PLUGINRSI: RECURSIVE IMPROVEMENT OF AGENT HARNESSES WITH REUSABLE PLUGINS

Yaorui Shi[1,2], **Yuchun Miao**[2,3], **Yuxin Chen**[2,4], **Jiayuan Zhang**[2], **Yueqing Sun**[2], **Xierui Song**[2,†], **Xiang Wang**[1], **An Zhang**[1,†]

[1]University of Science and Technology of China [2]Meituan
[3]Wuhan University [4]National University of Singapore

[†]Corresponding authors: guqi03@meituan.com, an_zhang@ustc.edu.cn

arXiv:2609.32423v2 [cs.AI] 18 Oct 2026

ABSTRACT

The harness surrounding a language model is a central determinant of agent performance. Recent methods optimize harnesses by searching over complete programs, where individual mechanisms are difficult to isolate and reuse. We introduce PluginRSI, which represents a harness as a composition of atomized plugins and organizes harness evolution around these plugins. Individual plugins are improved independently and accumulated in a shared library, then recombined into new harnesses at each iteration. PluginRSI improves over existing harness optimization methods across software engineering, command-line interaction, and question-answering tasks. The resulting harnesses retain their advantage when transferred to other solver models without further optimization. The evolved plugin library accelerates subsequent optimization from the initial harness, which helps faster and higher convergence on unseen tasks. These results show that accumulating reusable mechanisms provides an effective basis for continued harness improvement.

Date: October 20, 2026
Homepage: https://syr-cn.github.io/PluginRSI/
Code: https://github.com/syr-cn/PluginRSI

Everything is a plugin.

—- *DeepSeek Harness*

1 INTRODUCTION

Recursive self-improvement aims to enable large language model (LLM) agents to become more capable as they continue to operate (Liu et al., 2026a; Ren et al., 2026; Duan et al., 2026). Such improvement can come from updating the underlying model parameters or refining the harness that organizes an agent's interaction mechanisms such as tools, roles and skills (Zhang et al., 2024; Meng et al., 2026; Ma et al., 2026). Harness optimization pursues the latter, allowing agent behavior to be adapted without the training cost of weight updates (Ning et al., 2026; Pan et al., 2026). Modern harnesses use plugin systems to implement new mechanisms as independent, reusable components, supporting continued improvement (DeepSeek-AI, 2026; Anthropic, 2025; OpenAI, 2026a).

Recent work explores recursive harness optimization by iteratively rewriting harness implementations based on execution feedback (Hu et al., 2025; Zhang et al., 2025; 2026b; Lee et al., 2026; Lin et al., 2026a; Karten et al., 2026; Zhang et al., 2026a). Despite its effectiveness, iterative harness rewriting faces two limitations. First, it limits the exploration of new mechanisms. Multiple mechanisms are coupled within the harness implementation, making it difficult to independently explore new mechanisms and incorporate their benefits into the harness. Consequently, optimization could repeatedly converge to similar designs. Second, valuable mechanisms are difficult to preserve and reuse. Rewriting can overwrite effective local mechanisms, preventing improvements from accumulating into reusable experience across iterations. Together, these limitations hinder continued improvement by restricting mechanism discovery and the accumulation of reusable mechanisms.

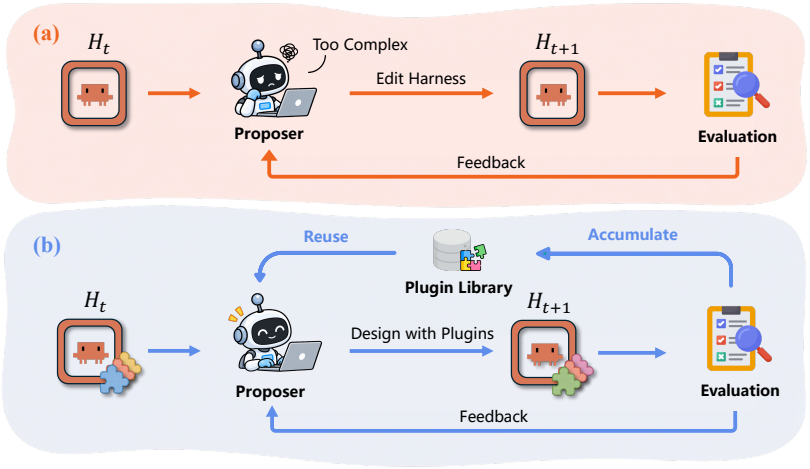


Figure 1: **Comparison of harness optimization paradigms.** **(a)** Prior approaches expose the complete harness H_t as the object of modification. **(b)** PluginRSI explores recursive optimization of plugin-parameterized harnesses, where each plugin encapsulates an independent mechanism that can be accumulated and reused.

We argue that harness evolution should center on discovering new mechanisms and reusing them across candidates, rather than repeatedly rewriting harness implementations. Inspired by this, we study the recursive optimization of **plugin-parameterized harnesses**. As shown in Figure 1, plugin-parameterized harnesses encapsulate individual mechanisms as plugins with standardized interfaces. This formulation allows individual mechanisms to be developed and their contributions evaluated while keeping the rest of the harness unchanged. A shared plugin library retains empirically beneficial implementations, allowing mechanisms discovered during optimization to accumulate and be reused to construct subsequent harness candidates.

Towards this end, we introduce **PluginRSI**, an algorithm that enables **r**ecursive harne**ss** **i**mprovement by accumulating and reusing **plug**ins. PluginRSI features a shared plugin library that accumulates beneficial harness-design mechanisms as plugins. Each harness consists of a set of plugins drawn from this library and a workflow that coordinates their execution. One optimization iteration begins with a plugin mutation step, where execution feedback guides changes to the plugins used by a candidate harness, exploring new mechanisms while keeping the remaining harness fixed. Plugin variants that improve harness performance are added to the library. Harness recomposition then selects plugins from the updated library and revises the workflow to assemble new harness candidates, building on the mechanisms discovered in earlier iterations.

We evaluate PluginRSI on software engineering, command-line interaction, and question-answering tasks. Across these settings, PluginRSI outperforms baseline methods. On SWE-bench Verified, it improves held-out resolve rates over Meta-Harness by 6–10 percentage points across two optimization backbones. The resulting harnesses retain their advantage across solver models without further optimization. Optimization curves show that PluginRSI finds strong harnesses with fewer solver rollouts. Ablations demonstrate the contributions of both plugin mutation and harness recomposition, with strong performance retained without the initial plugin library. The accumulated plugins support larger harness implementations with compact workflows and accelerate subsequent optimization. Reusing the evolved library from the initial harness reaches a held-out resolve rate of 69% after two evolution steps, compared with 65% without reuse.

2 Problem Formulation

2.1 Optimization Objective

A harness specifies how an agent interacts with its environment by defining and coordinating tools, skills, and other mechanisms. Together, a harness H and a policy model Θ define an agent $\Phi(\cdot \mid$

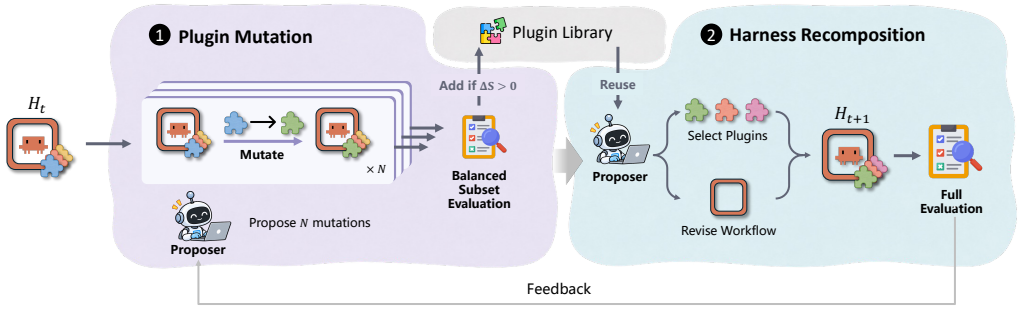


Figure 2: **Overview of PluginRSI. (1) Plugin mutation** explores individual plugin changes in parallel while keeping the remaining harness fixed. **(2) Harness recomposition** selects plugins from the updated library and revises the workflow to construct a new harness candidate.

$H, \Theta)$. Given a task x sampled from a task distribution $\mathcal{D}$, the agent generates a trajectory $\tau \sim \Phi(\cdot \mid x, H, \Theta)$, whose performance is evaluated by a reward function $r(\tau, x)$. We define the expected reward of a harness as

$$J(H) = \mathbb{E}_{x\sim\mathcal{D},\tau\sim\Phi(\cdot|x,H,\Theta)}\left[r(\tau, x)\right]. \tag{1}$$

Harness optimization seeks a harness $H^\star \in \arg\max_H J(H)$ that maximizes Eq. 1 while keeping the policy model Θ fixed.

2.2 Recursive Harness Optimization

Recursive harness optimization uses feedback from previously evaluated harnesses to guide the construction of new candidates. A separate proposer agent Φ_{proposer} examines a candidate's implementation and execution trajectories, reviews the agent's behavior, and proposes new candidates. Both the policy model Θ and the proposer Φ_{proposer} remain fixed throughout the search.

