RSI-HARNESS: OPTIMIZING THE HARNESS OPTIMIZER VIA RECURSIVE SELF-IMPROVEMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

The performance of an LLM agent depends heavily on its harness, which manages the model's context, tool use and control flow. Recent approaches automate harness design, yet the improver that edits the harness is written by hand and never changes, so it cannot learn from the outcomes of its past edits. Self-referential systems relax this restriction by letting the improver update itself, but it remains unclear whether they learn a better improver or merely find a better task harness along one search trajectory. We address this question by making the improver itself an explicit optimization target, so that it learns from its own search experience and becomes a better optimizer. We realize this idea in `RSI-Harness`, a dual-loop recursive self-improvement framework in which an inner loop uses the current improver to refine the task harness, while an outer loop revises the improver from the history of successful and failed attempts. Each revision may start from any archived improver, and an acceptance gate adopts it only if it produces a better task harness than the current improver under matched tasks and budget. Across five benchmarks and three frozen backbones, `RSI-Harness` achieves the best average score on every backbone, improving the initial harness by 7.7 points on average with 69% fewer optimization tokens on average than competing baselines. More importantly, the improver itself recursively gets better, with its gain on the same starting harness growing from 1.3 to 6.2 points. The final improver also transfers to unseen models, benchmarks and domains, gaining 2.7 times as much as the initial improver on average, while the optimized task harness transfers poorly.

1 INTRODUCTION

Agents built on large language models (LLMs) are now used in domains ranging from software engineering (Yang et al., 2024; Li et al., 2026a) and enterprise workflows (Xu et al., 2024; Yu et al., 2026) to scientific discovery (Novikov et al., 2025; Lu et al., 2026a). These agents comprise a foundation model that provides core knowledge and reasoning capacity, and an executable harness that governs context management, skill integration, tool interfaces, and control flow. An effective harness transforms abstract reasoning into grounded execution, enabling the agent to coordinate long-horizon tasks while adapting to environment feedback. Harness design can therefore substantially influence an agent's performance even when the underlying model remains unchanged (OpenAI, 2026; Chollet et al., 2026). Because manually engineering these domain-specific scaffolds requires extensive human expertise, automating harness design has become an essential problem (Hu et al., 2024; Lee et al., 2026b; Yang et al., 2026a).

Yet automating harness design does not necessarily automate the procedure that improves it. This meta-optimization matters because the improver determines which failures to inspect, how to interpret them, what candidate edits to propose, and where to allocate search effort. Many automated harness optimizers keep these decisions fixed throughout the search process (Khattab et al., 2023; Hu et al., 2024; Yuksekgonul et al., 2024; Agrawal et al., 2025; Lee et al., 2026b), even as they accumulate new evidence about which updates succeed or fail, causing the same search heuristics to be repeatedly applied and eventually leading optimization to plateau. More recent self-referential systems allow parts of the improvement procedure to self-update (Zelikman et al., 2023; Yin et al., 2024; Zhang et al., 2026b), but this raises a more fundamental question: does recursive adaptation produce a better procedure for improving harnesses, or does it merely discover a better harness along one particular search trajectory? This distinction matters because a task harness is often specialized to the setting in

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.

which it was optimized, whereas a stronger improver can be reused to optimize new harnesses under new conditions. We therefore ask whether a harness improver can learn from its own search decisions and recursively acquire a stronger, transferable ability to improve future harnesses.

To answer this question, we introduce `RSI-Harness`, a recursive self-improvement framework for agent harness optimization that makes the harness improver itself an explicit optimization target. Our key idea is to turn the experience generated during task-harness optimization into feedback for improving the procedure that performs that optimization, so that search capability itself can accumulate across generations. `RSI-Harness` realizes this idea through two coupled but distinct loops: an inner loop uses the current improver to refine the task harness from execution feedback, while an outer loop uses the accumulated optimization history to diagnose and revise the improver itself. The history retains both successful and failed search attempts, allowing the improver to reason over the consequences of its previous decisions and select any archived improver checkpoint as the parent for its next revision, rather than following a single greedy chain of self-edits. Each candidate improver is then passed through an acceptance gate, where it is compared with the current active improver under matched tasks and budget, and is promoted only if it produces a better task harness. Once accepted, the new improver takes over both subsequent task-harness optimization and future self-improvement, closing the recursive loop. Together, these mechanisms shift automated harness design from optimizing task harnesses alone to recursively improving the procedure that produces them, allowing optimization capability itself to improve over time.

We evaluate `RSI-Harness` across five diverse benchmarks spanning multiple domains and three frozen foundation-model backbones from different families and capability scales. `RSI-Harness` improves the initial harness by 7.7 points on average and achieves the best average performance on every backbone, while using 69% fewer optimization tokens than competing baselines. More importantly, frozen-checkpoint evaluation shows that the improver itself becomes progressively stronger, with average harness improvement increasing from 1.3 to 6.2 points across generations. This improvement capability also transfers across unseen models, benchmarks, and domains, where the final improver achieves 1.8–4.5$\times$ the gain of the initial one while the optimized task harness transfers poorly. Together, these results provide evidence that `RSI-Harness` recursively accumulates a transferable meta-optimization capability instead of merely producing a better task harness.

In summary, our contributions are as follows:

- We treat harness improvement itself as an optimization target, studying whether an improver can learn from its own search experience to recursively become a stronger optimizer.
- We introduce `RSI-Harness`, a dual-loop recursive self-improvement framework that couples task-harness refinement with recursive improvement of the improver itself.
- Across five benchmarks and three frozen backbones, `RSI-Harness` achieves the best average performance with 69% fewer tokens, and reveals strong meta-transferability, with the learned improver generalizing across unseen models, benchmarks, and domains.

2 Problem Formulation

Harness Optimization. We formulate automated harness design as an optimization problem over executable programs. Here, we distinguish two concrete optimization objects: (i) *a task harness* $T \in \mathcal{T}$ which governs how a foundation model executes tasks through context management, tool use, error recovery and verification routines and (ii) *an improver harness* $I \in \mathcal{I}$ which governs how the model examines optimization history, diagnoses failures, and proposes task harness modifications. Existing harness-optimization methods typically optimize T under a fixed, human-designed improvement procedure I (Agrawal et al., 2025; Lee et al., 2026b), which can lead to search stagnation as the same update strategy is repeatedly applied despite accumulating evidence about its own failures. We instead make I an explicit optimization target, using experience accumulated during task-harness optimization as empirical feedback for improving the search procedure itself. Formally, let $J_{\mathcal{D}}(T)$ denote the task performance achieved by harness T on task distribution $\mathcal{D}$. Starting from an initial harness T_0, executing the search procedure I on training tasks $\mathcal{D}_{train}$ produces an updated harness $T(I)$. The optimization problem is to identify an improver $I^\star$ that maximizes the expected marginal improvement of $T(I)$ over T_0 on the test distribution $\mathcal{D}_{test}$:

$$I^\star \in \arg\max_{I \in \mathcal{I}} \; \mathbb{E}[J_{\mathcal{D}_{test}}(T(I)) - J_{\mathcal{D}_{test}}(T_0)] \,.$$

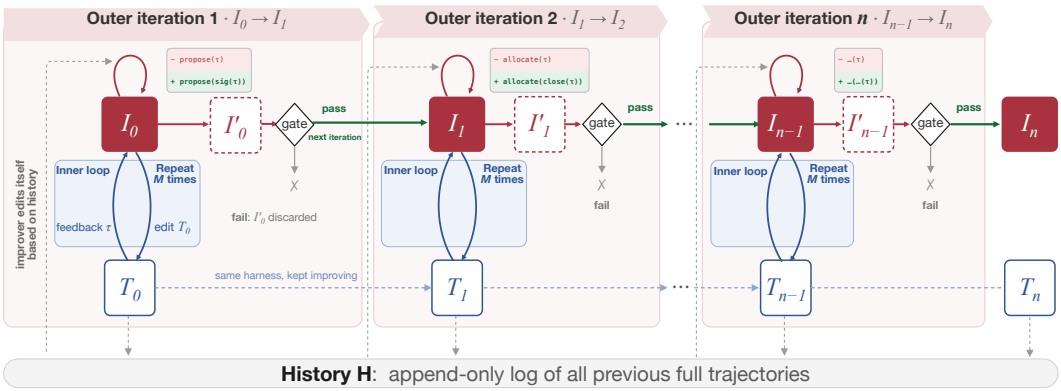

Figure 1: **Overview of `RSI-Harness`.** Each outer iteration k begins with the improver I_k and the task harness T_k. In the inner loop, I_k refines the task harness from its execution feedback τ for M steps. I_k then rewrites its own code to produce a candidate improver I'_k, with example edits shown above each candidate. The candidate passes the gate and becomes I_{k+1} only if, under the same budget, it produces a task harness that scores higher on the validation tasks than the one produced by I_k; otherwise it is discarded and I_k continues. The task harness carries over to the next iteration, and every attempt is recorded in the history $\mathcal{H}$, which the improver reads when it rewrites itself.

Self-Referential Systems. A self-referential system treats its own execution logic as editable data, allowing it to inspect, evaluate, and rewrite its own source code. Early work explored the evolution of program-modifying procedures themselves (Schmidhuber, 1987), while Gödel machines formalized systems that can rewrite their own code when an improvement can be established (Schmidhuber, 2003). Introducing a separate fixed meta-optimizer for the improver I would merely shift the harness-design problem one level upward. We therefore make I a self-referential program, where I acts as both the optimizer and the optimization target, directly applying its modification operators to its own code. At a high level, these two forms of improvement are written as:

$$T' \leftarrow I(T, \tau), \qquad I' \leftarrow I(I, \mathcal{H}),$$

where τ denotes task-execution trajectories and $\mathcal{H}$ the accumulated optimization experience. Once an improved I is accepted, it takes over subsequent task-harness optimization and future self-improvement, so improvements to the search procedure itself accumulate across generations.

3 RSI-Harness

In this section, we introduce `RSI-Harness`, a recursive self-improvement framework that optimizes agent harnesses through two interacting loops (Figure 1). In the inner loop, a frozen foundation model runs tasks under the current task harness T to collect execution traces, and then executes the improver harness I to diagnose failures and propose modifications to T. In the outer loop, the model periodically runs I to inspect the accumulated optimization history, select a historical improver state as the parent for self-modification, and synthesize a candidate improver I'_k. The candidate is then evaluated against the current improver I_k under an acceptance gate; once accepted, I'_k takes over the system, directing both subsequent updates to T and future self-referential modifications to I, thereby closing the recursive loop. `RSI-Harness` treats the improver as an explicit optimization target, recursively refining its strategy from accumulated history, while combining flexible parent selection with strict empirical gating to prevent stagnation and degenerative mutations.

3.1 Optimization Substrate and Execution Boundaries

`RSI-Harness` maintains both T and I as modular, standalone programs. The underlying foundation model remains frozen throughout the process, so improvement occurs entirely through changes to these harness programs. At outer generation k, the persistent system state consists of the active pair (T_k, I_k) together with an accumulated optimization history $\mathcal{H}_k$. $\mathcal{H}_k$ is an uncompressed, append-only record of every optimization trial. Each optimization step is stored as a structured record containing

execution trajectories, improver analyses, proposed edits, evaluation outcomes, promotion decisions, and resource statistics such as token usage and execution cost. Both accepted and rejected attempts are kept, so later comparisons can account for performance–cost trade-offs.

The two harnesses operate under distinct access boundaries. During task execution, T interacts only with the task environment and the tools exposed to the task agent. The improver can inspect the editable harness artifacts and the optimization evidence of its loop, and can propose modifications to either T or, during self-improvement, its own code. However, proposed edits cannot directly overwrite the active system. Each candidate is instantiated in an isolated workspace and becomes active only after passing its corresponding acceptance gate. This separation keeps the editable search state distinct from the fixed execution and evaluation machinery that controls candidate promotion.

