of scenario IDs currently owned by the arm.

Workflow

1. Understand the current harness (history-analysis)

Run the `history-analysis` skill. Inspect the current prepared node with `evo-dag show node {prepared_candidate_idx}` and its provisional parent with `evo-dag show node {provisional_parent_idx}`. Read the actual prepared-harness change
with `git diff {provisional_parent_commit} {prepared_commit}`.
You need this context to assess all four scoring axes: *severity* (whether the failure pattern is likely still active under the prepared harness), *fixability* (whether and where a patch can address it), *breadth* (how broadly a successful repair may generalize), and *side-effect risk* (which components or behaviors a patch may affect or regress).

2. Review prior pulls for each candidate arm

Read the histories of all arms that have previously been selected:
```bash
pattern history
```
With no `-pattern-id`, the command shows every historically pulled arm. If the full output is too large to review efficiently, you may optionally narrow it to a subset in one call with `pattern history -pattern-id <id1> <id2>`, or use `-last-k <K>` to show only the K most recent entries for each selected arm during initial triage. These options are convenience filters, not substitutes for complete review: inspect an arm's full history before rating it whenever older attempts could change your judgment. The output distinguishes patched attempts, all-pass skips, and failed attempts and includes diagnoses, patch intents, dev-set impact, and per-scenario reflections.

3. Inspect each candidate pattern

For every pattern in the table:
- `pattern show <pattern_id>` -- label, tagged scenarios, evidence (root causes).
- `evo-dag show scenario <sid>` -- for each of the pattern's scenarios, read its
failure history and relate it to the current harness change.

4. Estimate learning progress (progress-scoring)

Follow the `progress-scoring` skill. For each pattern, judge the four considerations (severity, fixability, breadth, side-effect), then rate **all four axes carefully** in [0, 1]. The final score ϕ is the mean of those four axes with side-effect inverted --
`(severity + fixability + breadth + (1 - side_effect)) / 4`. Your per-axis ratings (not a single holistic judgment) determine the result.
Use the per-arm pull history as primary evidence, and weigh the dev-set, not just the mini-batch. A scenario merely passing the mini-batch in a recent pull
does NOT by itself mean the pattern is resolved -- learning progress is measured
on the **dev/held-out** distribution:
- If a prior patch fixed the scenario but **dropped dev-set accuracy** (so it was

(continued)

```
rejected and reverted), a better dev-preserving fix is still needed →
progress
may remain high.
- If the scenario is genuinely resolved (passing with no dev headroom left),
or
its failure is intrinsically unfixable without regressing dev → progress is
low.
- If recent pulls show the scenarios repeatedly passing, lower the score,
**but
do not force it to 0** -- the pass may be intermittent, and the sampler's
minimum-probability floor still re-checks low-scored arms periodically to
catch
silent regressions.
### 5. Record every score
> Reminder: φ is computed from your four axes (see step 4). `-side-effect`
is a RISK -- higher means worse.
For each candidate pattern:
"'bash
pattern rate -pattern-id <id> \
-severity <s> -fixability <f> -breadth <b> -side-effect <e> \
-rationale "..."
"'
CRITICAL: rate every pattern in the table before finishing. An un-rated
pattern is treated as low priority (score 0.0) -- if you skip one it gets
deprioritized rather than reflecting your judgment, so do not leave any
un-rated.
## What Happens Next
The outer loop samples one pattern with probability proportional to `softmax(φ
/ τ)` and sends its scenarios to the diagnosis-and-patching step.
```

Figure 15: Arm Scoring Prompt: the agent estimates a learning-progress score $\phi_t(a)$ for every instantiated failure-pattern arm by considering severity, fixability, breadth, and side-effect risk. The resulting scores define the stochastic prioritization of known arms.

Unseen Scenario Exploration Prompt

```
## Mandatory Skills
You MUST read and follow the SKILL.md for each skill listed below.
Do NOT skip or summarize the skill -- execute the full procedure described in
it. This skill is installed at `.claude/skills/<name>/SKILL.md` in the
current
worktree.
| Order | Skill | When | Why |
|-----|-----|----|---|
| 1 | `history-analysis` | Before the decision (Step 1) | Understand the
Prepared harness and evolution history |
## Goal
Decide whether this iteration should exploit a known failure-pattern arm
(PULL) or explore unseen scenarios that may reveal a new failure type (DRAW).
Record exactly one decision with `pattern decide`.
This is an exploration-exploitation trade-off in arm space:
- PULL spends this iteration on an already-discovered pattern. arm-scoring
step
then scores every known pattern so the sampler can choose an arm to repair.
- DRAW probes scenarios that have never been executed. This may reveal a
failure pattern not yet represented by the known arms; Arm scoring is skipped.
```