The search begins with an initial harness H_0. We consider a search procedure in which, at iteration t, the proposer generates a new candidate based on the previous best harness and its feedback:

$$H_t \sim \Phi_{\text{proposer}}\left(\cdot \mid \widehat{H}_t, \mathcal{T}_{\widehat{H}_t}\right), \tag{2}$$

where $\widehat{H}_t \in \arg\max_{H\in\{H_i\}_{i=0}^{t-1}} J(H)$ is the best harness found so far, and $\mathcal{T}_{\widehat{H}_t}$ denotes the execution trajectories collected for $\widehat{H}_t$. The agent defined by the proposed harness, $\Phi(\cdot \mid H_t, \Theta)$, is then evaluated on tasks sampled from $\mathcal{D}$, producing rewards and trajectories for subsequent iterations. After T iterations, the search returns $H^* \in \arg\max_{H\in\{H_i\}_{i=0}^{T}} J(H)$.

3 PluginRSI: Plugin-Oriented Harness Optimization

We introduce PluginRSI, a recursive harness optimization algorithm that operates over plugin-parameterized harnesses. The algorithm represents a harness through plugin implementations and the workflow that coordinates them, while maintaining a shared library that retains beneficial implementations across iterations (§3.1).

The optimization loop of PluginRSI is described in §3.2.1 and illustrated in Figure 2. At each iteration, PluginRSI first explores new mechanisms through plugin mutation, where individual plugins are selected, mutated, and evaluated while the remaining harness is kept fixed (§3.2.2). Plugin variants that yield empirical gains are accumulated in the shared plugin library. The algorithm then performs harness recomposition, which selects plugins from the updated library and revises the workflow that coordinates them to explore new harness designs (§3.2.3).

3.1 Plugin-Parameterized Harnesses

A plugin p is an executable implementation of a harness mechanism exposed through a standardized interface. This interface separates a mechanism's implementation from the workflow that invokes

it. In PluginRSI, each plugin consists of two components: (1) a YAML specification defining its interface and associated configuration, including input and output schemas, prompts, rubrics, and metadata, and (2) an executable implementation written in `Python` code. Further schema details are provided in Appendix J.

Building on this abstraction, we represent a plugin-parameterized harness as $H = \mathcal{H}(W, \mathcal{P})$, where $\mathcal{P}$ denotes the set of selected plugins and W denotes the workflow. The plugins in $\mathcal{P}$ implement the agent's roles, tools, skills, and memory mechanisms according to the schema. The workflow W determines when these plugins are invoked and how their outputs are used in subsequent interactions. The function $\mathcal{H}$ is a deterministic program that compiles W and $\mathcal{P}$ into an executable harness.

This representation supports two complementary optimization operations. First, individual plugin implementations can be discovered or refined while keeping the workflow and the remaining plugins fixed. Second, plugins can be selected from a plugin library and the workflow revised to coordinate their execution. These operations correspond to the plugin mutation and harness recomposition stages of PluginRSI, respectively, as discussed in §3.2.

3.2 The Optimization Loop

3.2.1 Initialization and overview.

The search starts from an initial harness $H = H_0 = \mathcal{H}(W_0, \mathcal{P}_0)$ and a plugin library $\mathcal{L}$, with $\mathcal{P}_0 \subseteq \mathcal{L}$. The harness H is iteratively evaluated on validation tasks $\mathcal{D}_{\text{val}}$, and the scores and execution trajectories are used for analysis to propose a new harness H'. Here $\mathcal{D}_{\text{val}}$ and $\mathcal{D}_{\text{test}}$ are disjoint validation and test datasets drawn from task distribution $\mathcal{D}$, where $\mathcal{D}_{\text{test}}$ is held-out and not involved in the optimization process.

Each iteration consists of two stages. The **plugin mutation** stage explores changes to individual plugins and adds beneficial implementations to $\mathcal{L}$. **Harness recomposition** then selects plugins from the updated library and revises the workflow to construct a new candidate. After T iterations, the algorithm returns the best harness on $\mathcal{D}_{\text{val}}$. Algorithm 1 summarizes this process.

3.2.2 Plugin mutation.

Plugin mutation uses execution feedback to guide changes to individual mechanisms in the current harness. At each iteration, we generate N mutation candidates in parallel from H, with each candidate using a separately sampled minibatch $\mathcal{M}$ of b tasks from $\mathcal{D}_{\text{val}}$. Based on the current harness's most recent evaluation results, each minibatch contains equal numbers of correctly and incorrectly solved tasks. The separate sampling of minibatches provides varied execution feedback to encourage diverse mechanism exploration.

For each minibatch, the proposer agent examines H and its execution feedback on that minibatch to select a plugin $p_i \in \mathcal{P}$ and produce a modified implementation p_i'. The modified plugin replaces p_i while the workflow and the remaining plugins stay fixed. The resulting plugin set is compiled with the unchanged workflow to obtain $H_i = \mathcal{H}(W, (\mathcal{P} \setminus \{p_i\}) \cup \{p_i'\})$. We evaluate H_i on its corresponding minibatch and compare with the score of H on the same minibatch. Plugin variants with empirical improvements are added to $\mathcal{L}$.

3.2.3 Harness recomposition.

Harness recomposition uses the implementations accumulated in $\mathcal{L}$ to explore new harness designs. This process allows both the plugin selection $\mathcal{P}$ and the workflow W to change.

At this stage, the proposer examines previous harness execution feedback and the mutation results. These comparisons guide the selection of a plugin set $\mathcal{P}' \subseteq \mathcal{L}$ that incorporates promising implementations from the updated library. The proposer also revises the workflow from $W \to W'$ to improve the plugin coordination, including when they are invoked and how information flows. The new harness candidate is then compiled as $H' = \mathcal{H}(W', \mathcal{P}')$ and evaluated on $\mathcal{D}_{\text{val}}$.

Algorithm 1 PluginRSI: Plugin-Oriented Harness Optimization

Require: Initial harness $H_0 = \mathcal{H}(W_0, \mathcal{P}_0)$, proposer agent Φ_{proposer}, validation tasks $\mathcal{D}_{\text{val}}$
Require: Minibatch size b, iterations T, mutation candidates per iteration N
Ensure: Optimized harness H^*

Initialize plugin library $\mathcal{L}$ and incumbent $(H, W, \mathcal{P}) \leftarrow (H_0, W_0, \mathcal{P}_0)$
Evaluate initial $(S[H], \mathcal{T}_H) \leftarrow \text{Evaluate}(H, \mathcal{D}_{\text{val}})$
for $t = 1, \ldots, T$ **do**
    `// Step 1: Plugin Mutation`
    for $i = 1, \ldots, N$ **in parallel do** ▷ parallel exploration on different subsets
        $\mathcal{M} \leftarrow$ balanced minibatch of b tasks from $\mathcal{D}_{\text{val}}$,
        Gather score $S_{\mathcal{M}}[H]$ and execution trace $\mathcal{T}_{H,\mathcal{M}}$ for H on $\mathcal{M}$
        Φ_{proposer} selects and mutates plugin $p_i \to p_i'$
        $H_i \leftarrow \mathcal{H}\left(W, (\mathcal{P} \setminus \{p_i\}) \cup \{p_i'\}\right)$ ▷ recompile with mutated plugin
        Gather score $S_{\mathcal{M}}[H_i]$ and execution trace $\mathcal{T}_{H_i,\mathcal{M}}$ for H_i on $\mathcal{M}$
        Add each p_i' with $S_{\mathcal{M}}[H_i] > S_{\mathcal{M}}[H]$ to $\mathcal{L}$ ▷ accumulate beneficial plugins
    end for
    `// Step 2: Harness Recomposition`
    Φ_{proposer} revises workflow $W \to W'$ and selects plugins $\mathcal{P}' \subseteq \mathcal{L}$
    $H' \leftarrow \mathcal{H}(W', \mathcal{P}')$ ▷ recompile with new workflow and plugins
    Evaluate H' on $\mathcal{D}_{\text{val}}$ to obtain $(S[H'], \mathcal{T}_{H'})$
    if $S[H'] > S[H]$ **then**
        $(H, W, \mathcal{P}) \leftarrow (H', W', \mathcal{P}')$
    end if
end for
return H

4 Experiments

4.1 Experimental Setup

Benchmarks and protocol. We evaluate on software engineering tasks from SWE-bench Verified (Chowdhury et al., 2024); command line interaction tasks from Terminal-Bench-2.1 (Merrill et al., 2026); and Question-Answering (QA) tasks from SuperGPQA (Du et al., 2026a), SciCite (Cohan et al., 2019), FinanceQA (Mateega et al., 2025), FrontierScience (Wang et al., 2026) and OlympiadBench (He et al., 2024). We split SWE-bench Verified into 400 held-in and 100 held-out tasks, and Terminal-Bench-2.1 into 59 held-in and 30 held-out tasks, as detailed in Appendix I. For QA, we sample 50 tasks from each of eight domains with random seed 50. We optimize on 150 tasks from Olympiad-Math, SciCite, and SuperGPQA-Economics, and evaluate on the other five domains. For each optimization method, we select the harness with the highest held-in score and fix it for subsequent evaluation. We report resolve rates (%) for SWE-bench and Terminal-Bench, and accuracies for QA.

Backbone models. We separately optimize SWE-Bench Verified harnesses with GPT-5.6 Terra (OpenAI, 2026b) and Kimi-K3 (Team et al., 2026) as proposers. We use Kimi-K3 as the solver for Terminal-Bench and QA. We then conduct evaluation on three different base models GPT-5.6 Terra, Kimi-K3 and GLM-5.2 (Zeng et al., 2026) to measure cross-model utility. By default, reasoning effort is set to `xhigh` for the proposer and `none` for the solver. We also evaluate each selected harness with the other solver backbones, keeping its workflow and plugins fixed. Full model configurations are provided in Appendix E.