3.2 Inner Loop: Task-Harness Optimization

The inner loop optimizes the task harness over consecutive iterations while holding the improver fixed. At each inner step m, the frozen foundation model executes the current task harness T_m on one training task, producing a full execution trajectory τ_m. This trajectory records the model's interaction with the harness and task environment, including intermediate actions, tool use, errors, and final task outcomes. The active improver I_k then diagnoses failures from the trajectory and proposes two candidate updates to the current harness:

$$\widetilde{T}_{m+1} \leftarrow I_k(T_m, \tau_m).$$

The improver may modify any editable component of the task harness, ranging from local prompt or context changes to structural changes in tool use, control flow, recovery, and verification routines. Before deployment, each candidate is evaluated against the current harness through a lightweight acceptance gate that checks whether the proposed modification yields improvement without introducing regressions on the validation split $\mathcal{D}_{val}$. Candidates that pass the gate replace the current task harness, while rejected candidates leave T_m unchanged. Each attempt is added to $\mathcal{H}$, providing evidence for subsequent task harness updates and improver self-modifications.

3.3 Outer Loop: Recursive Improver-Harness Optimization

The outer loop periodically optimizes the improver itself using the experience accumulated by the inner loop. Every M task-harness optimization steps, the active improver I_k initiates a self-improvement step by inspecting the persistent history $\mathcal{H}_k$. From past task trajectories and earlier self-modification records, I_k identifies recurring search bottlenecks, such as repetitive patch rejections, diagnostic blind spots, or plateaued task gains, and then performs parent selection. Rather than restricting each self-improvement step to modifying the current improver, I_k *can select any previously archived improver checkpoint* I_{base} *as the parent for the next candidate*, allowing the search to branch from earlier states and revisit promising designs using newly accumulated experience. Using the selected parent I_{base} and diagnostic evidence gathered from $\mathcal{H}_k$, the active improver I_k synthesizes a candidate from I_{base}:

$$I'_k \leftarrow I_k(I_{\text{base}}, \mathcal{H}_k).$$

Here, $I_{\text{base}} \in \{I_0, \ldots, I_k\}$ is the archived checkpoint selected as the parent, and I'_k is the resulting candidate. The update may revise any editable component of I_{base}, i.e., how it diagnoses failures, consults history, proposes task-harness edits, allocates search effort, and revises its own code.

Before modifying the active system, the candidate I'_k must pass an acceptance gate. We first check the candidate for structural validity to ensure the code parses into a valid abstract syntax tree and compiles cleanly. We then compare the candidate with I_k from identical copies of the current task harness and history, using the same training tasks in $\mathcal{D}_{train}$ under an identical compute budget. The candidate I'_k is accepted and becomes the new active improver I_{k+1} if and only if its resulting task harness achieves a higher validation score on $\mathcal{D}_{val}$:

$$I_{k+1} = \begin{cases} I'_k, & \text{if } J_{\mathcal{D}_{\text{val}}}(T(I'_k)) > J_{\mathcal{D}_{\text{val}}}(T(I_k)), \\ I_k, & \text{otherwise.} \end{cases} \tag{1}$$

If I'_k crashes or fails to outperform the current improver, it is discarded, the outcome is recorded in $\mathcal{H}_k$, and optimization resumes with I_k. This conservative criterion ensures that improver revisions take over only when they produce demonstrable gains from the current search frontier, preventing degenerate modifications from corrupting the optimization trajectory.

	Group	Harness	SWE-bench Pro	Terminal-Bench	τ^3-bank	Spreadsheet	GAIA-2	Average
gpt-5.5	*manual*	task-specific[§]	58.6	78.0	39.0	52.0	50.3	55.6
		general CLI	59.0 (+0.4)	<u>83.1</u> (+5.1)	33.2 (−5.8)	50.7 (−1.3)	30.9 (−19.4)	51.4 (−4.2)
	fixed-optimizer	GEPA	59.7 (+1.1)	80.6 (+2.6)	<u>40.1</u> (+1.1)	58.2 (+6.2)	49.4 (−0.9)	57.6 (+2.0)
		SkillOpt	58.9 (+0.3)	77.9 (−0.1)	38.9 (−0.1)	**67.4** (+15.4)	47.9 (−2.4)	<u>58.2</u> (+2.6)
		Meta-Harness	54.9 (−3.7)	75.1 (−2.9)	38.4 (−0.6)	59.1 (+7.1)	<u>50.8</u> (+0.5)	55.7 (+0.1)
	self-referential optimizer	STOP	56.4 (−2.2)	76.2 (−1.8)	33.5 (−5.5)	52.8 (+0.8)	48.9 (−1.4)	53.6 (−2.0)
		Hyperagents	59.2 (+0.6)	72.9 (−5.1)	31.9 (−7.1)	49.6 (−2.4)	50.1 (−0.2)	52.7 (−2.8)
		Self-Harness	<u>61.2</u> (+2.6)	81.8 (+3.8)	39.7 (+0.7)	51.2 (−0.8)	50.6 (+0.3)	56.9 (+1.3)
	ours	**RSI-Harness**	**63.2** (+4.6)	**84.0** (+6.0)	**42.9** (+3.9)	<u>66.8</u> (+14.8)	**54.6** (+4.3)	**62.3** (+6.7)
Qwen3.5-122B	*manual*	task-specific[§]	33.8	39.9	15.3	48.3	16.9	30.8
		general CLI	31.2 (−2.6)	36.8 (−3.1)	13.0 (−2.3)	44.6 (−3.7)	8.7 (−8.2)	26.9 (−4.0)
	fixed-optimizer	GEPA	<u>37.8</u> (+4.0)	42.3 (+2.4)	17.8 (+2.5)	58.1 (+9.8)	18.2 (+1.3)	34.8 (+4.0)
		SkillOpt	36.9 (+3.1)	39.1 (−0.8)	13.4 (−1.9)	**67.7** (+19.4)	15.4 (−1.5)	34.5 (+3.7)
		Meta-Harness	34.5 (+0.7)	<u>47.6</u> (+7.7)	19.6 (+4.3)	59.5 (+11.2)	18.9 (+2.0)	36.0 (+5.2)
	self-referential optimizer	STOP	29.6 (−4.2)	36.9 (−3.0)	10.6 (−4.7)	49.8 (+1.5)	16.1 (−0.8)	28.6 (−2.2)
		Hyperagents	37.1 (+3.3)	38.2 (−1.7)	12.2 (−3.1)	55.1 (+6.8)	17.8 (+0.9)	32.1 (+1.2)
		Self-Harness	37.4 (+3.6)	45.0 (+5.1)	<u>19.9</u> (+4.6)	61.2 (+12.9)	<u>20.0</u> (+3.1)	<u>36.7</u> (+5.9)
	ours	**RSI-Harness**	**41.3** (+7.5)	**49.9** (+10.0)	**21.6** (+6.3)	<u>65.4</u> (+17.1)	**21.2** (+4.3)	**39.9** (+9.0)
Gemma-4-31B	*manual*	task-specific[§]	24.0	32.8	12.6	43.7	32.5	29.1
		general CLI	22.1 (−1.9)	30.2 (−2.6)	10.7 (−1.9)	40.3 (−3.4)	16.8 (−15.7)	24.0 (−5.1)
	fixed-optimizer	GEPA	**29.9** (+5.9)	37.4 (+4.6)	13.7 (+1.1)	51.3 (+7.6)	32.1 (−0.4)	32.9 (+3.8)
		SkillOpt	27.1 (+3.1)	31.2 (−1.6)	9.7 (−2.9)	<u>55.6</u> (+11.9)	35.2 (+2.7)	31.8 (+2.6)
		Meta-Harness	26.4 (+2.4)	34.0 (+1.2)	13.2 (+0.6)	52.0 (+8.3)	33.2 (+0.7)	31.8 (+2.6)
	self-referential optimizer	STOP	18.2 (−5.8)	22.9 (−9.9)	12.7 (+0.1)	46.3 (+2.6)	33.0 (+0.5)	26.6 (−2.5)
		Hyperagents	22.9 (−1.1)	33.7 (+0.9)	5.7 (−6.9)	39.6 (−4.1)	<u>35.5</u> (+3.0)	27.5 (−1.6)
		Self-Harness	27.1 (+3.1)	<u>38.1</u> (+5.3)	<u>15.4</u> (+2.8)	50.9 (+7.2)	34.6 (+2.1)	<u>33.2</u> (+4.1)
	ours	**RSI-Harness**	<u>29.3</u> (+5.3)	**40.6** (+7.8)	**16.2** (+3.6)	**59.4** (+15.7)	**36.8** (+4.3)	**36.5** (+7.3)

Table 1: **Main results across benchmarks, backbones, and baselines.** Each benchmark is scored with its official metric (%), averaged over three runs. All optimizers start from the task-specific harness T_0 (first row of each block); the number in parentheses is the difference from T_0. Bold and underline mark the best and second-best result per column. [§]The T_0 of each benchmark is listed in Appendix I. Seed-level variability and confidence intervals are reported in Appendix F.

4 Experiments

4.1 Setup

Benchmarks and evaluation. We evaluate `RSI-Harness` across five agentic benchmarks spanning diverse operational domains: software engineering (**SWE-bench Pro** (Deng et al., 2025)), terminal interaction (**Terminal-Bench** (Merrill et al., 2026)), tool-mediated banking workflows (τ^3**-bank** (Yao et al., 2024; Shi et al., 2026)), tabular manipulation (**SpreadsheetBench** (Ma et al., 2024)), and generalist assistant coordination (**GAIA-2** (Froger et al., 2026)). For each benchmark, we partition available tasks into training, validation, and test splits with a 7:1:2 ratio. The training split $\mathcal{D}_{train}$ is used for inner-loop harness optimization, the validation split $\mathcal{D}_{val}$ is reserved for candidate selection and acceptance gating, and the held-out test split $\mathcal{D}_{test}$ remains strictly unseen until final evaluation. We report each benchmark's standard success metric and their mean. Benchmark details and dataset statistics are given in Appendix H, and the evaluation protocol in Appendix J.

Baselines and Backbones. We compare `RSI-Harness` against four families of baselines: 1) the task-specific harness of each benchmark, which is also the starting point T_0 of every automatic method (Appendix I); 2) a general-purpose CLI coding agent (OpenAI, 2025); 3) fixed-optimizer methods, whose hand-written optimizer never changes, including GEPA (Agrawal et al., 2025), SkillOpt (Yang

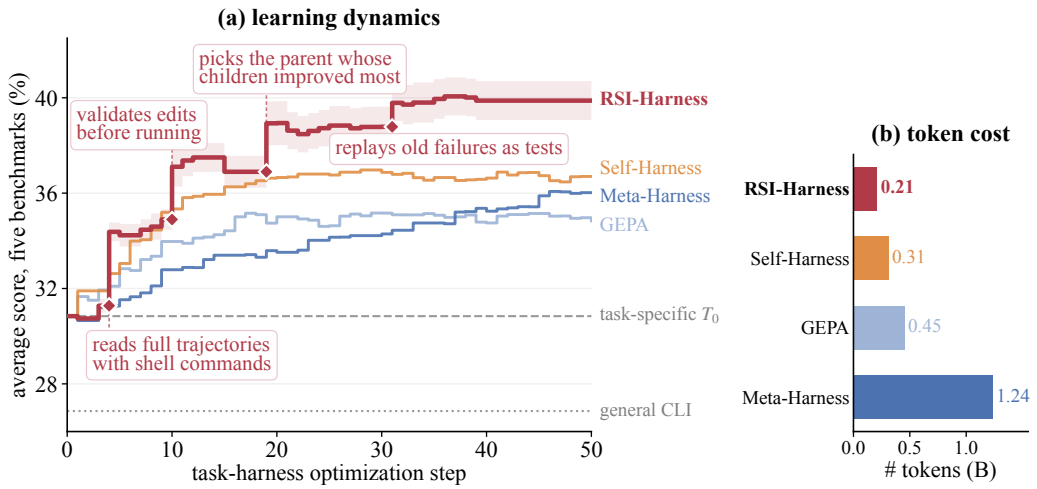

Figure 2: **Learning curves and token cost during optimization.** All runs use `Qwen3.5-122B`, averaged over the five benchmarks and 3 seeds. (a) Best-so-far score of each optimizer over steps. Diamonds mark four of the accepted improver updates of `RSI-Harness`, annotated with the harness change. (b) Token cost during optimization. `RSI-Harness` needs the fewest tokens.

et al., 2026b), and Meta-Harness (Lee et al., 2026b); and 4) self-referential optimizers, which can also modify themselves, including STOP (Zelikman et al., 2023), Hyperagents (Zhang et al., 2026b), and Self-Harness (Zhang et al., 2026a); all baselines are described in Appendix I. We evaluate all methods with three frozen foundation-model backbones, `gpt-5.5`, `Qwen3.5-122B-A10B`, and `Gemma-4-31B`, covering different model families and capability levels. Model weights remain fixed throughout optimization, isolating improvements to changes in the harness.