(continued)

```
There is no fixed formula or threshold for this decision. Do not decide
mechanically from `|P_t|`, `|U_t|`, Activity, or any single statistic. Weigh
all
available evidence and make one holistic decision.
Here, `P_t` is the set of failure patterns discovered by iteration `t`, and
`U_t` is the set of scenarios still unseen at that iteration; `|...|` denotes
the size of a set.
This is a read-only analysis session: do NOT modify the codebase and do NOT
rate arms.
## Context
- Iteration: {iteration}
- Prepared harness: C{prepared_candidate_idx}
- Prepared worktree: `{prepared_worktree_path}`
- Prepared commit: `{prepared_commit}`
- Provisional parent: C{provisional_parent_idx}
(`{provisional_parent_commit}`)
- Selection JSON: `{selection_session_json_path}`
- Session root: `{session_root}`
- Discovered failure patterns: |P_t| = {num_arms}
- Unseen scenarios remaining: |U_t| = {unseen_pool_size}
### Prepared Harness Summary
{prepared_harness_summary}
## Candidate Failure Patterns (known arms)
Use the patterns below to judge whether a worthwhile known target exists.
{candidate_patterns}
### Candidate Table Columns
- `Pattern`: Stable failure-pattern ID used by the `pattern` and `evo-dag`
CLIs.
- `Activity`: Rested EMA of the raw observations, seeded at `1.00`. This is
a
mechanical severity reference, not the Agent learning-progress score φ.
- `Observations`: Complete per-iteration observation history. Each entry
uses
`iteration:tagged/evaluated (active=...), tagged=[...], evaluated=[...]`.
`evaluated` lists the scenarios associated with this pattern that were
evaluated in that iteration's mini-batch, and `tagged` lists the subset that
was still tagged with the pattern after patching. `active` is the
corresponding
`tagged/evaluated` fraction. For legacy observations whose scenario IDs were
not recorded, the counts appear as `?/¿ and both lists as `(not recorded)`;
`(none)` means the arm has not been selected again since the pattern was most
recently observed in a scenario, so no subsequent activity observation has
been recorded.
- `#Scen`: Number of unique scenarios currently tagged with the pattern.
- `LastObs`: Most recent iteration in which the arm was selected and an
activity observation was recorded, or `0` if the arm has not been selected
again since the pattern was most recently observed in a scenario.
- `Label`: Symptom-level failure-pattern description.
- `Scenarios`: Complete list of scenario IDs currently owned by the arm.
## Workflow
### 1. Understand the current harness (history-analysis)
Run the `history-analysis` skill. Inspect the current prepared node with
`evo-dag show node {prepared_candidate_idx}` and its provisional parent with
`evo-dag show node {provisional_parent_idx}`. Read the actual
prepared-harness change
with `git diff {provisional_parent_commit} {prepared_commit}`.
Use this context to judge whether known-arm evidence still applies to the
```

(continued)

````
prepared harness and whether its weaknesses remain reachable by another patch.
### 2.  Review prior pulls for each candidate arm
Read the histories of all arms that have previously been selected:
"'bash
pattern history
"'
With no '-pattern-id', the command shows every historically pulled arm.  If
the
full output is too large for initial triage, narrow it with
'pattern history -pattern-id <id1> <id2>' or apply '-last-k <K>'
separately to each selected arm.  These are convenience filters, not
substitutes
for complete review when older attempts could change the PULL/DRAW decision.
The output distinguishes patched attempts, all-pass skips, and failed attempts
and includes patch approaches, dev-set impact, and per-scenario reflections.
### 3.  Inspect promising known patterns
Use 'pattern show <pattern_id>' for any pattern that appears impactful.  Read
its
label, tagged scenarios, root-cause evidence, and observations.  You do not
need
to assign a learning-progress score in this session; make a coarse judgment of
whether at least one strong known target exists.
### 4.  Weigh the PULL/DRAW considerations
Use all two lenses below.  They guide attention but do not form a formula.
1.  Is there a clearly worthwhile known pattern to fix?
A severe, plausibly fixable, broad pattern argues for PULL. If known patterns
look near-resolved, unreachable, or exhausted by repeated low-yield attempts,
diminishing returns argue for DRAW.
2.  How complete is coverage of the failure surface?
Few discovered arms and a large unseen pool suggest that known patterns do
not yet represent the harness's weaknesses.  A small or exhausted unseen pool
leaves little discovery value and favors PULL.
### 5.  Make and record one decision
The discovery value of DRAW is a prediction:  unseen scenario contents are not
known.  Do not over-explore late when a strong fix is available, and do not
over-exploit early when the failure surface is poorly mapped.
Record exactly one action:
"'bash
pattern decide -action pull \
-rationale "why a known pattern is the best use of this iteration"
# OR
pattern decide -action draw \
-rationale "why discovery is more valuable than fixing known patterns now"
"'
Do NOT call 'pattern rate' in this session.
## What Happens Next
- After PULL, arm scoring runs on the same prepared harness and scores every
arm.
- After DRAW, arm scoring is skipped and the outer loop samples unseen
scenarios.
````

Figure 16: Unseen Scenario Exploration Prompt: the agent decides whether to continue optimizing a known failure-pattern arm or explore scenarios that may reveal new failure patterns by weighing the value of unresolved known targets against the remaining value of further exploration.