Baselines. We compare with two groups of methods. **(1) Expert-curated harnesses** include ReAct (Yao et al., 2023), which follows a reasoning–acting loop in zero-shot and few-shot settings; and ACE (Zhang et al., 2026d), which accumulates experience in a context playbook. **(2) Automatically optimized harnesses** include GEPA (Agrawal et al., 2026b), which performs reflective evolution at the prompt level; and Darwin-Godel Machine (DGM) (Zhang et al., 2026b) and Meta-Harness (Lee et al., 2026), which search over complete harness implementations. All methods share the same task splits, API endpoints, and environment sandboxes.

Table 1: Comparison on SWE-bench Verified (Chowdhury et al., 2024). Harnesses are optimized and selected on 400 held-in tasks, then fixed for evaluation across solver backbones. Values are resolve rates (%). † marks the score used for harness selection. Bold marks the best result in each column within each optimization setting.

Method	Held-In (400)			Held-Out (100)		
	GPT-5.6 Terra	Kimi-K3	GLM-5.2	GPT-5.6 Terra	Kimi-K3	GLM-5.2
Expert-Curated Harnesses						
ReAct (Zero-Shot)	38.25	41.00	37.75	39.00	41.00	41.00
ReAct (Few-Shot)	52.50	44.25	38.25	55.00	47.00	41.00
ACE	46.75	41.00	30.25	43.00	37.00	31.00
Automatically Optimized Harnesses w/ GPT-5.6 Terra						
GEPA	50.50†	42.50	39.25	51.00	44.00	48.00
DGM	49.00†	42.00	37.25	50.00	51.00	43.00
Meta-Harness	54.50†	54.00	45.25	55.00	53.00	44.00
PluginRSI	**58.50**†	**60.50**	**48.00**	**65.00**	**60.00**	**52.00**
Automatically Optimized Harnesses w/ Kimi-K3						
GEPA	55.00	61.00†	47.00	61.00	62.00	52.00
DGM	57.75	58.00†	50.00	58.00	55.00	51.00
Meta-Harness	**58.25**	64.25†	**59.75**	60.00	63.00	59.00
PluginRSI	58.00	**66.75**†	59.50	**64.00**	**69.00**	**64.00**

Table 3: Results on QA benchmarks. We optimize harnesses on 150 tasks from three held-in domains. Both optimization and evaluation are on Kimi-K3. Values are accuracies rounded to 3 decimals. Bold indicates the best results.

Method	Held-In (150)				Held-Out (250)						Overall
	Math	SciCite	Eco	Avg.	Fin	FS	Physics	Law	Med	Avg.	
ReAct	0.480	0.540	0.420	0.480	0.280	0.620	0.140	0.320	0.300	0.332	0.388
GEPA	0.680	0.640	0.580	0.633	0.380	0.660	0.200	0.620	0.500	0.472	0.533
Meta-Harness	0.780	0.680	**0.800**	0.753	0.480	**0.740**	**0.280**	0.800	0.540	0.568	0.638
PluginRSI	**0.880**	**0.780**	**0.800**	**0.820**	**0.560**	0.720	**0.280**	**0.820**	**0.560**	**0.588**	**0.675**

4.2 Main Results

Results on SWE-bench Verified. Table 1 shows that PluginRSI improves performance with both optimization backbones. We observe higher held-in results for both GPT- and Kimi-optimized harnesses. With GPT-5.6 Terra and Kimi-K3, it achieves held-out resolve rates of 65.00% and 69.00%, exceeding the best baseline by 10.00 and 6.00 percentage points, respectively. These gains are larger than the corresponding held-in gains of 4.00 and 2.50 points. The discovered improvements therefore remain effective beyond the tasks used for harness selection.

Transfer across backbone models. The optimized harnesses retain their advantage when transferred to other solver models without further optimization. The harness optimized with GPT-5.6 Terra achieves held-out resolve rates of 60.00% with Kimi-K3 and 52.00% with GLM-5.2, exceeding the strongest baselines by 7.00 and 4.00 points. The harness optimized with Kimi-K3 achieves 64.00% with both GPT-5.6 Terra and GLM-5.2, improving over the strongest baselines by 3.00 and 5.00 points. For some held-in cross-model evaluations we also observe a performance degradation against baseline methods, *e.g.,* 58.00% against previous best 58.25%, where the performance gaps remain modest. We regard these as comparable results.

Table 2: Results on Terminal-Bench-2.1 with Kimi-K3. Values are resolve rates (%). Bold marks the best result in each column.

Method	Held-In (59)	Held-Out (30)
ReAct (Zero-Shot)	47.5	50.0
ACE	54.2	56.7
GEPA	57.6	63.3
DGM	57.6	60.0
Meta-Harness	64.4	66.7
PluginRSI	**69.5**	**73.3**

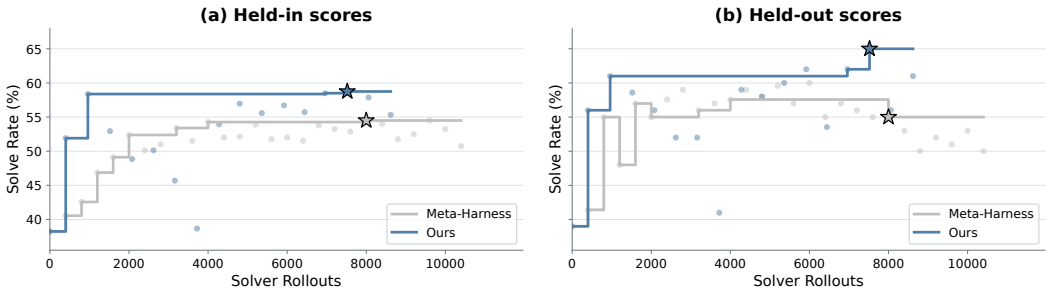


Figure 3: **Optimization dynamics.** Held-in and held-out resolve rates of the best held-in harness against cumulative solver rollouts. PluginRSI yields higher performing harness with fewer solver rollouts, and held-in performances aligns better with held-out scores.

Results on Terminal-Bench. Table 2 shows that the benefits extend to command-line interaction tasks. With Kimi-K3, PluginRSI achieves 69.5% on held-in tasks and 73.3% on held-out tasks, compared with 64.4% and 66.7% for Meta-Harness. The improvements are larger on held-out tasks.

Results on QA benchmarks. Table 3 shows the results. Math and Physics are from OlympiadBench (He et al., 2024); Eco(nomics), Law, and Med(icine) are from SuperGPQA (Du et al., 2026a). FS stands for FrontierScience (Wang et al., 2026) and Fin stands for FinanceQA (Mateega et al., 2025). PluginRSI improves held-in accuracy from 0.753 to 0.820 and overall accuracy from 0.638 to 0.675 over baselines. Held-out accuracy increases more modestly from 0.568 to 0.588. The smaller held-out gains are expected given the limited overlap among QA domains, as different domains require different specialized strategies. For example, strategies discovered for mathematical reasoning may not directly transfer to law questions.

Obs. 1: PluginRSI improves harness performance across task settings. The advantages are retained across solver models, and are larger on unseen similar tasks.

4.3 Optimization Dynamics

We examine how efficiently PluginRSI improves harness performance. Figure 3 compares PluginRSI with Meta-Harness using GPT-5.6 Terra. We plot the best held-in scores and the held-out scores of the corresponding harnesses against cumulative solver rollouts. The x-axis is the number of solver rollouts for fair comparison, since the plugin mutation stage introduces extra burden. Appendix L provides a token cost analysis.

PluginRSI rapidly reaches a held-in resolve rate that Meta-Harness does not attain within the evaluated budget. Harnesses with better held-in performance also improve on held-out tasks, with early gains followed by further improvements in later iterations. Meta-Harness improves more gradually on held-in tasks, while its held-out performance fluctuates and remains lower at the end of optimization.

Obs. 2: PluginRSI finds higher-performing harnesses with fewer solver rollouts, with held-in improvements accompanied by held-out gains.

4.4 Ablation Study

We examine the contributions of the plugin library and the two optimization stages. Table 4 reports results under Kimi-K3 optimization and transfer to GPT-5.6 Terra and GLM-5.2. For the *w/o initial plugins* variant, we retain only the four basic plugins from the 75-plugin library and keep the initial workflow unchanged. For the *w/o harness recomposition* variant, we adopt the mutation-stage harness with the highest score on its b-task minibatch.

Table 4: Ablation study on SWE-bench Verified. Harnesses are optimized and selected with Kimi-K3 on 400 held-in tasks, then fixed for evaluation across solver backbones. Values are resolve rates (%) and † marks the score used for harness selection. Bold marks the best result in each column.

	Held-In (400)			Held-Out (100)		
Method	GPT-5.6 Terra	Kimi-K3†	GLM-5.2	GPT-5.6 Terra	Kimi-K3	GLM-5.2
Meta-Harness	58.25	64.25	**59.75**	60.00	63.00	59.00
+ Plugin Library	58.25	65.50	55.25	58.00	64.00	56.00
+ Plugin Library w/o Initial Plugins	61.00	64.75	56.25	62.00	62.00	58.00
PluginRSI	58.00	66.75	59.50	**64.00**	**69.00**	**64.00**
w/o Initial Plugins	**62.25**	**67.00**	57.75	62.00	67.00	63.00
w/o Plugin Mutation	52.25	58.50	46.25	49.00	57.00	50.00
w/o Harness Recomposition	57.25	62.50	59.00	57.00	59.00	60.00

Plugin library and initial plugins. Adding an initialized or empty plugin library to Meta-Harness gives Kimi-K3 held-out resolve rates of 64.00% and 62.00%, close to its original 63.00%. This suggests that providing a plugin library alone yields limited gains. Meanwhile, removing the initial plugins from PluginRSI yields similar Kimi-K3 held-in scores (66.75% versus 67.00%), and reduces held-out scores by only 1.00–2.00 points across the three solvers. Strong performance is retained without these initial plugins.