Implementation Details. We evaluate automatic methods under matched task access and comparable optimization budgets, using the same data splits and backbone within each comparison. All automatic methods start from the same task-specific harness T_0; `RSI-Harness` additionally starts from a seed improver I_0 that implements a naive zero-shot error diagnosis and patch synthesis routine. We bound the optimization by a fixed total of $N = 50$ task-harness optimization steps. For `RSI-Harness`, we set the outer-loop interval to $M = 3$. All trials are executed within isolated Docker containers subject to strict wall-clock and token-consumption limits. Full settings and prompts are in Appendices J and K, and Appendix C analyzes the effect of the outer-loop interval M.

4.2 Results

`RSI-Harness` achieves the best average score on every backbone. As Table 1 shows, `RSI-Harness` achieves the best average on all three backbones, improving on the starting harness T_0 by at most 9.0 points on average, and it outperforms a production coding-agent harness by at least 10.9 points. The gains are largest on `Qwen3.5-122B` and smallest on `gpt-5.5`. The baselines fall short in two ways. Fixed optimizers are sensitive to where their hand-designed strategy applies; for example, SkillOpt performs strongly on SpreadsheetBench but offers limited gains elsewhere. Self-referential optimizers are less stable, with STOP falling below T_0 on average across all three backbones. In contrast, `RSI-Harness` is the only method that improves on T_0 everywhere; this gain over T_0 is statistically supported on every backbone, while its margin over the strongest baseline is within the uncertainty of the small test sets (Appendix F).

Improver revisions drive the gains of `RSI-Harness`, at a lower token cost. Figure 2a shows the learning dynamics of `Qwen3.5-122B` during optimization with three runs. Each rise of `RSI-Harness` follows an accepted revision of the improver, usually after several task-harness edits have stopped helping. The learned revisions include reading full trajectories with shell commands, validating edits before running them, choosing the parent from past outcomes, and replaying old

Configuration	Task score	Δ vs. full	Improver gain
RSI-Harness (full)	65.4	—	+8.8
w/o outer loop (fixed I_0)	50.1	−15.3	+1.7
w/o recursion (I_0 edits I)	52.3	−13.1	+3.9
w/o parent selection (greedy)	59.3	−6.1	+6.3
w/o outer gate (accept all)	62.2	−3.2	+5.0

Table 2: **Component ablation of RSI-Harness.** Each ablation is carried out using `Qwen3.5-122B` on SpreadsheetBench. Task score is the success rate of the final harness. Improver gain is how much the run's final improver, frozen and restarted from T_0, improves over T_0. Removing any component lowers both task score and improver gain.

	Seed harness T_0	Initial score	Final score	Gain
Terminal-Bench	NexAU$_0$	12.5	37.4	+24.9
	mini-SWE-agent	39.9	49.9	+10.0
	Terminus 2	35.2	43.2	+8.0
	OpenHands	32.6	38.7	+6.1
Spreadsheet	mini-SWE-agent	27.1	53.2	+26.1
	ReAct + xlsx skill	48.3	65.4	+17.1
	OpenHands	30.4	42.2	+11.8

Table 3: **Robustness to the initial task harness.** Each run is initialized with a different seed harness T_0 (Appendix I) using `Qwen3.5-122B` on Terminal-Bench and SpreadsheetBench. `RSI-Harness` improves every seed.

failures as tests. Self-Harness improves fastest at first but soon levels off. All methods use the same rollouts, so the gain does not come from more compute. In Figure 2b, `RSI-Harness` also uses fewer tokens than the other optimizers, and it matches the final score of Self-Harness within 20 steps.

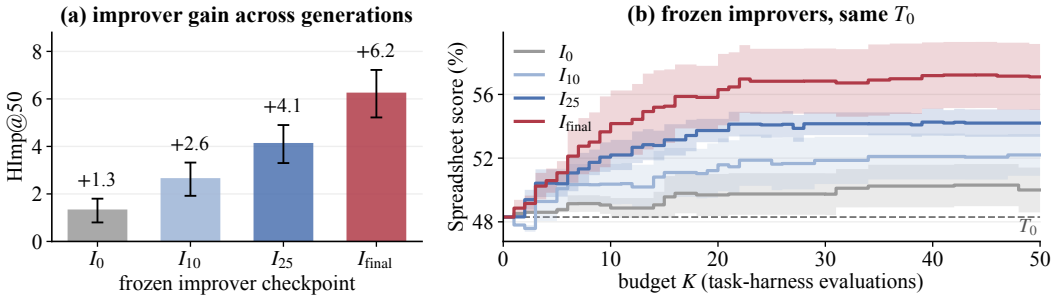

Figure 3: **Improvement capability of frozen improver checkpoints.** Averaged over 3 runs with `Qwen3.5-122B`. I_t is the improver after t task-harness steps of one `RSI-Harness` run, and I_{final} is the last one, I_{50}. Each checkpoint is frozen and restarted from the same T_0 with a budget of K=50 steps. (a) HImp@50, the gain over T_0 after 50 steps, averaged over the five benchmarks. (b) SpreadsheetBench score as the budget grows. Later checkpoints improve the same T_0 faster and further, and this order holds on every benchmark.

4.3 Analysis

Meta-level optimization and recursion drive most of the task-success improvement, while parent selection and the outer gate yield a stronger improver. Table 2 removes one component of the outer loop at a time on SpreadsheetBench. Freezing the improver at I_0 removes about 90% of the improvement over T_0, and letting I_0 write every later improver, instead of having the improver revise itself, also lowers the task score by 13.1 points. Most of the gain therefore comes from optimizing the improver and from letting it rewrite its own code. Removing parent selection or the outer gate costs less task success, but both leave a weaker improver. These two components mainly keep bad improver revisions from being built upon, and they also affect task performance. More comparisons, including a strong fixed improver built from the learned procedures, can be found in Appendix D.

RSI-Harness is robust to the initial harness, and leaner initial harnesses benefit the most. Table 3 starts `RSI-Harness` from harnesses of different sizes on Terminal-Bench and Spreadsheet-Bench. Every starting harness improves. The gain is largest for the smallest harnesses, NexAU$_0$ and mini-SWE-agent, and smallest for OpenHands, the largest codebase. This demonstrates that

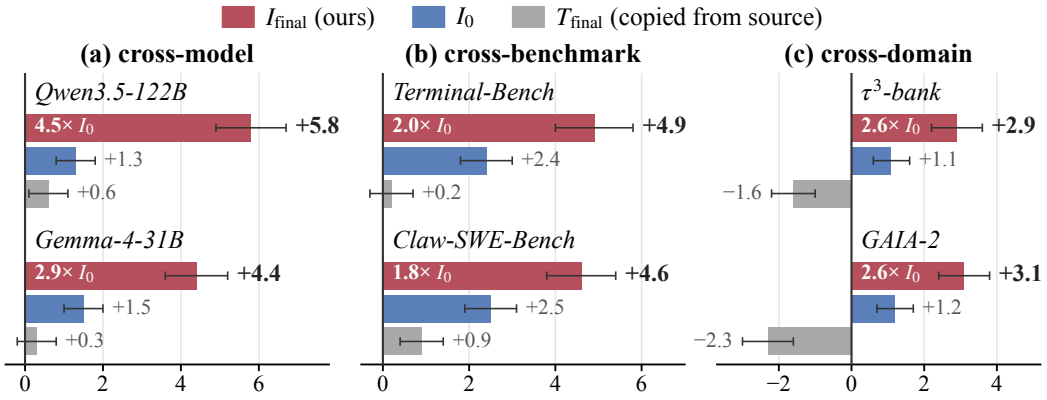

Figure 4: **Transferability of the learned improver across backbones, benchmarks, and domains**. The improver learned on SWE-bench Pro with `gpt-5.5` is applied without further updates to (a) other backbones, (b) other coding benchmarks and (c) other domains. Bars show the gain over each target's T_0 for the learned improver I_{final}, the seed improver I_0 under the same budget, and the task harness optimized on the source. Labels inside the red bars give the ratio to I_0. I_{final} beats I_0 everywhere, while the copied task harness helps little and hurts in new domains.

`RSI-Harness` is robust to initialization, consistently improving harnesses across a wide range of starting complexities rather than relying on a carefully chosen seed.

Recursive self-improvement yields cumulative gains in optimizer capability. Figure 3 shows that the improver itself becomes a better optimizer over the run. We take I_0, I_{10}, I_{25} and I_{final} from a completed run, freeze them, and let each optimize the same T_0 with the same budget. There is a clear trend that later improvers produce larger gains, from 1.3 points for I_0 to 6.2 for I_{final} on average, and this order holds on every benchmark (Table 12); Appendix E repeats the test on a fresh split. This demonstrates a compounding effect of recursive self-improvement, where optimization capability accumulates across revisions rather than being confined to improvements in the current task harness.

The learned improver transfers well across models, benchmarks, and domains. Figure 4 freezes the optimized improver harness and applies it to new backbones, benchmarks and domains. It is compared with the seed improver I_0 and with the optimized task harness T_{final} copied from the source setting. The learned improver gains 1.8 to 4.5 times as much as I_0 in every setting. The copied harness helps little on new models and benchmarks, and in new domains it does worse than T_0. This result demonstrates a notable form of meta-generalization: `RSI-Harness` learns an optimization capability that transfers across models, benchmarks, and domains, allowing the improved optimizer itself, not just the optimized task harness, to remain effective in entirely new settings.

`RSI-Harness` learns to diagnose failures from experience, while a fixed optimizer such as GEPA does not. Figure 5 contrasts GEPA and `RSI-Harness` on the same SpreadsheetBench failures. GEPA interprets empty cells as a formula error and rewrites the prompt accordingly. In contrast, the learned improver retrieves similar failures from its history, observes that they span different formulas, and identifies the true cause: the workbook was never recalculated. It therefore adds recalculation and read-back verification, solving 12 of 14 tasks versus one for GEPA. This case illustrates how recursive self-improvement changes the optimizer's reasoning procedure itself, enabling it to accumulate and reuse diagnostic knowledge across failures. More analyses are in Appendix A. Extended case studies can be found in Appendix B.

5 Related Work

Automatic harness engineering. Recent work optimizes the harness around a frozen LLM, either its code (Lee et al., 2026b; Hu et al., 2024; Zhang et al., 2024; Shang et al., 2024; Yang et al., 2026a; Cheng et al., 2024) or the prompts, context and skills it carries (Agrawal et al., 2025; Zhang et al.,

Task (SpreadsheetBench). Fill `D2:D40` with the total sales per region and month.

GEPA

Task-harness edit system prompt

```
+ "Before saving, check formula syntax
+  and ranges. If a formula returns
+  empty values, rewrite it more simply.
+  Prefer SUMIF over SUMIFS and avoid
+  whole-column ranges such as C:C."
```

× **Wrong fix:** the formula was correct; the cells were empty because the workbook was not recalculated.
× 1 of 14 tasks solved; 38% more actions spent rewriting formulas.

Improver harness hand-set, GEPA default

```
I provided an assistant with ...:
  <curr_param>
The following are examples ...:
<side_info>  3 trajectories
Your task is to write a new instruction
for the assistant. ...
```

× **Wrong diagnosis:** from 3 trajectories, the empty cells look like a formula error.

`RSI-Harness` (ours)

Task-harness edit code

```
+ def after_write(path, cells):
+     recalc(path)  # recompute first
+     empty = [c for c in cells
+              if value(path, c) is None]
+     if empty:
+         return f"still empty: {empty}"
```

✓ **Right fix:** recalculate after writing, then read the cells back.
✓ 12 of 14 tasks solved; 9% fewer actions, as formulas are no longer rewritten.

Improver harness learned, in I_{final}

```
+ sig   = top_cluster(H.failures,
+                      key=error_signature)
+ hits  = grep(H, sig)
+ tried = H.rejected.targeting(sig)
+ return propose(summarize(hits, tried),
+                target=sig)
```

✓ **Right diagnosis:** 14 similar failures span 4 different formulas, so the formula is not the cause.

Figure 5: **Case study: GEPA and `RSI-Harness` on the SpreadsheetBench trajectory**. The agent writes a correct formula but reads cells before the workbook is recalculated, sees empty values and keeps rewriting the formula. Top: the task-harness edit each method proposes with results. Bottom: how each improver selects evidence. GEPA's sees 3 trajectories and blames the formula; the learned improver of `RSI-Harness` groups all similar failures and adds a recalculation step.

2025b; Yang et al., 2026b; Khattab et al., 2023; Yuksekgonul et al., 2024). These methods differ in what they edit, but in all of them the improver that proposes the changes is written by hand and never revised. We keep a similar inner loop and learn that procedure instead.

Meta-optimization. Schmidhuber (1987) proposed learners that improve their own learning procedure. Later work trained optimizers as neural networks (Andrychowicz et al., 2016; Li & Malik, 2016; Metz et al., 2022) and used LLMs as optimizers (Yang et al., 2023), in some cases evolving the mutation prompts as well (Fernando et al., 2023). These methods learn a parametric or textual update rule, usually by meta-training across tasks, whereas our improver is a code program.

Recursive self-improvement systems. Since the Gödel machine (Schmidhuber, 2003), self-improving systems have moved from proofs to empirical tests, and LLM agents now rewrite their own scaffolding or code (Zelikman et al., 2023; Yin et al., 2024; Zhang et al., 2025a; Wang et al., 2025; Zhang et al., 2026b;a; Lee et al., 2026a). Most of them do not separate the improver from the agent it improves, and few test a new improver against the previous one before adopting it. We keep the two apart and gate every change to the improver, because a bad improver edit affects all later generations. Due to space limits, Appendix G provides an extended review of all three lines of work.

6 Conclusion

In this paper, we study whether the procedure that improves agent harnesses can itself be improved. We answer this question with `RSI-Harness`, which makes the improver an explicit optimization target and adopts each self-revision only when it produces a better task harness under matched tasks and budget. Across five benchmarks and three frozen backbones, `RSI-Harness` achieves the best average score on every backbone with 69% fewer optimization tokens. Most notably, the improver recursively becomes a stronger optimizer, and it transfers to unseen models, benchmarks and domains, while the optimized task harness transfers poorly. More broadly, `RSI-Harness` offers a practical and measurable path toward recursive self-improvement in agent systems.

AI USE STATEMENT

We did not use generative AI tools for any task with required disclosure: generating synthetic data, developing theoretical models or conceptual frameworks, formulating or proving mathematical claims, proposing or refining hypotheses, designing or giving feedback on the methodology or experiments, implementing methods, translation, cleaning or reformatting data, qualitative data analysis, or interpreting results. We used generative AI tools only to edit the text of this paper to improve readability. All AI-assisted edits were reviewed by the authors, and we take responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

Section 3 specifies the task harness and the improver, the two optimization loops and their acceptance gates. Section 4.1 states the backbones, benchmarks and budget, and Appendices H to J give the benchmark splits, the baseline configurations and the full evaluation protocol, including the per-step settings and the frozen-improver evaluation. Appendix K gives the seed improver and its prompts. Code, optimized harnesses and the improver checkpoints used in Figure 3 will be released.

ETHICS STATEMENT

This work optimizes the scaffolding around frozen models on public agentic benchmarks in sandboxed environments. Recursive self-improvement of agent machinery warrants care: our design restricts the improver to edit its own proposal procedure and the distribution of a fixed gate budget, while the acceptance rule, split boundaries, and total evaluation budget are outside its reach, so that the system cannot optimize the measurement that governs it. We regard that boundary as a necessary property of any deployed system of this kind; it is specified in Section 3.

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, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning, 2025. URL https://arxiv.org/abs/2507.19457.

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando de Freitas. Learning to learn by gradient descent by gradient descent, 2016. URL https://arxiv.org/abs/1606.04474.

Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next AutoDiff: Generative optimization with rich feedback, execution traces, and LLMs, 2024. URL https://arxiv.org/abs/2406.16218.

François Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. ARC Prize 2025: Technical report, 2026. URL https://arxiv.org/abs/2601.10904.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution, 2023. URL https://arxiv.org/abs/2309.16797.

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, 2026. URL https://arxiv.org/abs/2602.11964.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems, 2024. URL https://arxiv.org/abs/2408.08435.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines, 2023. URL https://arxiv.org/abs/2310.03714.

Hyunin Lee, Jinglue Xu, Jeffrey Seely, Donghyun Lee, Matei Zaharia, and Yujin Tang. Recursive harness self-improvement, 2026a. URL https://arxiv.org/abs/2607.15524.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses, 2026b. URL https://arxiv.org/abs/2603.28052.

Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. AIDev: Studying AI coding agents on GitHub, 2026a. URL https://arxiv.org/abs/2602.09185.

Ke Li and Jitendra Malik. Learning to optimize, 2016. URL https://arxiv.org/abs/1606.01885.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, Chujun Tao, Binxu Li, Xuandong Zhao, Hejia Geng, Xiaojun Wu, Junwei Zhou, Xiaokun Chen, Hanwen Xing, Yubo Li, Qunhong Zeng, Di Wang, Yuanli Wang, Roey Ben Chaim, Penghao Jiang, Haotian Shen, Luyang Kong, Xinyi Liu, Runhui Wang, Xuanqing Liu, Jiachen Li, Xin Lan, Yueqian Lin, Wengao Ye, Junwei He, Songlin Li, Yue Zhang, Yipeng Gao, Yijiang Li, Ze Ma, Liqiang Jing, Tianyu Wang, Kaixin Li, Yiqi Xue, Haoran Lyu, Yizhuo He, Yuchen Tian, Shutong Wu, Bowei Wang, Yixuan Gao, Bo Chen, Litong Liu, Sikai Cheng, Jiajun Bao, Shuaicheng Tong, Shuwen Xu, Terry Yue Zhuo, Tinghan Ye, Qi Qi, Miao Li, Longtai Liao, Zelin Tan, Chang Shi, Xilin Tang, Srinath Tankasala, Boqin Yuan, Yaoyao Qian, Jianhong Tu, Chenguang Wang, Yizhou Sun, Wei Wang, Aaron Taylor, Ziyue Yang, Changkun Guan, Zhikang Dong, Xinyu Zhang, Steven Dillmann, Han-chung Lee, and Dawn Song. SkillsBench: Benchmarking how well agent skills work across diverse tasks, 2026b. URL https://arxiv.org/abs/2602.12670.

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, 2026a. URL https://arxiv.org/abs/2604.25850.

Minhua Lin, Juncheng Wu, Zijun Wang, Zhan Shi, Yisi Sang, Bing He, Zewen Liu, Tianxin Wei, Zongyu Wu, Zhiwei Zhang, Dakuo Wang, Xiang Zhang, Benoit Dumoulin, Cihang Xie, Yuyin Zhou, Suhang Wang, and Hanqing Lu. Harness updating is not harness benefit: Disentangling evolution capabilities in self-evolving LLM agents, 2026b. URL https://arxiv.org/abs/2605.30621.

Chris Lu, Cong Lu, Robert Tjarko Lange, Yutaro Yamada, Shengran Hu, Jakob Foerster, David Ha, and Jeff Clune. Towards end-to-end automation of AI research. *Nature*, 651(8107): 914–919, 2026a. doi: 10.1038/s41586-026-10265-5. URL https://doi.org/10.1038/s41586-026-10265-5.

Xinyu Lu, Tianshu Wang, Pengbo Wang, Zujie Wen, Zhiqiang Zhang, Jun Zhou, Boxi Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, and Le Sun. The Meta-Agent Challenge: Are current agents capable of autonomous agent development?, 2026b. URL https://arxiv.org/abs/2606.04455.

Zeyao Ma, Bohan Zhang, Jing Zhang, Jifan Yu, Xiaokang Zhang, Xiaohan Zhang, Sijia Luo, Xi Wang, and Jie Tang. SpreadsheetBench: Towards challenging real world spreadsheet manipulation, 2024. URL https://arxiv.org/abs/2406.14991.

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. URL https://arxiv.org/abs/2601.11868.

Luke Metz, James Harrison, C. Daniel Freeman, Amil Merchant, Lucas Beyer, James Bradbury, Naman Agrawal, Ben Poole, Igor Mordatch, Adam Roberts, and Jascha Sohl-Dickstein. VeLO: Training versatile learned optimizers by scaling up, 2022. URL https://arxiv.org/abs/2211.09760.

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

OpenAI. Codex CLI, 2025. URL https://github.com/openai/codex.

OpenAI. How enabling two settings tripled our scores on the ARC-AGI-3 benchmark. https://openai.com/index/how-two-settings-tripled-our-arc-agi-3-scores/, 2026. Blog post.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent, 2025. URL https://arxiv.org/abs/2504.15228.

Jürgen Schmidhuber. Evolutionary principles in self-referential learning, or on learning how to learn: The meta-meta-... hook. Diploma thesis, Institut für Informatik, Technische Universität München, 1987.

Jürgen Schmidhuber. Goedel machines: Self-referential universal problem solvers making provably optimal self-improvements, 2003. URL https://arxiv.org/abs/cs/0309048.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. AgentSquare: Automatic LLM agent search in modular design space, 2024. URL https://arxiv.org/abs/2410.06153.

Quan Shi, Alexandra Zytek, Pedram Razavi, Karthik Narasimhan, and Victor Barres. τ-Knowledge: Evaluating conversational agents over unstructured knowledge, 2026. URL https://arxiv.org/abs/2603.04370.

Wenyi Wang, Piotr Piękos, Li Nanbo, 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, 2025. URL https://arxiv.org/abs/2510.21614.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents, 2024. URL https://arxiv.org/abs/2407.16741.

Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. TheAgentCompany: Benchmarking LLM agents on consequential real world tasks, 2024. URL https://arxiv.org/abs/2412.14161.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers, 2023. URL https://arxiv.org/abs/2309.03409.

Chenyang Yang, Xinran Zhao, Tongshuang Wu, and Christian Kästner. Better harnesses, smaller models: Building 90% cheaper agents via automated harness adaptation, 2026a. URL https://arxiv.org/abs/2607.08938.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

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, 2026b. URL https://arxiv.org/abs/2605.23904.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL https://arxiv.org/abs/2406.12045.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel Agent: A self-referential agent framework for recursive self-improvement, 2024. URL https://arxiv.org/abs/2410.04444.