Plugin mutation and harness recomposition. Removing plugin mutation or harness recomposition reduces the Kimi-K3 held-out resolve rate from 69.00% to 57.00% and 59.00%, respectively. Both ablations also reduce held-out scores with GPT-5.6 Terra and GLM-5.2. These results support combining local improvements to plugins with global changes to their coordination.

Obs. 3: Both plugin mutation and harness recomposition improve held-out performance. A plugin library alone does not bring obvious gains.

4.5 Plugin Accumulation and Reuse

We examine how plugins accumulate during optimization and whether the evolved library accelerates subsequent harness optimization.

Plugin accumulation. Figure 5(a) shows that the library grows from 75 to 132 plugins over 15 evolution steps, with additions across all four categories. Figure 5(b) tracks candidate size in `Python` lines of code (LoC). By the final step, PluginRSI candidates reach approximately 1,300 LoC, compared with 400 LoC for Meta-Harness. Most of this growth comes from plugin implementations, while the workflow remains below 300 LoC. These results show that accumulated plugins support larger harness implementations while keeping the workflow compact.

Reusing evolved plugins. Figure 4 compares optimization from the initial harness, the initial harness with the evolved library, and the evolved harness with its plugins. Reusing the library from the initial harness reaches a held-out resolve rate of 69% after two evolution steps, compared with 65% without reuse. This approaches the 70% reached after three steps of continued optimization from the evolved harness. The results show that the evolved library preserves useful improvements that accelerate optimization from a different starting harness.

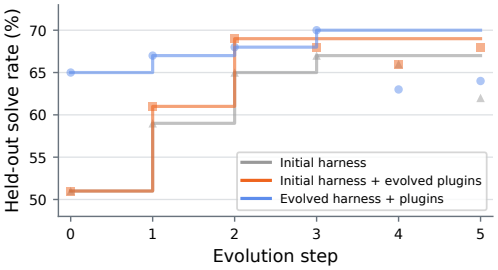
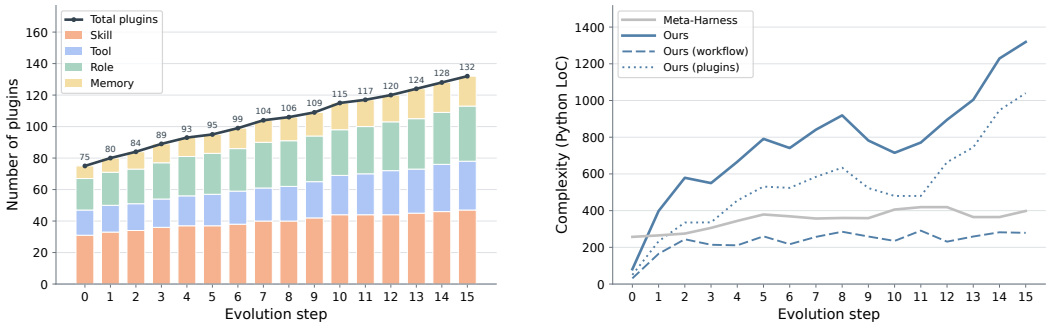


Figure 4: **Plugin reuse effectiveness.** On held-out tasks, optimizing from initial harness w/ the evolved plugin library accelerates convergence.

Obs. 4: Accumulated plugins support larger harness implementations with compact workflows. Reusing the evolved library accelerates subsequent optimization.



(a) Number of Plugins

(b) Harness implementation size

Figure 5: **Plugin accumulation statistics.** (a) Number of plugins in the plugin library. (b) Candidate implementation size during optimization. PluginRSI explores more complex harness structure by reusing discovered plugins, while the workflow implementation remains relatively brief.

5 Related Work

Self-Improving Agent. Existing work enables agents to improve from feedback through model parameter updates and the refinement of external artifacts. Policy learning approaches let agents themselves automatically generate tasks and interaction trajectories, and be optimized via reinforcement learning (Zhao et al., 2026; Huang et al., 2026; Qi et al., 2025; Zhai et al., 2025; Xia et al., 2026b) or supervised fine-tuning (Zweiger et al., 2026). For artifact optimization, prompt optimizers update instructions using execution feedback (Zhou et al., 2022; Yuksekgonul et al., 2024; Agrawal et al., 2026b), while context and memory methods organize information and experiences for subsequent tasks (Zhang et al., 2026d; Ouyang et al., 2026; Shi et al., 2026b; Shinn et al., 2023; Zhao et al., 2024; Yan et al., 2026; Shi et al., 2026c). Other approaches acquire and retain reusable behaviors as agent skills (Wang et al., 2024; Xia et al., 2026a; Ma et al., 2026). PluginRSI studies the improvement of executable mechanisms and their coordinating workflows via harness optimization.

Recursive Harness Optimization. Recursive harness optimization improves agent performance by searching over harness designs. Existing methods explore this space by iteratively rewriting harness implementations. One line of work targets specific aspects of the harness, including agentic workflows (Hu et al., 2025; Zhang et al., 2025), prompts and role assignments (Agrawal et al., 2026b;a), and executable components (Yang et al., 2026b; Yu et al., 2026; Fei et al., 2025; Shi et al., 2026a). Other studies directly optimize algorithms (Romera-Paredes et al., 2024; Novikov et al., 2025) and executable implementations of agents (Zhang et al., 2026b;c; Lee et al., 2026; Zhang et al., 2026a; Karten et al., 2026; Liu et al., 2026b). However, mechanisms remain entangled within the harness during iterative rewriting, making their individual contributions difficult to evaluate and preventing their reuse. PluginRSI uses a plugin system to decouple individual mechanisms, enabling their evaluation with the rest of the harness fixed and their reuse across candidates.

Modular Harness Design. Expert-curated agent systems adopt modular harness designs to organize human-designed mechanisms into composable components. For example, DeepSeek Harness exposes runtime capabilities as swappable plugins (DeepSeek-AI, 2026). This modular organization allows human-designed mechanisms to be incorporated into harness engineering (Shang et al., 2025; Du et al., 2026b; Han et al., 2025). Recent work (Chen et al., 2026; Luo et al., 2026) combines typed processors to automatically edit harness components and configurations. PluginRSI explores recursive harness optimization of plugin-parameterized harnesses, focusing on how plugins can be independently discovered, accumulated and reused across iterations.

6 Conclusion

We introduce PluginRSI, a plugin-oriented algorithm for recursive harness optimization. It improves individual mechanisms through plugin mutation and recomposes harnesses from an accumulated plugin library. Experiments show performance gains across software engineering, command-line

interaction, and QA tasks. On SWE-bench Verified, PluginRSI finds stronger harnesses with fewer solver rollouts and retains its held-out advantage across solver models. Ablations support the contributions of both optimization stages. Reusing the evolved library accelerates subsequent optimization from the initial harness. These results show that harness evolution can accumulate reusable mechanisms that support further improvement.

REFERENCES

Lakshya A Agrawal, Donghyun Lee, Shangyin Tan, Wenjie Ma, Karim Elmaaroufi, Sanjit A. Seshia, Koushik Sen, Dan Klein, Ion Stoica, Joseph E. Gonzalez, Omar Khattab, Alexandros G. Dimakis, and Matei Zaharia. optimize_anything: A universal api for optimizing any text parameter. In *Proceedings of the ACM Conference on AI and Agentic Systems*, pp. 1300–1304, 2026a.

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026b.

Anthropic. Claude code. https://github.com/anthropics/claude-code, 2025, 2025. URL `https://github.com/anthropics/claude-code`.

Tingyang Chen, Shuo Lu, Kang Zhao, Weicheng Meng, Hanlin Teng, Tianhao Li, Chao Li, Xule Liu, Jian Liang, Zhizhong Zhang, Yuan Xie, Heng Qu, Kun Shao, and Jian Luan. HarnessX: A composable, adaptive, and evolvable agent harness foundry. *arXiv preprint arXiv:2606.14249*, 2026.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench verified, 2024. URL `https://openai.com/index/introducing-swe-bench-verified/`. Accessed: 2026-09-22.

Arman Cohan, Waleed Ammar, Madeleine van Zuylen, and Field Cady. Structural scaffolds for citation intent classification in scientific publications. In Jill Burstein, Christy Doran, and Thamar Solorio (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 3586–3596, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1361.

DeepSeek-AI. Deepseek harness: Everything is a plugin. `https://deepseek.com/harness/en/`, 2026. Accessed: 2026-09-22.

Xeron Du, Yifan Yao, Kaijing Ma, Bingli Wang, Tianyu Zheng, Minghao Liu, Yiming Liang, Xiaolong Jin, Zhenlin Wei, Chujie Zheng, et al. Supergpqa: Scaling llm evaluation across 285 graduate disciplines. *Advances in Neural Information Processing Systems*, 38, 2026a.

Yiming Du, Yuxin Jiang, Tao Yuan, Jianbo Dai, Shaowei Wang, Jierun Chen, Chaofan Tao, Xianzhi Yu, Lifeng Shang, Kam-Fai Wong, et al. Lego-rl: Harness-native reinforcement learning for coding agents. *arXiv preprint arXiv:2608.17393*, 2026b.

Yi Duan, Ying Liu, Zirui Tang, Haodong Chen, Jun Zhou, Yumou Liu, Bangrui Xu, Yukai Wu, Sidi Chen, Yuhan Zhou, et al. The last ai built by humans: Toward genuine recursive self-improvement. *arXiv preprint arXiv:2609.11873*, 2026.