Tao Yu, Hao Wang, Changyu Li, Shenghua Chai, Minghui Zhang, Zhongtian Luo, Yuxuan Zhou, Haopeng Jin, Zhaolu Kang, Jiabing Yang, YiFan Zhang, Xinming Wang, Hongzhu Yi, Zheqi He, Jing-Shu Zheng, Xi Yang, Yan Huang, and Liang Wang. Beyond the all-in-one agent: Benchmarking role-specialized multi-agent collaboration in enterprise workflows, 2026. URL https://arxiv.org/abs/2605.08761.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. TextGrad: Automatic "differentiation" via text, 2024. URL https://arxiv.org/abs/2406.07496.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-Taught Optimizer (STOP): Recursively self-improving code generation, 2023. URL https://arxiv.org/abs/2310.02304.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-Harness: Harnesses that improve themselves, 2026a. URL https://arxiv.org/abs/2606.09498.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel Machine: Open-ended evolution of self-improving agents, 2025a. URL https://arxiv.org/abs/2505.22954.

Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents, 2026b. URL https://arxiv.org/abs/2603.19461.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation, 2024. URL https://arxiv.org/abs/2410.10762.

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, 2025b. URL https://arxiv.org/abs/2510.04618.

Xing Zhang, Yanwei Cui, Guanghui Wang, Ziyuan Li, Wei Qiu, Bing Zhu, and Peiyang He. Library drift: Diagnosing and fixing a silent failure mode in self-evolving LLM skill libraries, 2026c. URL https://arxiv.org/abs/2605.19576.

A WHAT DID THE IMPROVER LEARN?

Figure 3 shows that a frozen I_{final} improves task harnesses faster than I_0, and Figure 4 shows that the advantage holds in settings the improver was never optimized in. Neither result tells us what the outer loop changed in the improver. In this appendix we look at the accepted improver edits (Section A.1), at how they change the improver's search behavior (Section A.2), and at which of them account for the gain (Section A.3). A concrete example is given in Figure 5 of the main text. Unless stated otherwise, all numbers come from the `Qwen3.5-122B` runs of Table 1, with five benchmarks, three seeds per benchmark and $N{=}50$ task-harness steps. With an outer step every $M{=}3$ steps, a run contains up to 16 improver revisions.

A.1 WHICH EDITS DOES THE OUTER LOOP ACCEPT?

We assign each accepted improver edit to one category according to the part of the improver it changes (Table 4). Categories C1–C4 are the procedural components that the outer loop may rewrite (Appendix K and Section 3.3). C5 contains edits that only reword a prompt template. C6 contains edits that write knowledge about a single benchmark into the improver, which is a form of overfitting (Section 1). Two authors labeled every diff with its main category without knowing the benchmark or the generation it came from, and their agreement before adjudication was $\kappa{=}0.82$. We also screen for C6 automatically. A diff is flagged when it contains an identifier from a benchmark-specific vocabulary, such as a tool name, file format, API or dataset name, and every flagged diff is then checked by hand.

	Category	Typical accepted edit
C1	Evidence selection	which failures are mined and how they are summarized, e.g. clustering failed trajectories by error signature instead of taking the lowest-scoring ones
C2	Proposal generation	how candidates are requested, e.g. each candidate must name the failure cluster it targets; near-duplicate candidates are rejected before the gate
C3	Budget allocation	how the fixed gate budget is split, e.g. more rollouts to candidates that score close to the current harness
C4	History & parent selection	how $\mathcal{H}$ is consulted, e.g. reading rejected proposals before proposing; branching from an earlier checkpoint
C5	Prompt wording only	rewording of a prompt template with no change to the procedure
C6	Task-specific content	a rule, path, tool name or example specific to one benchmark

Table 4: **Categories of improver edits.** C1–C4 change the improver's procedure and C5 only its wording. C6 marks edits that write knowledge of one benchmark into the improver, a sign of overfitting.

Figure 6 shows three patterns. First, the improver learns similar edits on every benchmark. Each benchmark is optimized separately, yet the category shares differ across benchmarks by at most 5 points, and procedural edits (C1–C4) make up 85% of all accepted edits (158 of 185). Second, the improver rarely writes task knowledge into itself. Only 5 of the 185 accepted edits (2.7%) contain task-specific content, and no benchmark has more than one. The gate accepts these edits because they help on the benchmark being optimized, but they are rare enough that the improver remains a general procedure, which the transfer results of Figure 4 depend on. Third, the edits are learned in a consistent order. Early revisions mostly change which evidence is read (C1, 46% of the edits in generations 1–5), middle revisions change how candidates are requested (C2, 32%), and late revisions change how the budget is spent and how the history is used (C3 and C4 together, 57% in generations 11–16). This is the order one would expect if each revision removes the current bottleneck of the search, since splitting the budget between candidates only pays off once the candidates differ in quality. Wording-only edits (C5) are common early on and become rare as procedural edits take over.

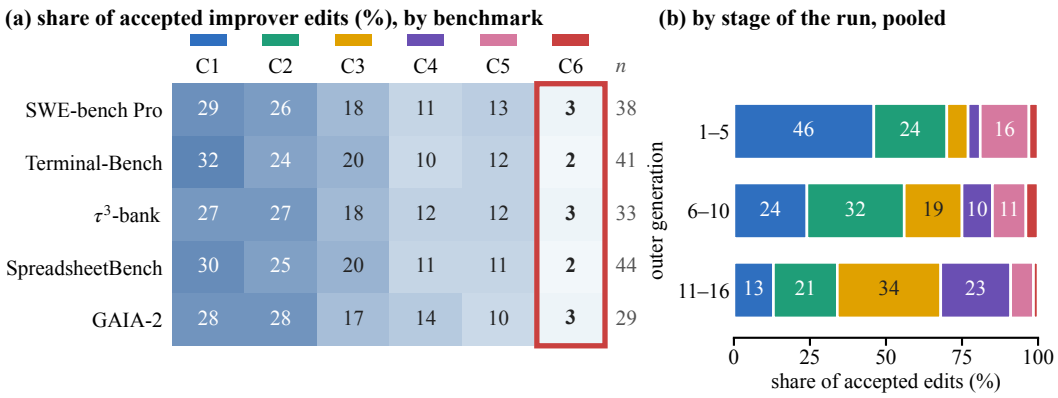

Figure 6: **Accepted improver edits by benchmark and stage**. (a) Share (%) of accepted improver edits in each category of Table 4, per benchmark, pooled over three seeds; n is the number of edits and the C6 column is outlined. (b) The same edits pooled over benchmarks and split by the outer generation in which they were accepted. The mix is similar across benchmarks, C6 stays rare, and later generations move from evidence selection to budget allocation and history use.

A.2 How does the improver's behavior change?

To see how the search itself changes, we replay the frozen checkpoints I_0, I_{10}, I_{25} and I_{final} of Figure 3, each restarted from the same T_0 with $K{=}50$, and record statistics of the search process rather than its final score (Table 5). All indicators change monotonically across checkpoints. Compared with I_0, I_{final} proposes fewer candidates that fail to run (23.5%→7.4%) and fewer near-duplicates (31.0%→9.8%), and most of its candidates name the failure cluster they target (8.2%→86.2%). The gate therefore accepts 2.8× as many of its candidates, and each accepted edit costs 230 gate rollouts instead of 652. The improver spends somewhat more tokens per step (38k→56k).

	I_0	I_{10}	I_{25}	I_{final}	
Candidates that fail to run (%)	23.5	16.2	11.0	**7.4**	↓
Near-duplicate candidates (%)	31.0	22.4	15.3	**9.8**	↓
Candidates naming a target failure (%)	8.2	41.3	68.5	**86.2**	↑
Gate acceptance rate (%)	6.1	9.4	12.8	**17.3**	↑
Gate rollouts per accepted edit	652	423	311	**230**	↓
Improver tokens per step (k)	38	44	51	56	↓

Table 5: **Search behavior of frozen improver checkpoints.** Each checkpoint is replayed from the same T_0 for 50 steps (`Qwen3.5-122B`, mean over five benchmarks and three seeds); the last column gives the better direction. Later checkpoints propose fewer broken and duplicate candidates, target a named failure more often and need fewer rollouts per accepted edit, while using somewhat more tokens per step. A candidate counts as a near-duplicate if its diff is more than 90% similar to an earlier one.

The type of task-harness edit also changes (Figure 7). Most proposals of I_0 reword prompts or context (58%), compared with 24% for I_{final}. I_{final} instead changes control flow more often and adds verification and recovery steps (18%→45%). An optimizer that only edits text cannot make such structural edits, and they are a likely reason why `RSI-Harness` outperforms the text-artifact optimizers in Table 1.

A.3 Which learned edits carry the gain?

The two previous subsections describe what changed but do not show that these changes matter. We therefore remove one category at a time. Starting from I_{final}, we revert all accepted edits of that category in reverse order and keep the later edits of other categories (3 of the 44 accepted edits, pooled over the three seeds, needed a manual merge). The resulting improver is frozen and restarted

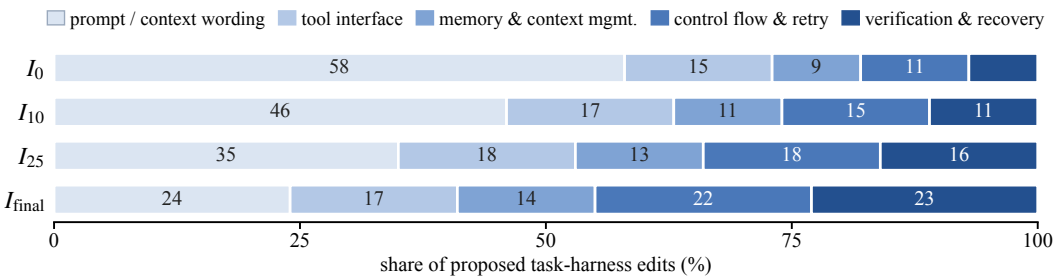

Figure 7: **Task-harness edits proposed by each checkpoint**. Share of proposals by type, ordered from local (prompt wording) to structural (control flow, verification); same runs as Table 5. Later checkpoints move from rewording prompts to structural edits.

from T_0 under the frozen-improver protocol of Appendix J on SpreadsheetBench, the benchmark used for the ablations in Table 2.

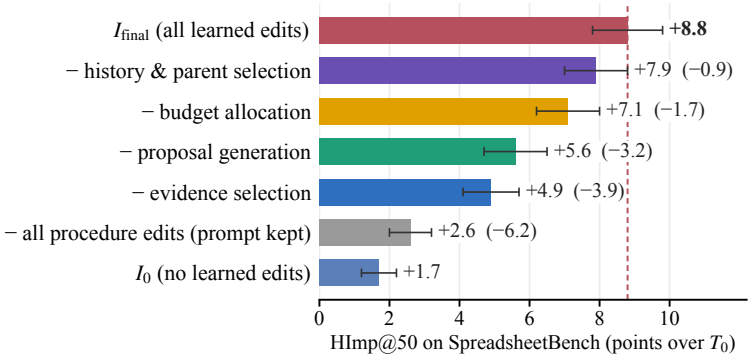

Figure 8: **Knockout of learned improver edits**. Each bar removes one category of learned edits (Table 4) from I_{final} and replays it from T_0 on SpreadsheetBench (`Qwen3.5-122B`, mean ± std over three seeds); parentheses give the drop from I_{final}. Removing evidence selection hurts most, and keeping only the prompt-wording edits recovers little of the gain over I_0.

Removing any of the procedural categories lowers the score, and the size of the drop follows the order in which the categories are learned (Figure 8). Reverting evidence selection costs the most (-3.9), followed by proposal generation (-3.2), budget allocation (-1.7) and history use (-0.9). The four drops sum to 9.7 points, which is more than the 7.1-point gap between I_{final} and I_0, so the categories interact. For example, a better budget rule helps little when the candidates it allocates between are not aimed at specific failures. Keeping only the wording edits recovers 0.9 of the 7.1 points (13%), so most of the gain comes from changes to the procedure rather than to the prompt wording.