Xiang Fei, Xiawu Zheng, and Hao Feng. Mcp-zero: Active tool discovery for autonomous llm agents. *arXiv preprint arXiv:2506.01056*, 2025.

Dongge Han, Camille Couturier, Daniel Madrigal Diaz, Xuchao Zhang, Victor Rühle, and Saravan Rajmohan. Legomem: Modular procedural memory for multi-agent llm systems for workflow automation. *arXiv preprint arXiv:2510.04851*, 2025.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, et al. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3828–3850, 2024.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, 2025.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning llm from zero data. In *International Conference on Learning Representations*, volume 2026, pp. 130770–130790, 2026.

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

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. *arXiv preprint arXiv:2605.09998*, 2026.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses. In *Conference on Language Modeling*, 2026.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026a.

Minhua Lin, Hanqing Lu, Zhan Shi, Bing He, Rui Mao, Zhiwei Zhang, Zongyu Wu, Xianfeng Tang, Hui Liu, Zhenwei Dai, et al. Position: Agentic evolution is the path to evolving llms. *arXiv preprint arXiv:2602.00359*, 2026b.

Shuaiqi Liu, Zhengkai Lin, Yuxiang Zhang, Yuanyi Ren, Yue Wu, Yongbin Li, Zheng Wang, Zhihang Fu, and Jieping Ye. The path to recursive self-improving agents: Foundation, framework, and future directions. *Preprints*, August 2026a. doi: 10.20944/preprints202608.0051.v1.

Zewen Liu, Zhan Shi, Yisi Sang, Bing He, Minhua Lin, Tianxin Wei, Dakuo Wang, Benoit Dumoulin, Wei Jin, and Hanqing Lu. Adaptive auto-harness: Sustained self-improvement for agentic system deployment on open-ended task streams. *arXiv preprint arXiv:2606.01770*, 2026b.

Xiaotian Luo, Fengxingyu Wang, Chuanrui Hu, Dizhan Xue, and Yafeng Deng. Self-evolving agent harnesses via gated semantic quality-diversity. *arXiv preprint arXiv:2607.13683*, 2026.

Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. Skillclaw: Let skills evolve collectively with agentic evolver. *arXiv preprint arXiv:2604.08377*, 2026.

Spencer Mateega, Carlos Georgescu, and Danny Tang. Financeqa: A benchmark for evaluating financial analysis capabilities of large language models. *arXiv preprint arXiv:2501.18062*, 2025.

Q. Meng, Y. Wang, L. Chen, Y. Li, W. Wu, W. Jiang, Q. Wang, C. Lu, Y. Gao, Y. Wu, and Y. Hu. Agent harness for large language model agents: A survey. *Preprints*, 2026. doi: 10.20944/preprints202604.0428.v3.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, 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 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, 2026.

Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, et al. Code as agent harness. *arXiv preprint arXiv:2605.18747*, 2026.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. Codex. Note: Product page, 2026a. URL `https://openai.com/codex/`.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition, 2026b. URL `https://openai.com/index/gpt-5-6/`.

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

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

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Jiadai Sun, Xinyue Yang, Yu Yang, Shuntian Yao, Wei Xu, et al. Webrl: Training llm web agents via self-evolving online curriculum reinforcement learning. In *International Conference on Learning Representations*, volume 2025, pp. 79791–79821, 2025.

Zhe Ren, Yimeng Chen, Dandan Guo, Guowei Rong, Tonghui Li, RB Xiong, Qingfeng Lan, Wenyi Wang, Li Nanbo, Yibo Yang, et al. Self-improvements in modern agentic systems: A survey. *arXiv preprint arXiv:2607.13104*, 2026.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. AgentSquare: Automatic LLM agent search in modular design space. In *The Thirteenth International Conference on Learning Representations*, 2025.

Yaorui Shi, Yuxin Chen, Zhengxi Lu, Yuchun Miao, Shugui Liu, Qi Gu, Xunliang Cai, Xiang Wang, and An Zhang. Skill1: Unified evolution of skill-augmented agents via reinforcement learning. *arXiv preprint arXiv:2605.06130*, 2026a.

Yaorui Shi, Yuxin Chen, Siyuan Wang, Sihang Li, Hengxing Cai, Qi Gu, Xiang Wang, and An Zhang. Look back to reason forward: Revisitable memory for long-context llm agents. In *International Conference on Learning Representations*, volume 2026, pp. 78055–78082, 2026b.

Yaorui Shi, Shugui Liu, Yu Yang, Wenyu Mao, Yuxin Chen, Qi GU, Hui Su, Xunliang Cai, Xiang Wang, and An Zhang. MemOCR: Layout-aware visual memory for efficient long-horizon reasoning. In *Forty-third International Conference on Machine Learning*, 2026c.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y Charles, et al. Kimi k3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856.

Miles Wang, Robi Lin, Kat Hu, Joy Jiao, Neil Chowdhury, Ethan Chang, and Tejal Patwardhan. Frontierscience: Evaluating ai's ability to perform expert-level scientific tasks. *arXiv preprint arXiv:2601.21165*, 2026.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. SkillRL: Evolving agents via recursive skill-augmented reinforcement learning. In *ICLR 2026 Workshop on Lifelong Agents: Learning, Aligning, Evolving*, 2026a.

Peng Xia, Kaide Zeng, Jiaqi Liu, Can Qin, Fang Wu, Yiyang Zhou, Caiming Xiong, and Huaxiu Yao. Agent0: Unleashing self-evolving agents from zero data via tool-integrated reasoning. In *Third Conference on Language Modeling*, 2026b.

Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Jinhe Bi, Kristian Kersting, Jeff Z. Pan, Hinrich Schuetze, Volker Tresp, and Yunpu Ma. Memory-r1: Enhancing large language model agents to manage and utilize memories via reinforcement learning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12805–12825, July 2026. ISBN 979-8-89176-390-6.

J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press. Swe-agent: agent-computer interfaces enable automated software engineering. In *The Thirty-Eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=mXpq6ut8J3&referrer=%5Bthe%20profile%20of%20Shunyu%20Yao%5D(%2Fprofile%3Fid%3D~Shunyu_Yao1).

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

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, et al. Skillopt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026b.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

Zhuoyun Yu, Xin Xie, Wuguannan Yao, Chenxi Wang, Lei Liang, Xiang Qi, and Shumin Deng. Skilladaptor: Self-adapting skills for llm agents from trajectories. *arXiv preprint arXiv:2606.01311*, 2026.

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

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Yunpeng Zhai, Shuchang Tao, Cheng Chen, Anni Zou, Ziqian Chen, Qingxu Fu, Shinji Mai, Li Yu, Jiaji Deng, Zouying Cao, et al. Agentevolver: Towards efficient self-evolving agent system. *arXiv preprint arXiv:2511.10395*, 2025.

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

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

Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents. *arXiv preprint arXiv:2603.19461*, 2026c.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *The Fourteenth International Conference on Learning Representations*, 2026d.

Shaokun Zhang, Jieyu Zhang, Jiale Liu, Linxin Song, Chi Wang, Ranjay Krishna, and Qingyun Wu. Offline training of language model agents with functions as learnable weights. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 60315–60335. PMLR, 21–27 Jul 2024.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Thirty-Eighth AAAI Conference on Artificial Intelligence*, pp. 19632–19642. AAAI Press, 2024. doi: 10.1609/aaai.v38i17.29936.

Andrew Zhao, Yiran Wu, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. *Advances in Neural Information Processing Systems*, 38:105816–105879, 2026.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*, 2022.

Adam Zweiger, Jyo Pari, Han Guo, Yoon Kim, and Pulkit Agrawal. Self-adapting language models. *Advances in Neural Information Processing Systems*, 38:74084–74115, 2026.

A Reproducibility statement

We describe the optimization procedure in Section 3 and Algorithm 1, and the evaluation protocol in Section 4.1. Appendix E reports model configurations, search hyperparameters and the API access period. Appendices H and G provide the initial harness, seed plugin sources, and proposer prompts. Appendix I describes the SWE-Bench Verified split, and Appendix J specifies the plugin interfaces. An anonymous code repository is linked in the abstract.

B Ethics statement

The experiments and research within the scope of this paper raise few ethical concerns, such as potentially harmful insights and discrimination/bias/fairness concerns. Our experiments use existing benchmarks for software engineering, command-line interaction, and question answering. Automatically evolved harnesses can improve software development, but generated code and commands may also introduce errors or unintended changes. Deployment beyond benchmark environments should therefore include code review and appropriate execution permissions.

C Use of AI Statement

We used generative AI tools to assist with literature review, code development and manuscript polishing. In particular, GPT-5.6 Sol assisted with mechanism extraction for the seed plugin library (as stated in Appendix G), and all resulting plugins underwent manual inspection. The use of LLMs as proposers and solvers is described in the method and implementation details (as stated in Appendix E).

D Limitations and Future Work

Despite its effectiveness, this work faces several limitations. First, we evaluate harness evolution on fixed task sets over a limited number of iterations. Longer-term evolution under changing task distributions remains to be explored, together with the management of a growing plugin library. Second, the transfer of discovered mechanisms depends on the task domain and solver model. Gains are smaller across dissimilar QA domains; thus, optimizing across multiple domains and models could improve broader reuse. Third, plugin mutation evaluates each change with the remaining harness fixed. This can miss mechanisms that require coordinated changes to other plugins or the workflow. Future work could explore harness optimization on long-horizon tasks, more significant cross-domain transferability and the plugin-workflow joint optimization.

E Implementation Details

For the SWE-Bench experiments in Table 1, we use GPT-5.6 Terra and Kimi-K3 as proposers in the respective optimization settings, with the same backbone serving as the optimization solver. Harnesses are selected using held-in scores from the corresponding optimization solver and then fixed for evaluation across GPT-5.6 Terra, Kimi-K3, and GLM-5.2. For Terminal-Bench, QA and all ablation experiments, Kimi-K3 serves as both proposer and optimization solver. Reasoning effort is set to `xhigh` for the proposer and `none` for all solvers. At each iteration, PluginRSI generates $N = 8$ plugin mutation candidates per iteration and evaluates each on a separately sampled minibatch of $b = 20$ tasks.

Each minibatch contains 10 tasks solved by the current harness and 10 unsolved tasks. For example, on SWE-Bench Verified, each iteration evaluates mutation candidates and one recomposition candidate using $|\mathcal{D}_{\text{val}}| + N \times b = 400 + 8 \times 20 = 560$ solver rollouts. We run our method for $T = 15$ iterations and the baselines for $T = 25$ iterations. The subsequent harness recomposition stage generates one candidate, which is evaluated on all held-in tasks. Tables 5 and 6 summarize the search hyperparameters and model configurations. For all models, we use response APIs accessed in September 2026. It is worth noting that we observe an obvious performance degradation in GPT-series models around the end of August. Although the cause of this degradation remains unclear, we reran all experiments in September to ensure a fair comparison.

Baseline implementations. ReAct follows a reasoning–acting loop with either zero-shot instructions or few-shot demonstrations. ACE augments the harness with a context playbook that accumulates experience. In our experiment we only turn on the "online" memory of ACE with no cross-task experience. Thus, it is considered an "expert-curated" baseline instead of an automatically updated one.

All the automatically optimized harnesses start from a ReAct-style initial harness. GEPA revises prompt instructions using execution feedback, where we let it recursively optimize the solver's system prompt. DGM and Meta-Harness revise complete executable harness implementations from the initial harness. All methods use the same task splits, API endpoints, and environment sandboxes, with model backbones following the corresponding experimental setting. All automatically optimized baselines run for 25 iterations. The selected harness is then kept fixed for held-out and cross-model evaluation.

Plugin reuse setup. The evolved harness and plugin library used in Figure 4 come from the search that produced the best held-in-selected PluginRSI result in the corresponding optimization setting of Table 1. The three conditions start from the initial harness with the initial library, the same initial harness with the evolved library, and the selected evolved harness with its evolved library, respectively. The subsequent harness optimization runs on the held-out 100 tasks following the held-in protocol in Section 4.1.

Table 5: Hyperparameters.

Hyperparameter	Value
Proposer Agent	
Optimization iterations (T)	15
Mutation candidates per iteration (N)	8
Mutation minibatch size (b)	20
Recomposition candidates per iteration	1
Solved:Unsolved tasks per minibatch	10:10
Max interaction turns	40
Random Seed	42
Solver Agent	
Max interaction turns	20
Timeout seconds	360

Table 6: Model configurations for the proposer and solvers.

Role	Model	Reasoning	Max Output Tokens
Proposer	GPT-5.6 Terra	`xhigh`	16,384
Proposer	Kimi-K3	`xhigh`	16,384
Solver	GPT-5.6 Terra	`none`	8,192
Solver	Kimi-K3	`none`	8,192
Solver	GLM-5.2	`disabled`	8,192

F Additional Results

We further compare against SkillOPT (Yang et al., 2026b), A-Evolve (Lin et al., 2026b) and Agentic Harness Engineering (AHE) (Lin et al., 2026a) on SWE-bench Verified with Kimi-K3 as both proposer and solver.

The results in Table 7 show that the advantage of PluginRSI also holds over the additional baselines. Compared with SkillOPT, A-Evolve, and AHE, PluginRSI improves held-out resolve rates by

Table 7: Results on SWE-bench Verified with Kimi-K3 as both proposer and solver. Values are resolve rates (%). Bold marks the best result in each column.

Method	Held-In (400)	Held-Out (100)
GEPA	61.00	62.00
DGM	58.00	55.00
SkillOPT (Yang et al., 2026b)	54.75	60.00
A-Evolve (Lin et al., 2026b)	61.75	57.00
AHE (Lin et al., 2026a)	60.25	63.00
Meta-Harness	64.25	63.00
PluginRSI	**66.75**	**69.00**

Table 8: Sources used to construct the seed plugin library. Repository and specification names link to their corresponding resources.

Source category	**Papers and resources**
Research papers	SWE-agent (Yang et al., 2024), Agentless (Xia et al., 2024), SWE-smith (Yang et al., 2026a), and SWE-Bench (Jimenez et al., 2024).
Agent implementations	mini-SWE-agent, Aider, DeepSeek Harness, and OpenHands Software Agent SDK.
Skill instructions and organization	Agent Skills specification, Anthropic Skills, Superpowers, Addy Osmani Agent Skills, and OpenHands Extensions.
Trajectories and evaluation records	SWE-Bench experiment repository and SWE-smith trajectory corpus.

9.00, 12.00, and 6.00 points, respectively. These results further support the effectiveness of plugin-oriented harness optimization beyond the tasks used during search. For computational burdens, we do not put these baseline methods in § 4.2 for comparison.

G Initial Harness

Seed Plugin Library. We construct the seed plugin library $\mathcal{L}_0$ from prior research and open-source agent systems, with sources summarized in Table 8. We manually curate the paper list and use GPT-5.6 Sol to extract role responsibilities and prompts, reusable skill procedures, tool schemas, and memory record structures. We adapt these assets into `Python` plugins with standardized interfaces and manually inspect all plugins. Each plugin retains its configuration, supporting resources, and provenance documenting its sources and adaptation. The resulting library contains 75 components across four categories (Table 9), from which each harness selects the components it uses.

Roles render prompts specifying responsibilities and constraints, with input/output metadata where applicable. Tools expose structured argument schemas and executable operations: repository actions run in the task environment, while patch submission writes a prediction artifact. Skills provide loadable instructions and supporting resources. Memory components manage the mechanisms to store and retrieve task-local evidence using lexical overlap.

Seed Workflow. The initial harness instantiates four plugins from $\mathcal{L}_0$. The harness starts with a coder role, a terminal tool, a debugging skill, and an experience-bank memory. The workflow loads the skill instructions into the role prompt and exposes the selected tools to the solver. At each turn, it retrieves task-local observations, queries the solver, executes any requested tools, and stores their outputs in memory. Execution ends when the solver returns without tool calls or reaches the limit of 20 turns. Memory is initialized separately for each task. The complete workflow and its plugin configuration are shown in Figure 6.

Table 9: Composition of the seed plugin library. Coverage is summarized by function.

Category	Count	Functional coverage
Roles	20	Baseline coding and boundary enforcement; task analysis and context preparation; fault localization and diagnosis; patch planning, generation, and ranking; verification and finalization; experience extraction and plugin or workflow editing.
Tools	16	Terminal and shell execution; file access and repository navigation; text and symbol search; patch application; Git inspection; test discovery and execution; linting and failure inspection; patch submission.
Skills	31	Task analysis and repository orientation; skill routing and instruction loading; code localization and call-chain tracing; reproduction and debugging; patching and contract preservation; testing and completion checks; blocker diagnosis and recovery; lesson extraction and skill improvement; harness-authoring guidance.
Memory	8	A baseline experience bank and seven structured record types: trajectories, task outcomes, failure signatures, lesson candidates, counterexamples, component credit, and skill evaluations.
Total	**75**	

```python
class Workflow:
    async def run(self, task, runtime, plugins):
        tools = {alias: plugin for alias, plugin in plugins.items() if isinstance(plugin, Tool)}
        memories = [plugin for plugin in plugins.values() if isinstance(plugin, Memory)]
        guidance = "\n".join(plugin.load({}).instructions for plugin in plugins.values() if isinstance(plugin, Skill))
        system = "\n".join(plugin.render({"guidance": guidance}) for plugin in plugins.values() if isinstance(plugin, Role))
        schemas = [dict(plugin.schema(), name=alias) for alias, plugin in tools.items()]
        messages = [{"role": "system", "content": system}, {"role": "user", "content": task.instruction}]
        for turn in range(MAX_TURNS):
            if turn:
                recalled = [item.content for memory in memories for item in await memory.retrieve({"query": task.instruction})]
                if recalled:
                    messages.append({"role": "user", "content": "Relevant task-local observations:\n" + "\n".join(recalled)})
            response = await runtime.model.complete(messages, schemas)
            messages.extend(response["output"])
            calls = [item for item in response["output"] if item["type"] == "function_call"]
            if not calls:
                return SolveResult(response["output_text"])
            for call in calls:
                result = await tools[call["name"]].execute(json.loads(call["arguments"]))
                messages.append({"type": "function_call_output", "call_id": call["call_id"], "output": result.content})
            for memory in memories:
                await memory.update({"content": result.content, "metadata": {"turn": turn, "tool": call["name"]}})
        return SolveResult("Turn limit reached.", termination_reason="turn_limit")
```

Figure 6: `Python` implementation of the initial harness workflow.

H System Prompt for Proposer

The proposer uses separate instructions for plugin mutation and harness recomposition. Each proposal starts a new conversation, with the current harness, accessible plugin references, evaluation feedback, and historical candidate summaries supplied as context. Trajectory summaries provide an overview of the feedback, while file-reading actions allow the proposer to inspect full implementations and relevant trajectory segments. The phase-specific instructions are given below.

Plugin Mutation. Each mutation branch receives the current harness and its scores and execution trajectories on a separately sampled minibatch. The proposer uses this feedback to select a plugin used by the harness and develop a modified implementation, while keeping the workflow, the remaining plugins, and the selected instance's configuration fixed. The mutation instructions are summarized in `Plugin Mutation Instructions`.