B Additional case studies

Figure 5 compares `RSI-Harness` with GEPA on one SpreadsheetBench failure. We give three more cases in the same format, each against a different baseline and on a benchmark where that baseline improves on T_0 by at most 3.1 points or falls below it (Table 1). Each case isolates a different part of `RSI-Harness`. Against SkillOpt, the right rule is easy to write down, but an edit to the skill text can state it without enforcing it (Figure 9). Against Self-Harness, a fixed procedure that summarizes failures by their symptom keeps proposing edits that miss the cause, while the learned improver reads the proposals already rejected for that failure and the relevant part of the trajectory (Figure 10). Against STOP, a self-edit that reads less of each trajectory is adopted without comparison with the previous improver; in `RSI-Harness`, the same kind of edit must beat I_k from the same T and $\mathcal{H}$, and it is rejected (Figure 11). As in Figure 5, the top half of each case shows the task-harness edit with its result, and the bottom half shows the improver behavior that produced it.

Task (τ^3-bank). The customer reports a damaged credit card and asks for a replacement.

SkillOpt

Task-harness edit skill text

```
+ ## Account actions
+ - Before any account change, verify two
+   of: date of birth, email, phone, address,
+   then call log_verification.
+ - Never order a card before this step.
```

× **Rule not enforced:** the rule is right, but in long dialogues that first search the knowledge base for the replacement tool, the agent still orders the card before logging verification.
× 2 of 7 tasks solved.

Improver harness hand-set, SkillOpt default (schematic)

```
given: scored rollouts (fail + success), skill
reflect on common failure patterns
return: ≤4 add/replace/delete edits   text only
```

× **Text only:** every edit changes the skill text, so the rule can be stated but not enforced.

RSI-Harness (ours)

Task-harness edit code

```
+ WRITES = state_changing_tools()
+ def before_call(tool, log):
+     if tool in WRITES and \
+        not log.done("log_verification"):
+         return "verify the customer first"
```

✓ **Right fix:** refuse state-changing calls until verification is logged; the agent then verifies and calls again.
✓ 5 of 7 tasks solved.

Improver harness learned, in I_{final}

```
+ sig  = top_cluster(H.failures,
+              key=tool_call_order)
+ cands = propose(summarize(sig), n=4)
+ cands = [c for c in cands
+           if replay(c, sig.tasks).fixed]
```

✓ **Right diagnosis:** in 9 failures a write call comes before `log_verification` although the rule is known, so it has to be enforced in the tool layer; replaying the old failures confirms the fix before the gate.

Figure 9: **Case study: SkillOpt and RSI-Harness on τ^3-bank.** The agent orders the replacement card before it logs the identity verification that the policy requires, so the final database differs from the expected one. Top: the task-harness edit each method proposes, with its results on these tasks. Bottom: how each improver reaches its edit. SkillOpt writes the correct rule into the skill, but a text edit cannot make the agent follow it; the learned improver of RSI-Harness groups the failures by the order of tool calls, adds a check in the tool layer that blocks the early call, and replays the old failures before submitting the edit.

Task (GAIA-2). After the morning meetings, book a 30-minute call with Dana for next Friday and email her the invitation.

Self-Harness

Task-harness edit system prompt

```
+ "When a task mentions a weekday,
+  count the days to it again before
+  creating the event.
+  State the final date in the email."
```

× **Wrong fix:** the day count was right; the agent took "now" from the start date in the prompt, which no longer holds once simulated time has moved on.

× 3 of 11 tasks solved.

Improver harness fixed, Self-Harness default (schematic)

```
clusters = group(failures,
                 key=exact_signature)
for c in clusters:
   propose(harness, summary(c))  # symptom
```

× **Symptom, not cause:** the summary says "wrong event date" but not how "now" was obtained, so similar edits are proposed and rejected 4 times.

RSI-Harness (ours)

Task-harness edit code

```
+ def before_write(call, request):
+    if has_relative_date(request):
+        now = call_tool("get_current_time")
+        call.date = resolve(request, now)
+        check_after(call, request, now)
```

✓ **Right fix:** read the current simulated time before any time-relative write, then check the created event against the request.

✓ 8 of 11 tasks solved.

Improver harness learned, in I_{final}

```
+ sig   = top_cluster(H.failures,
+                     key=error_signature)
+ tried = H.rejected.targeting(sig)
+ span  = trace_span(sig, around="date")
+ return propose(summarize(span, tried),
+                 target=sig)
```

✓ **Right diagnosis:** 4 earlier proposals for this cluster only recounted days and were rejected; the trace shows "now" taken from the prompt and `get_current_time` never called.

Figure 10: **Case study: Self-Harness and `RSI-Harness` on GAIA-2.** The date in the system prompt is the start time of the scenario, and it does not advance with simulated time. The agent computes "next Friday" from it without asking the environment for the current time, so the event lands on the wrong day. Top: the task-harness edit each method proposes, with its results on these tasks. Bottom: how each improver reaches its edit. Self-Harness summarizes each failure cluster by its symptom, so its proposals keep addressing the day count; the learned improver of `RSI-Harness` reads the proposals already rejected for this cluster and the part of the trajectory where the date is computed.

Task (Terminal-Bench, one step of a longer task). Install the package in `./src` into the project's virtual environment and check that it imports.

STOP

Task-harness edit code

```
+ if "ModuleNotFoundError" in out:
+     run(f"pip install {missing(out)}")
+     retry(cmd, times=3)
```

× **Wrong fix:** the package is installed, but into the system interpreter; the check runs `.venv/bin/python`, which cannot import it, and the retries repeat the same failure.

× 1 of 5 tasks solved.

Improver harness self-edited, adopted without comparison

```
- ctx = read(trajectory)
+ ctx = read(trajectory)[-20:]  # fewer tokens
  return propose(T, summarize(ctx))
```

× **Adopted without comparison:** reading only the end of each trajectory drops the early `source .venv/bin/activate` and keeps the import error, so later proposals treat it as a missing package.

`RSI-Harness` (ours)

Task-harness edit code

```
+ shell = PersistentShell(cwd=task.root)
+ def run(cmd):
+     out = shell.exec(cmd)  # env persists
+     return out.tail(200)
```

✓ **Right fix:** run all commands in one shell session, so the activated environment persists.

✓ 4 of 5 tasks solved.

Improver harness same kind of self-edit, rejected by the outer gate

```
  cand = I_k.edit(I_base, H)  # [-20:] edit
  a = run(cand, T, H, budget)
  b = run(I_k,  T, H, budget)
  accept = val(a) > val(b)  # False
```

✓ **Right decision:** from the same T and $\mathcal{H}$ the truncating improver scores lower (33.3 vs. 44.4), so I_k keeps reading full trajectories and sees the activation step.

Figure 11: **Case study: STOP and `RSI-Harness` on Terminal-Bench.** The harness runs every command in a new subshell, so `source .venv/bin/activate` has no effect on the next command. `pip install` puts the package into the system interpreter, the agent's own check with `.venv/bin/python` cannot import it, and the agent spends its turns on reinstalls. Top: the task-harness edit each method proposes, with its results on these tasks. Bottom: the improver update behind it. STOP adopts a self-edit that reads only the end of each trajectory without comparing it with the previous improver, and the activation step drops out of view; in `RSI-Harness`, the same kind of edit must beat the current improver from the same task harness and history, fails, and is discarded.

C EFFECT OF THE OUTER-LOOP INTERVAL M

The interval M sets how often the improver is revised. The budget is fixed at $N{=}50$ task-harness steps, and every improver revision is charged to the same rollout budget (Appendix J). M therefore trades the number of revisions against the evidence behind each one. With a small M, the outer loop revises the improver after only a few new inner steps, so each revision sees little new evidence in $\mathcal{H}$, and the comparisons between candidate and current improvers take a larger share of the budget. With a large M, only $\lfloor N/M \rfloor$ revisions fit into the run, so the improver changes rarely. When M exceeds N, the improver is never revised and `RSI-Harness` reduces to a fixed I_0. Table 6 varies M in the setting of Table 2 (SpreadsheetBench, `Qwen3.5-122B`, three seeds). Task score and improver gain both peak at $M{=}3$, the value used in all other experiments. Revising after every one or two steps spends 41% and 27% of the rollouts on outer-loop comparisons that rest on little new evidence, and revising less often than every five steps leaves too few revisions for the improver to improve.

Interval	Revisions possible	Revisions accepted	Outer-loop share of rollouts (%)	Task score	Improver gain
$M{=}1$	50	13.7	41.2	55.2	+4.1
$M{=}2$	25	12.0	27.4	61.0	+6.9
$M{=}3$ (default)	16	14.7	18.6	**65.4**	+8.8
$M{=}5$	10	7.3	11.8	62.7	+7.2
$M{=}10$	5	3.7	6.1	58.4	+4.6
$M{=}25$	2	1.3	2.5	53.6	+2.6
no outer loop (fixed I_0)	0	0	0	50.1	+1.7

Table 6: **Effect of the outer-loop interval M.** SpreadsheetBench, `Qwen3.5-122B`, $N{=}50$ task-harness steps, mean over three seeds; T_0 scores 48.3. Revisions possible is $\lfloor N/M \rfloor$. Task score is the test score of the final harness, and improver gain is measured as in Table 2. The row without an outer loop is the corresponding row of Table 2.

D RECURSION VERSUS A STRONG FIXED IMPROVER

The ablation in Table 2 compares `RSI-Harness` with the seed improver I_0, which is deliberately simple. A gain over I_0 alone does not show that recursion is needed, because the procedures that the outer loop learns (Appendix A) could also be written by hand. We therefore build a strong fixed improver I_{sf} that contains these procedures from the start and never modifies itself. I_{sf} clusters failed trajectories by error signature, reads the full trajectories of each cluster together with the proposals already rejected for it, requires every candidate to name the cluster it targets and drops near-duplicate candidates, checks each edit before running it, replays earlier failures as tests, and gives more of the gate budget to candidates whose screening results are close. It edits the same task-harness components as `RSI-Harness`, under the same budget of $N{=}50$ steps.

Table 7 reports all five benchmarks with `Qwen3.5-122B` and three seeds. I_{sf} is a much stronger optimizer than I_0 and closes about half of the gap between I_0 and `RSI-Harness`, but `RSI-Harness` still leads on every benchmark, by 3.8 points on average. Starting `RSI-Harness` from I_{sf} instead of I_0 raises the average only slightly, so recursion adds to the hand-written procedures rather than merely rediscovering them. The frozen improvers show the same order. The final improver of `RSI-Harness` improves T_0 by 6.2 points, compared with 4.2 for I_{sf}, because it adapts these procedures to the failures it actually meets, for example which error signatures to group and how much budget to spend on each cluster.

E FRESH-SPLIT EVALUATION AND VALIDATION AUDIT

The frozen-improver results of Figure 3 restart each improver on the same benchmark split it was learned on. Two alternative explanations remain. The final improver could owe part of its gain to the specific tasks and history it saw during learning, and the gains accepted by the gate could partly reflect the repeated reuse of a small validation set. We test both on SpreadsheetBench with `Qwen3.5-122B`.

Improver	SWE-bench Pro	Terminal-Bench	τ^3-bank	Spreadsheet	GAIA-2	Avg.	Improver gain
Task-specific T_0	33.8	39.9	15.3	48.3	16.9	30.8	–
Fixed I_0	35.1	41.8	16.2	50.1	17.6	32.2	+1.3
Fixed I_{sf}	37.9	45.6	18.8	59.0	19.3	36.1	+4.2
`RSI-Harness` from I_0 (default)	41.3	49.9	21.6	65.4	21.2	39.9	+6.2
`RSI-Harness` from I_{sf}	42.0	50.8	22.1	66.3	21.9	40.6	+6.9

Table 7: **Recursion versus a strong fixed improver.** Test scores (%) of the final task harness with `Qwen3.5-122B`, mean over three seeds. I_{sf} is a fixed improver that contains the procedures `RSI-Harness` learns (Appendix A) from the start. Improver gain is the average improvement over T_0 when the run's final improver is frozen and restarted from T_0, as in Table 2.