Harness Recomposition. The proposer receives the incumbent's full held-in feedback and the mutation results from the current iteration. It selects plugins from the updated library and revises the

workflow to coordinate them. Plugin implementations remain fixed during this stage. The detailed raw prompt is included in `System Prompt for Harness Recomposition`.

Both prompts are followed by a shared file-action protocol, which is included in `File-Action Protocol`. The available actions are `list_files`, `read_file`, `read_file_range`, `write_file`, and `submit_proposal`.

Plugin Mutation Instructions

Use this local experiment to invent a plugin-level solution to a concrete failure in the supplied trajectories. Read summaries, then inspect relevant full trace ranges and plugin code to identify where the attempt went wrong; check successes for regressions.

Inspect the current harness and the supplied minibatch feedback. Develop one transferable mechanism that could fix that failure. You may revise the existing implementation or replace it from scratch; the copied code is only a scaffold. You may revise the selected plugin's implementation or replace it from scratch.

Do not implement explicit content hashing or digests. Use direct string equality for deduplication.

Write the modified implementation to a new `new_plugins/<kind>/<name>/<version>/` package for the selected plugin. `library/` is READ-ONLY. Never write plugins under `child/plugins/` or any other `child/` subdirectory.

Keep every workflow file byte-for-byte unchanged. In `harness.yaml`, change only the selected instance's plugin reference to the new version; keep its alias, configuration, and all other manifest values unchanged.

Create exactly one new plugin with the same kind/name as the selected source reference and a new version. Implement a transferable improvement, not task-specific answers.

Other plugins and the shared runtime/evaluator are immutable. Keep the public interface and existing instance configuration compatible. Do not install dependencies or inspect hidden tests.

Use local feedback and your previous attempts, including failures, to refine the mechanism. In hypothesis, cite the task/trace evidence, the failure mechanism and the expected repair.

Deliver actual files and call `submit_proposal` with `hypothesis`, `changes:list[string]`, `new_plugin_refs`. New plugin provenance requires `label`, `sources`, `upstream_assets`, `adaptation` and `parent_refs`.

System Prompt for Harness Recomposition

Build a better complete Harness by integrating useful ideas from this round's plugin experiments. Prioritize newly published plugins with positive local score gains; read their implementations and relevant parent/child trajectories to understand which failures they repaired, any regressions, and when their mechanisms should help.

Local gains are evidence for integration, not a guarantee of full-set improvement.

Redesign workflow control flow, context and plugin composition so selected plugins are actually used where they can help. Combine compatible mechanisms; avoid blindly stacking plugins or preserving the incumbent assembly by default. Use rejected attempts to avoid repeating failures.

In hypothesis, name the selected plugin refs, supporting local results and how the workflow will use them; explain if no new plugin is suitable.

Do not implement explicit content hashing or digests. Use direct string equality for deduplication.

You may rewrite the entire child workflow package, helpers, resources and plugin assembly. Use only frozen `allowed_plugin_refs`. Do not create or modify plugin packages in this phase.

Do not change the model, runtime, evaluator, resource ceilings or hidden tests; do not install dependencies. Do not hard-code task answers.

Deliver executable files and `submit_proposal` with `new_plugin_refs=[]`.

File-Action Protocol (shared for both stages).

Use the same confined file actions described below, but encode them in ordinary text.

Every response must be one JSON object:

```
{
  "actions": [
    {
      "name": "read_file",
      "arguments": {
        "path": "child/workflow.py"
      }
    }
  ],
  "final": false
}
```

You may batch actions. Their results arrive as quoted data in the next user message.

Use `final=true` only after delivering all executable files and `proposal_result.json`. Use `submit_proposal` for final metadata. Each changes entry must be a string, not an object.

Never emit API function calls, shell commands as actions, prose outside JSON, or a mere diff description.

File actions have exactly the same read/write scope as the supplied coding-agent contract.

I Dataset Split

SWE-Bench Verified. We use a fixed partition of the 500 tasks in SWE-Bench Verified into a held-in split $\mathcal{D}_{\text{val}}$ of 400 tasks and a held-out split $\mathcal{D}_{\text{test}}$ of 100 tasks. The partition is specified by disjoint task-ID lists and is shared across all methods and solver backbones. During optimization, mutation minibatches are sampled exclusively from $\mathcal{D}_{\text{val}}$, and recomposition candidates are evaluated on the full held-in split. We select the harness with the highest held-in resolve rate under the corresponding optimization solver and keep its workflow and plugin configuration fixed for held-out and cross-model evaluation.

Subdomain composition. We treat each source repository as a subdomain and compare its proportion within each split in Figure 7. Normalizing by split size allows a direct comparison despite the 4:1 size ratio. Django accounts for 45.25% of held-in tasks and 50% of held-out tasks; SymPy accounts for 14.25% and 18%, respectively, while Sphinx accounts for 9.75% and 5%. The held-in split covers all 12 repositories, whereas the held-out split covers 10: the two Seaborn tasks and the single Flask task occur only in the held-in split. Thus, the held-out evaluation measures task-level generalization within the benchmark, rather than transfer to unseen repositories, and does not provide held-out evidence for Seaborn or Flask.

J Plugin Schema

Plugin packages. Each plugin is implemented as a versioned Python package stored at `<library>/<kind>/<name>/<version>/`. The package contains a `plugin.yaml` manifest, `__init__.py`, an implementation module, and supporting resources such as prompts and rubrics. The manifest specifies how the plugin is loaded and configured, while the Python implementation defines its behavior. Figure 8 shows an example plugin manifest and its use in a harness.

Plugin manifest. All four plugin categories share the same manifest structure. The fields `kind`, `name`, and `version` identify a plugin through a reference such as `tool/terminal/v0001`, and `entrypoint` specifies its implementation class. Instance configurations are defined by `config_schema` using JSON Schema. The loader fills in default values and validates the configuration before creating the instance. For example, the terminal plugin in Figure 8(a) specifies an output limit and an execution timeout. Optional `provenance` metadata records the plugin's sources, adaptation details, and parent references.

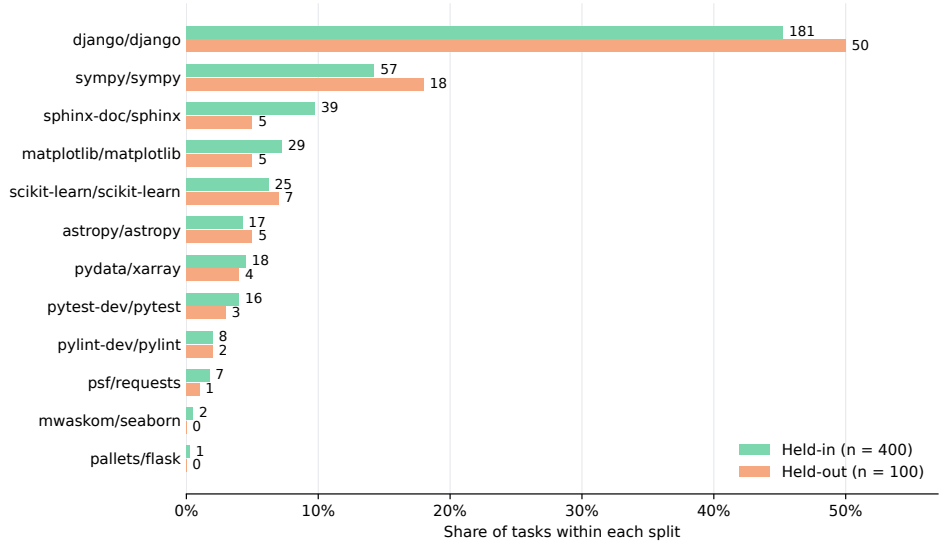


Figure 7: Sub-domain distribution of held-in and held-out tasks on Swe-Bench-Verified.

(a) Plugin manifest: `plugin.yaml`

```yaml
schema_version: 1
kind: tool
name: terminal
version: v0001
entrypoint: implementation:Terminal
description: Execute a shell command.
config_schema:
   type: object
   properties:
      max_output_bytes:
         type: integer
         minimum: 1
         default: 30000
   timeout_seconds:
      type: number
      exclusiveMinimum: 0
      default: 180
   additionalProperties: false
provenance:
sources: []
adaptation: Environment-command wrapper.
parent_refs: []
```

(b) Harness manifest: `harness.yaml`

```yaml
schema_version: 1
entrypoint: workflow:Workflow
plugins:
   coder:
      ref: role/coder/v0001
      config: {}
   terminal:
      ref: tool/terminal/v0001
      config:
            timeout_seconds: 180
   debugging:
      ref: skill/debugging/v0001
      config: {}
   experience:
      ref: memory/experience_bank/v0001
      config:
            top_k: 5
```

Figure 8: Examples of plugin and harness manifests. (a) `plugin.yaml` defines a terminal tool and its instance configuration. (b) `harness.yaml` assembles a harness by mapping workflow aliases to versioned plugins and their configurations.

Plugin interfaces. Each plugin implements the Python interface of its category, as summarized in Table 10. The workflow invokes these methods to render role prompts, load skill instructions, execute tools, and manage memory. Tool-call argument schemas are exposed through `schema()`, separately from the instance configuration defined in `plugin.yaml`.

Harness assembly. The `harness.yaml` manifest selects plugin versions and assigns each instance an alias and configuration. The loader checks the selected packages and their interfaces, then instantiates them with the specified configurations. The workflow accesses these instances through

Table 10: Plugin interfaces exposed to the workflow.