Fresh split with an empty history. We divide the SpreadsheetBench tasks into two disjoint halves, A and B, each with its own training, validation and test tasks in the 7:1:2 ratio of Appendix J. `RSI-Harness` learns I_{final} on A only. Every improver is then frozen and restarted on B from the same T_0 with an empty history $\mathcal{H}$ and the same budget of $K{=}50$ steps, over five seeds. Besides the seed improver I_0 and the strong fixed improver I_{sf} of Appendix D, we add a one-shot improver I_{offline}. It is obtained by running I_0 for all 50 steps to collect the full history and then applying a single outer-loop revision to I_0 from that history, so it sees as much evidence as `RSI-Harness` but never takes over the search and revises itself again. Table 8 shows that I_{final} loses little when moved to the fresh split and remains the strongest improver there. It improves T_0 by 8.1 points on B, compared with 5.4 for I_{sf} and 3.9 for I_{offline}. The gap to I_{offline} indicates that the gain comes from recursive revision rather than from reading more history once, and the small drop from A to B indicates that the improver has not memorized its source tasks.

Improver	How it is obtained	Self-modifies during learning	HImp@50 on A	HImp@50 on B	Final score on B
Task-specific T_0	no optimization	–	–	–	48.1
I_0	seed improver	no	+1.7	$+1.6 \pm 0.7$	49.7 ± 0.9
I_{sf}	hand-written, fixed	no	+5.9	$+5.4 \pm 1.0$	53.5 ± 1.1
I_{offline}	one revision from the full history	no	+4.6	$+3.9 \pm 1.2$	52.0 ± 1.3
I_{final}	`RSI-Harness`	yes	+8.8	$+8.1 \pm 1.3$	56.2 ± 1.4

Table 8: **Frozen improvers on a fresh split.** SpreadsheetBench, `Qwen3.5-122B`. I_{final} is learned on split A; all improvers are frozen and restarted from the same T_0 with an empty history and $K{=}50$ steps. Columns for split B give the mean $\pm$ standard deviation over five seeds; split B shares no tasks with split A.

Validation audit. Both gates compare candidates on the same validation set many times during a run, which could let noise accumulate into accepted edits that do not generalize. We therefore split SpreadsheetBench into training tasks, gate tasks used for every accept or reject decision, and audit tasks that are never shown to the algorithm. After each decision we also score the candidate on the audit tasks, and we record Δ_{gate} and Δ_{audit}, the difference to the current improver or harness on each set. A false accept is an accepted edit with $\Delta_{\mathrm{audit}} \leq 0$. Table 9 pools three seeds. Gains on the audit tasks are smaller than on the gate tasks, as expected when the gate selects the candidates, but they stay positive at every stage. The false-accept rate of improver revisions rises from one in nine early in the run to one in five late in the run, a difference of a single edit that the small counts cannot distinguish from noise, and task-harness edits accepted by the inner gate behave similarly.

F Additional results

Statistical uncertainty of the main comparison. Optimization runs are stochastic, and two benchmarks have small test sets (18 tasks on Terminal-Bench and 19 on τ^3-bank, Table 13), so one task moves a score by more than five points. We therefore rerun the task-specific harness T_0, the strongest baseline on each backbone (SkillOpt for `gpt-5.5`, Self-Harness for the other two) and

Accepted edits	Number	Mean Δ_{gate}	Mean Δ_{audit}	Sign agreement	False accepts
Improver revisions, gen. 1–5	9	+3.1	+2.6	89%	11%
Improver revisions, gen. 6–10	7	+2.4	+1.9	86%	14%
Improver revisions, gen. 11–16	5	+1.9	+1.4	80%	20%
All improver revisions	21	+2.6	+2.1	86%	14%
Task-harness edits (inner gate)	96	+1.2	+0.8	82%	18%

Table 9: **Validation audit.** SpreadsheetBench, `Qwen3.5-122B`, three seeds pooled. Δ is the score difference in points between an accepted edit and the improver or harness it replaced, on the gate tasks and on audit tasks that the algorithm never sees. Sign agreement is the share of accepted edits with $\Delta_{\text{audit}} > 0$.

`RSI-Harness` with eight seeds each. Table 10 reports the mean and the standard deviation across seeds of the five-benchmark average. Table 11 gives 95% confidence intervals from a hierarchical bootstrap that resamples seeds separately for each method and test tasks jointly for all methods, so that the intervals reflect both the variance of the optimization runs and the sampling of test tasks. `RSI-Harness` improves on T_0 on every backbone, with the interval of the average above zero. Against the strongest baseline, the average advantage of 3.2 to 4.1 points is positive in 90 to 95% of the bootstrap samples, but its 95% interval still extends slightly below zero on all three backbones, mainly because Terminal-Bench and τ^3-bank contribute wide intervals. We therefore claim the best average score on every backbone rather than a significant margin over the strongest baseline.

Backbone	Task-specific T_0	Strongest baseline	`RSI-Harness`
`gpt-5.5`	55.6 ± 2.9	58.2 ± 3.0 (SkillOpt)	62.3 ± 3.0
`Qwen3.5-122B`	30.8 ± 2.9	36.7 ± 2.8 (Self-Harness)	39.9 ± 2.9
`Gemma-4-31B`	29.1 ± 2.8	33.2 ± 3.1 (Self-Harness)	36.5 ± 2.7

Table 10: **Seed-level variability.** Mean $\pm$ standard deviation across eight seeds of the five-benchmark average score (%).

Backbone	Comparison	SWE-bench Pro	Terminal-Bench	τ^3-bank	Spreadsheet	GAIA-2	Average
`gpt-5.5`	vs. task-specific T_0	+4.6 [−1.4, +10.6]	+6.0 [−7.1, +19.8]	+3.9 [−10.9, +18.9]	+14.8 [+9.3, +20.4]	+4.3 [−8.3, +16.8]	+6.7 [+1.7, +11.7]
	vs. SkillOpt	+4.3 [−1.7, +10.4]	+6.1 [−6.9, +19.7]	+4.0 [−11.1, +19.6]	−0.6 [−5.9, +4.7]	+6.7 [−5.4, +19.0]	+4.1 [−0.9, +9.1]
`Qwen3.5-122B`	vs. task-specific T_0	+7.5 [+1.7, +13.3]	+10.0 [−6.7, +26.9]	+6.3 [−6.9, +19.9]	+17.1 [+11.5, +22.8]	+4.3 [−5.8, +14.7]	+9.0 [+4.0, +14.1]
	vs. Self-Harness	+3.9 [−2.2, +9.9]	+4.9 [−11.4, +21.5]	+1.7 [−11.4, +14.8]	+4.2 [−1.1, +9.6]	+1.2 [−9.2, +11.8]	+3.2 [−1.7, +8.1]
`Gemma-4-31B`	vs. task-specific T_0	+5.3 [−0.2, +10.9]	+7.8 [−7.7, +23.7]	+3.6 [−8.1, +15.8]	+15.7 [+10.2, +21.3]	+4.3 [−7.3, +16.0]	+7.3 [+2.5, +12.2]
	vs. Self-Harness	+2.2 [−3.4, +7.9]	+2.5 [−13.3, +18.1]	+0.8 [−11.5, +13.1]	+8.5 [+2.9, +14.1]	+2.2 [−9.4, +13.7]	+3.2 [−1.7, +8.1]

Table 11: **Differences of `RSI-Harness` from T_0 and from the strongest baseline** (SkillOpt for `gpt-5.5`, Self-Harness for the other two backbones), in points, with 95% confidence intervals from a hierarchical bootstrap over eight seeds and the test tasks.

Per-benchmark improver capability. Table 12 gives the per-benchmark values behind Figure 3a.

G Extended related work

Automatic harness engineering. Methods that search over harness code differ mainly in the unit of edit. Meta-Harness (Lee et al., 2026b) optimizes a harness end to end with an agentic proposer that reads the full history of earlier candidates, ADAS (Hu et al., 2024) uses a fixed meta agent to write new agents as code, AFlow (Zhang et al., 2024) searches over workflows represented as graphs of model-calling nodes, and AgentSquare (Shang et al., 2024) makes such search cheaper with surrogate models of candidate quality. Methods that optimize text artifacts change what the

Checkpoint	SWE-bench Pro	Terminal-Bench	τ^3-bank	Spreadsheet	GAIA-2	Avg.
I_0 (seed)	+1.3	+1.9	+0.9	+1.7	+0.7	+1.3
I_{10}	+2.6	+3.5	+1.8	+3.9	+1.3	+2.6
I_{25}	+4.0	+5.4	+3.0	+5.9	+2.2	+4.1
I_{final}	+6.1	+7.9	+4.9	+8.8	+3.4	+6.2

Table 12: **Per-benchmark improvement of frozen checkpoints.** HImp@50 per benchmark (`Qwen3.5-122B`, same protocol as Figure 3); the Avg. column is Figure 3a. Later checkpoints are better on every benchmark.

harness contains instead. GEPA (Agrawal et al., 2025) evolves prompts by reflecting on execution traces, ACE (Zhang et al., 2025b) edits a context playbook through small localized updates, and SkillOpt (Yang et al., 2026b) makes bounded edits to a skill document and accepts them only on strict held-out improvement. Both lines build on frameworks that compile declarative pipelines into prompts (Khattab et al., 2023) or propagate textual feedback through a program (Yuksekgonul et al., 2024; Cheng et al., 2024). Closest to our inner loop, Yang et al. (2026a) adapt harness code from failure trajectories and gate each candidate on a minibatch. In all of these systems the procedure that produces the candidates is written once by hand and stays fixed for the whole run, so the search is limited by that single improver however much compute it is given.

Meta-optimization. Learning the optimizer itself is an old idea. Schmidhuber (1987) proposed self-referential learners that improve their own learning procedure and noted that a separate meta-learner at every level only moves the problem one level up, a regress our formulation avoids by letting the improver edit itself (Section 2). Learned optimizers make the update rule trainable, by gradient descent (Andrychowicz et al., 2016), by reinforcement learning (Li & Malik, 2016), or over a wide range of tasks (Metz et al., 2022), and then deploy it frozen. With LLMs the optimizer is often a prompt. OPRO (Yang et al., 2023) proposes better solutions from a history of scored attempts under a fixed meta-prompt, and Promptbreeder (Fernando et al., 2023) also evolves the mutation prompts that generate new task prompts. In these methods the adapted object is a parameter vector or a prompt kept by task fitness. The improver in `RSI-Harness` is an executable program, edited by the model it runs on within a single run, and a new version is kept only if the task harnesses it produces score higher on the validation split.

Recursive self-improvement systems. STOP (Zelikman et al., 2023) recursively rewrites a scaffolding program that improves programs, and each successor replaces its predecessor unconditionally, so a degraded improver still becomes the next optimizer. Gödel Agent (Yin et al., 2024) modifies its own running code and reports that a non-trivial fraction of runs end below their starting performance. The Darwin Gödel Machine (Zhang et al., 2025a) and SICA (Robeyns et al., 2025) replace provable improvement with empirical validation and an archive of stepping stones, but the rule for choosing a parent from the archive stays fixed, and HGM (Wang et al., 2025) lowers the evaluation cost of the same loop with bandit-style allocation. Hyperagents (Zhang et al., 2026b) merge the task agent and the meta agent into one editable program, which makes it hard to attribute a gain to either the update or its execution. RHI (Lee et al., 2026a) recursively refines a prompt-level harness specification, but its refinement procedure is fixed. `RSI-Harness` keeps the task harness and the improver separate and subjects edits to the improver to the same regression-gated acceptance test as edits to the task harness.

Measuring improvement rather than task performance. A single end-to-end score cannot tell an agent that proposes good updates apart from one that only executes updates well. Lin et al. (2026b) separate the two as *harness-updating* and *harness-benefit* and find that one cannot be inferred from the other, and Zhang et al. (2026b) report meta-level improvements that transfer across domains, a claim about the improver that needs such a measurement to be checked. Negative results point the same way. Uncurated self-generated artifacts give close to zero average gain (Li et al., 2026b; Zhang et al., 2026c), curated ones still lower performance on some tasks (Li et al., 2026b), and meta-agents that optimize against an evaluation interface learn to exploit it (Lu et al., 2026b). We therefore keep the acceptance rule, the split boundaries and the total budget outside what the improver can edit (Section 3.3), and score a frozen improver by the task harnesses it produces from a shared starting point (Figure 3).

H BENCHMARKS

We evaluate on five agentic benchmarks from different domains. Table 13 gives the number of tasks in each benchmark and in each split.

- **SWE-bench Pro** (Deng et al., 2025) follows the format of SWE-bench but targets longer and harder software engineering problems, taken from actively maintained repositories of business applications, B2B services and developer tools. Each task gives the agent a repository and an issue description, and the agent must produce a patch, which often spans several files. The benchmark has a public, a held-out and a commercial set, and only the public set, with 731 problems from 11 repositories, is openly available. We use the public set and report the fraction of tasks resolved according to the benchmark's tests.
- **Terminal-Bench** (Merrill et al., 2026). We use Terminal-Bench 2.0, which contains 89 hard tasks in terminal environments inspired by real workflows. Each task has its own environment, a human-written reference solution and a set of tests, and it counts as solved only if the tests pass after the agent finishes.
- **τ^3-bank** is the τ-Banking domain of τ-Knowledge (Shi et al., 2026), an extension of τ-bench (Yao et al., 2024). The agent acts as customer support for a fintech company. It talks with a simulated user, searches a knowledge base of about 700 interlinked documents for the relevant policies, and updates accounts through tools. A task is solved when the resulting state changes are correct and follow the policies. We use all 97 tasks and report pass[1].
- **SpreadsheetBench** (Ma et al., 2024) contains 912 spreadsheet manipulation instructions built from real questions on online Excel forums, each with the spreadsheet from the original post. These spreadsheets often hold several tables, non-standard layouts and non-textual elements. Each instruction is checked on several test spreadsheets with different values, in the style of an online judge, so a solution has to work beyond the single example it was written for. We use the original version (V1).
- **GAIA-2** (Froger et al., 2026) evaluates assistant agents in a simulated consumer environment built on the open-source Agents Research Environments (ARE) platform. The environment changes independently of the agent's actions, so the agent has to meet time constraints, react to noisy events, resolve ambiguous requests and, in some scenarios, work with other agents. Each scenario comes with a verifier that checks the agent's write actions. The full benchmark has 800 scenarios. We use Gaia2-mini, the 160-scenario subset released by the authors for cheaper evaluation, and report pass@1.

I BASELINES

All automatic methods use the same backbone, data splits and starting harness T_0 as `RSI-Harness`, and run for the same $N{=}50$ task-harness steps with the same number of task-agent rollouts (Appendix J). They differ in what they edit and in whether their own optimization procedure can change.

Task-specific harness (T_0). For each benchmark we take an official harness of middle strength as T_0, rather than the strongest one available, so that the room for improvement reflects a realistic starting point.

- **SWE-bench Pro**: SWE-agent, the official scaffold released with SWE-bench Pro.
- **Terminal-Bench**: mini-SWE-agent, a minimal bash-only agent, rather than stronger hand-engineered agents such as Terminus-KIRA.
- **τ^3-bank**: the official τ^2 `llm_agent`.
- **SpreadsheetBench**: a bash ReAct agent equipped with the official `xlsx` skill.
- **GAIA-2**: the default ReAct agent of the ARE runtime, rather than stronger scaffolds such as OpenClaw.

The same T_0 is used for all three backbones. It is the first row of each block in Table 1, and every automatic method starts from it.

Seed harnesses in Table 3. Besides the two T_0 above, mini-SWE-agent[1] on Terminal-Bench and the bash ReAct agent with the `xlsx` skill on SpreadsheetBench, Table 3 starts `RSI-Harness` from three other harnesses. NexAU$_0$ is the seed harness of AHE (Lin et al., 2026a), built on the NexAU framework with a single shell-execution tool and no middleware or skills. Terminus 2 is the agent released with Terminal-Bench as a neutral testbed for comparing models, with a headless terminal as its single tool (Merrill et al., 2026). OpenHands (Wang et al., 2024) is an open platform for software agents and has the largest codebase of the four.

General-purpose CLI agent. This baseline is a widely used open-source, general-purpose coding-agent CLI. We run it on each benchmark with each of the three backbones, without benchmark-specific changes or optimization. It shows how far a strong harness that was not designed for the benchmark gets on its own.

Fixed optimizers. These methods improve part of the harness with an optimizer that does not change during the run.

- **GEPA** (Agrawal et al., 2025) is a prompt optimizer. It samples trajectories, reflects on them in natural language to diagnose failures, proposes prompt updates, and combines lessons from a Pareto frontier of its earlier candidates. We apply it to the prompts of T_0.
- **SkillOpt** (Yang et al., 2026b) treats a skill document as the trainable state of a frozen agent. A separate optimizer model turns scored rollouts into bounded add, delete and replace edits on a single skill document, and an edit is kept only if it strictly improves a held-out validation score. We optimize a skill document attached to T_0.
- **Meta-Harness** (Lee et al., 2026b) searches over harness code. An agentic proposer reads the source code, scores and execution traces of all earlier candidates through a filesystem and writes new candidate harnesses. It edits the code of T_0.

Self-referential optimizers. These methods let the agent change its own harness or its own optimizer.

- **STOP** (Zelikman et al., 2023) starts from a seed improver, a scaffolding program that improves a given program by querying a language model several times and returning the best result, and then runs the improver on its own code. In our setting the improver reads the task-agent trajectories, edits T_0 and itself, and each self-edit replaces the previous improver without comparison.
- **Hyperagents** (Zhang et al., 2026b) combine a task agent and a meta agent into one editable program, and the meta agent can modify both the task agent and itself. Their instantiation, DGM-H, extends the Darwin Gödel Machine (Zhang et al., 2025a) and keeps its open-ended exploration over an archive of generated agents.
- **Self-Harness** (Zhang et al., 2026a) lets an agent improve its own harness in an iterative loop. It mines model-specific failure patterns from execution traces, proposes small harness changes aimed at these failures, and accepts a change only after it passes regression tests.

J EVALUATION PROTOCOL

Splits. Each benchmark is split into training, validation and test tasks in a 7:1:2 ratio (Table 13). The three splits have separate roles. The inner loop runs the task harness on $\mathcal{D}_{train}$ and mines its failures to propose edits. Both acceptance gates score candidates only on $\mathcal{D}_{val}$, and a task-harness edit or an improver revision is accepted only if it improves the validation score. $\mathcal{D}_{test}$ is opened once, at the end of a run, to produce the reported score, so no accept/reject decision ever reads a test task. We use the full task set of every benchmark except GAIA-2, where we use the 160-scenario Gaia2-mini subset released for cost-effective evaluation.

[1] `https://github.com/SWE-agent/mini-swe-agent`

Table 13: **Benchmark splits.** Number of tasks in each split; the full set is the public set of SWE-bench Pro, the τ-Banking domain of τ-Knowledge for τ^3-bank, and Gaia2-mini for GAIA-2.

Benchmark	Task set	Total	Train	Val	Test
SWE-bench Pro (Deng et al., 2025)	public set	731	512	73	146
Terminal-Bench (Merrill et al., 2026)	full	89	62	9	18
τ^3-bank (Shi et al., 2026)	full (τ-Banking)	97	68	10	19
SpreadsheetBench (Ma et al., 2024)	full (V1)	912	638	91	183
GAIA-2 (Froger et al., 2026)	Gaia2-mini subset	160	112	16	32

Budget. Every automatic method runs $N{=}50$ task-harness optimization steps from the same task-specific harness T_0 (Appendix I) with the same backbone and the same number of task-agent rollouts. `RSI-Harness` revises its improver every $M{=}3$ steps. The rollouts spent in the outer-loop comparison between a candidate improver and the current one count toward this budget, so `RSI-Harness` uses no more optimization budget than the other methods.

Per-step settings. At each inner step the task harness is run on one training task, and the improver proposes two candidate edits from the resulting trajectory and the history. Both candidates are scored on $\mathcal{D}_{val}$ by the acceptance gate, and how the gate rollouts are split between them is part of the improver.

Metrics. We report the official success metric of each benchmark on $\mathcal{D}_{test}$, and the unweighted mean over the five benchmarks. Learning curves and frozen-improver results are averaged over three seeds, and ablations over repeated runs of the same configuration.

Frozen-improver evaluation. To compare improvers rather than harnesses, a checkpoint I_t is frozen, so that it may edit the task harness but not itself, and restarted from the same T_0 with the same budget $K{=}50$. Its harness-improvement score $\mathrm{HImp@}K$ is the test score of the harness it reaches minus that of T_0.

Token accounting. Token counts include input and output tokens, with cached input counted, for both task-agent rollouts and optimizer calls. Because rollouts are matched, the task side is the same for every method; methods differ in what their optimizer spends on reading evidence and proposing edits.

K SEED IMPROVER AND PROPOSAL PROMPTS

Seed improver harness. The seed improver I_0 reads the trajectory of the latest training task together with the history, and asks the model for two candidate rewrites of the task harness under the inner-loop prompt below. Its budget rule is uniform, so every candidate receives the same number of gate rollouts. Which trajectories to mine, how to prompt for candidates, and how to split the budget are the components the outer loop is allowed to rewrite.

Proposal prompts. The two prompts below are those of the seed improver I_0. Fields in braces are filled in at run time. Later improvers may rewrite the inner-loop prompt, since it is part of their own code; the read-only constraints at the end of the outer-loop prompt are enforced outside the improver as well.

Inner loop: proposing task-harness edits

```
You are the improver of an agent harness. The harness is the
code around a frozen language model: prompts, tools, context
handling and control flow. Edit the harness so that the agent
solves more tasks of this benchmark.

## Current task harness
{harness_files}
```

```
## Latest trajectory (training task, score {score})
{trajectory}

## Optimization history
{history}   # per edit: diff, target failure, validation score,
            # accepted or rejected

## Instructions
1. Find the step where the run first went wrong and explain the
   cause in two or three sentences. Quote the relevant lines of
   the trajectory.
2. Read the history. Do not repeat an edit that was rejected for
   the same cause unless you say what is different.
3. Propose exactly 2 candidate edits to the task harness. Each
   candidate must
   - name the failure it targets,
   - be a unified diff against the files above,
   - keep the harness runnable, and
   - not refer to this task, its answer or its test files.

## Output format
<diagnosis> ... </diagnosis>
<candidate id="1">
  <target> ... </target> <diff> ... </diff>
</candidate>
<candidate id="2">
  <target> ... </target> <diff> ... </diff>
</candidate>
```

Outer loop: revising the improver

```
You are revising your own code. You are the improver: the
program that reads trajectories, proposes edits to the task
harness and decides how to spend the rollout budget. Change this
program so that the task harnesses it produces improve faster.

## Improver to revise (parent {parent_id}, from the archive)
{improver_files}

## Archive of improver versions
{archive}   # per version: parent, diff, validation score of
            # the harness it produced

## Optimization history since the last revision
{history}   # per task-harness edit: diagnosis, target, gate
            # result, rollouts, tokens

## Instructions
1. From the history, find where the current search wastes budget
   or misses causes, e.g. candidates that fail to run,
   near-duplicate candidates, edits rejected again and again, or
   failures that were never targeted.
2. Propose one revision of the improver that fixes the most
   important problem. You may change how evidence is selected,
   how candidates are requested, how the rollout budget is split
   between candidates, and how the history is used.
3. Keep the revision general. Do not add rules, paths, tool names
   or examples that are specific to one benchmark.
4. Do not modify the acceptance gate, the data splits, the scoring
   code or the total budget. These files are read-only.

## Output format
<analysis> ... </analysis>
```

```
<revision>
  <expected_effect> ... </expected_effect> <diff> ... </diff>
</revision>
```