Category	Interface	Function
Role	`render(context)`	Returns a prompt string constructed from the supplied context.
Tool	`schema()` `execute(arguments)`	Exposes the tool-call schema and asynchronously executes an operation, returning a `ToolResult`.
Skill	`load(context)`	Returns a `SkillContent` containing instructions and associated resources.
Memory	`update(experience)` `retrieve(query)`	Asynchronously stores an experience and retrieves a list of `MemoryItem` objects.

Table 11: Selected plugin mutations and their screening results. Values are resolved tasks before and after mutation on each candidate's minibatch, using cached outcomes for the original harness. For incremental evidence retrieval, one verifier timeout leaves 19 tasks with valid outcomes in both evaluations.

Category	Discovered mechanism	Before	After
Role	Repair-scope control	10/20	13/20
	Discriminative checks and lifecycle continuity	10/20	13/20
Skill	Semantic source preservation	10/20	13/20
	Explicit precedence and representation checks	10/20	13/20
	Public-path checks with reachability controls	10/20	14/20
Memory	Incremental evidence retrieval	10/19	14/19
	Test-outcome classification	10/20	12/20
Tool	Failure-aware terminal feedback	10/20	14/20

the `plugins` mapping; for example, `plugins["terminal"]` refers to the terminal instance in Figure 8(b). Its asynchronous `run(task, runtime, plugins)` method coordinates model calls and plugin execution and returns a `SolveResult`. Each versioned plugin reference identifies a fixed implementation. Plugin mutation creates a new version and updates the corresponding reference while preserving the instance alias, configuration, and category interface. Harness recomposition selects existing plugin versions and revises the workflow that coordinates them.

K Discovered Plugins

We examine how plugin mutation changes the mechanisms that guide solver behavior. Table 11 summarizes eight selected mutations across roles, skills, memory, and tools, with one example from each category illustrated below.

Role: repair-scope control. The mutated coder role refines how the solver determines the scope of a repair. While the original role encourages small edits, the mutated instructions place guards, transactions, or cleanup around the smallest sequence of persistent mutations that requires them (Figure 9). In `django__django-16100`, the original solver wraps the entire admin view in a transaction, whereas the solver using the mutated role limits the transaction to the affected bulk-write loop and resolves the task. This case illustrates how a general preference for small edits becomes an explicit rule for locating the repair boundary.

Skill: semantic source preservation. The debugging skill adds explicit checks on where a repaired operation should obtain its values. The mutated instructions require the solver to identify the intended source and define the behavior when it is absent (Figure 10). In `pydata__xarray-6461`, the original solver falls back to the condition's attributes when the source is scalar. With the mutated skill, it returns empty attributes and resolves the task. The change preserves the intended source semantics rather than introducing a fallback solely to avoid an exception.

(a) Before mutation

Role guidance: `system_prompt.md`

Work in short evidence loops: orient and locate the responsible code, obtain a minimal reproduction or identify the closest existing test, then make the smallest plausible edit and add or update a focused regression test.

(b) After mutation

Role guidance: `system_prompt.md`

For a partial batch write, validation failure, transaction, cache update, or state reset, put the guard, rollback scope, or cleanup at the **smallest sequence of persistent mutations that needs it**.

Figure 9: Repair-scope control in the coder role. The mutated instructions specify where to place guards, transactions, or cleanup when repairing persistent mutations.

(a) Before mutation

Skill guidance: `SKILL.md`

Make the smallest implementation change that enforces the stated invariant; preserve unrelated APIs, ordering, types, and independently meaningful occurrences.

(b) After mutation

Skill guidance: `SKILL.md`

Do not replace an invalid positional access with a fallback to a different input merely because it avoids the exception; decide whether the correct absent-source result is **empty, a documented default, or an explicitly defined alternative**.

Figure 10: Semantic source preservation in the debugging skill. The mutated instructions require an explicit decision about the correct behavior when the intended source is absent.

Memory: incremental evidence retrieval. The memory plugin changes how execution evidence is returned to the solver (Figure 11). The original implementation retrieves raw tool outputs by lexical overlap, which can repeatedly return information already present in the conversation. The mutated plugin deduplicates stored outputs and summarizes recent entries that it has not previously returned, together with guidance for subsequent verification. Retrieval thus emphasizes newly collected evidence while reducing repeated memory output.

Tool: failure-aware terminal feedback. The terminal tool adds interpretation of command output to its original truncation behavior (Figure 12). It inspects the complete output for execution and test failures, including those masked by a successful shell status, and appends recovery guidance. It also tracks the focused test command and its first failure diagnostic, prompting the solver to inspect the failure when the same pair recurs. On its screening minibatch, this mutation resolves five previously unsolved tasks with one regression.

(a) Before mutation

Memory: `implementation.py`

```
# Store every incoming output.
self.entries.append(item)

# Rank outputs by query overlap.
words = set(re.findall(
    r"\w+", str(query).lower()
))
ranked = sorted(
    enumerate(self.entries),
    key=lambda pair: (
        len(words & set(re.findall(
            r"\w+",
            pair[1]["content"].
  lower()
        ))),
        pair[0]
    ),
    reverse=True
)

# Return the highest-ranked outputs
  .
return [
    MemoryItem(**item)
    for _, item in
    ranked[:self.config["top_k"]]
]
```

(b) After mutation

Memory: `implementation.py`

```
# Skip duplicate outputs in update
  ().
if any(
    existing["content"] == content
    for existing in self.entries
):
    return

# Retrieve only undelivered entries
  .
unseen = self.entries[
    self._delivered_count:
]
self._delivered_count = len(self.
  entries)
if not unseen:
    return []

# Summarize recent unseen evidence.
selected = unseen[-top_k:]
classified = [
    self._summary(item)
    for item in selected
]
kinds = [
    kind for kind, _ in classified
]
# ... build checkpoint and guidance
```

Figure 11: Incremental evidence retrieval in the memory plugin. Mutation replaces retrieval by lexical overlap with deduplication and summaries of entries not previously returned. Code excerpts are reformatted and omit supporting logic.

(a) Before mutation

Tool: `implementation.py`

```
# Execute the requested command.
result = await (
    self.services.environment.
  execute(
        command,
        self.config.get(
            "timeout_seconds", 180
        )
    )
)

# Convert the output to text.
content = (
    result.content
    if isinstance(result.content,
  str)
    else str(result.content)
)

# Bound the output and return it.
result.content = self._truncate(
  content)
return result
```

(b) After mutation

Tool: `implementation.py`

```
# Interpret output before
  truncation.
# ... compute status/failure
  indicators
notice = self._evidence_notice(
    command, raw_content,
    reported_nonzero,
  explicit_failure
)
notice += self.
  _repeat_failure_notice(
    command, raw_content,
    reported_nonzero,
  explicit_failure
)
result.content = self._truncate(
    raw_content + notice
)
return result

# Inside _repeat_failure_notice():
# ... extract the failure
  diagnostic
focus = self._focused_test_command(
  command)
previous = self.
  _read_failure_checkpoint()
repeated = (
    previous.get("focus") == focus
    and previous.get("diagnostic")
        == diagnostic
)
self._write_failure_checkpoint(
    focus, diagnostic
)
# ... construct repeated-failure
  guidance
```

Figure 12: Failure-aware terminal feedback. The mutated tool interprets command output and tracks repeated failures to guide subsequent checks. Code excerpts are reformatted and omit supporting logic.

L TOKEN COST ANALYSIS

Figure 13 compares token usage between PluginRSI and Meta-Harness. We report total tokens per solver rollout, including proposer usage, and separately examine solver and proposer costs. For PluginRSI, proposer costs are further divided into plugin mutation and harness recomposition. Total input and output tokens per solver rollout increase by 7.5% and 19.8%, respectively. Solver token usage remains similar, while the additional mutation stage contributes to the proposer overhead.

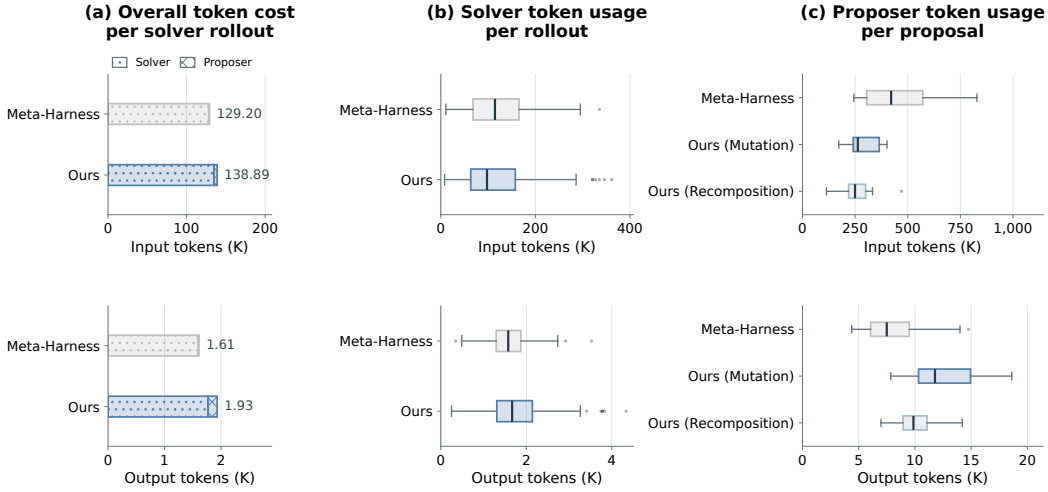


Figure 13: **Token cost analysis.** (a) Total token usage per solver rollout, including solver and proposer tokens. (b) Solver tokens per rollout. (c) Proposer tokens per proposal, with plugin mutation and harness recomposition reported separately.