SELF-HPO: SELF-EVOLVING CODING AGENTS VIA HARNESS–POLICY OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

While large language model (LLM) agents have shown promising potential in software engineering, their performance is jointly determined by the model weights and the harness that mediates their interaction with the environment. The two are mutually reinforcing: a better harness exposes and compensates for the weaknesses of the current weights, while stronger weights allow the harness to elicit more capable behavior. Yet existing methods optimize one component with the other fixed, so a harness tuned for one checkpoint may no longer suit an updated policy, while policy training remains limited to the trajectories that its fixed harness elicits. In this work, we propose Self-HPO, a recursively self-improving framework in which the harness and the policy co-evolve through alternating optimization. Specifically, in each round, the current policy first analyzes its own failures and proposes harness revisions (H-step); reinforcement learning (RL) then updates the policy under the selected harness (P-step), and hard tasks encountered during training guide the next H-step, where the updated policy serves as the proposer. Experiments show that Self-HPO consistently outperforms harness-only and policy-only optimization, improving Qwen3.5-35B-A3B from 61.0% to 69.5% on SWE-Bench Verified, with gains extending to out-of-domain benchmarks (*e.g.*, 54.2% $\rightarrow$ 59.7% on SWE-Bench Multilingual). Extensive analysis indicates that the gains are retained in both components and are complementary, and that fewer training prompts fail on every rollout during co-evolution than under RL with a fixed harness. Finally, we show that with a stronger external model as the harness proposer, the same framework yields a further performance boost.

1 INTRODUCTION

Large language model (LLM) agents have shown promising potential in solving complex, multi-step tasks, such as resolving real-world software issues (Jimenez et al., 2024; Yang et al., 2024). Such an agent consists of two components: the model weights and the harness—the external scaffold that decides what the model observes, how its actions are executed, what feedback it receives, and when an episode terminates (Yang et al., 2024; Lee et al., 2026). The weights determine the agent's capabilities, while the harness determines how these capabilities are elicited, so the same model can succeed or fail on the same problem depending on its harness.

There have been two main lines of research on improving such agents. Harness optimization holds the weights fixed and searches over prompts, workflows, or agent code (Hu et al., 2024; Lee et al., 2026; Zhang et al., 2026a), whereas policy optimization holds the harness fixed and updates the model from successful trajectories or verifiable rewards (Shao et al., 2024; DeepSeek-AI et al., 2025; Wei et al., 2025a). However, harness-only optimization cannot improve the underlying model, while policy-only training can learn only from the trajectories that its fixed harness elicits. Although recent works begin to optimize both components (Kim et al., 2026; Chen et al., 2026a; Luo et al., 2026), they rely on an external harness proposer or a fixed task set for harness search, leaving self-driven co-evolution for coding agents underexplored.

In light of this, we argue that the harness and the policy are mutually reinforcing and should be improved together. Harness optimization exposes and compensates for the weaknesses of the current weights, while policy optimization internalizes the elicited behaviors into the weights and shifts the failures that the next harness should target. Iterating the two turns the agent's own experience

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.

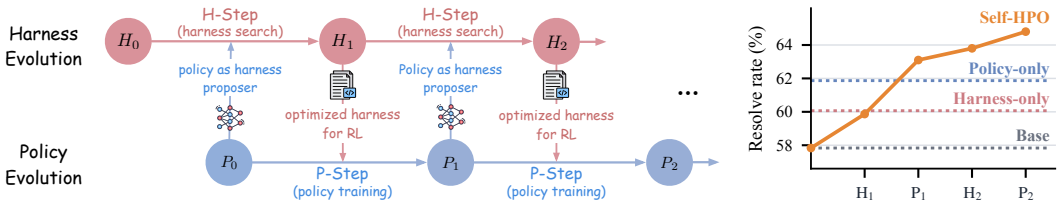


Figure 1: **Self-HPO recursively co-evolves the harness and policy. Left:** In each H-step, the current policy proposes an improved harness; each P-step then updates the policy through reinforcement learning with the optimized harness. **Right:** Average resolve rate over SWE-Bench Verified, SWE-Bench Multilingual, and SWE-Bench Pro along the H- and P-steps; Policy-only and Harness-only denote the final RLVR and Self-Harness baselines.

into a driver of both components, a form of recursive self-improvement (Zelikman et al., 2024; Xia et al., 2026b). In this work, we introduce Self-HPO, an iterative framework that alternates harness optimization (H-step) with policy optimization under the selected harness (P-step), much like the alternating updates of expectation–maximization (Dempster et al., 1977), as illustrated in Figure 1 (left). Specifically, in the H-step, the current policy analyzes its failed trajectories, proposes harness revisions targeting its own failure modes, and verifier scores select the updated harness (§2.2). In the P-step, reinforcement learning updates the policy from verifiable outcome rewards, and hard tasks identified from its training trajectories form the next search set (§2.3). The updated policy then serves as both solver and proposer in the next H-step. We evaluate Self-HPO with Qwen3.5-35B-A3B (Qwen Team, 2026), starting from the minimal ReAct-style coding-agent harness mini-SWE-agent (Yao et al., 2023; Yang et al., 2024). As shown in Figure 1 (right), Self-HPO improves the agent over successive harness and policy updates: on SWE-Bench Verified (Chowdhury et al., 2024), the first H-step improves the base success rate from 61.0% to 64.3%, and subsequent harness and policy updates reach 69.5% over two optimization rounds, with consistent improvements on the out-of-domain SWE-Bench Multilingual (SWE-bench Team, 2025) and SWE-Bench Pro (Deng et al., 2025). Further analysis shows that the gains are retained in both components and are complementary: the final harness H_2 improves the base policy from 61.0% to 64.5%, the final policy P_2 reaches 64.4% under the seed harness, and their combination reaches 69.5%. Moreover, during the second P-step, the fraction of training prompts on which every rollout fails declines steadily, whereas it does not under reinforcement learning with verifiable rewards (RLVR) and a fixed harness. Finally, using Opus 4.8 as a stronger external proposer raises the resolve rate on SWE-Bench Verified further to 71.6%.

The main contributions of this work are as follows:

- We propose Self-HPO, a self-improving framework in which the current policy proposes its own harness revisions, learns through reinforcement learning under the selected harness, and supplies hard tasks for the next round of harness search.
- We show that Self-HPO improves Qwen3.5-35B-A3B from 61.0% to 69.5% on SWE-Bench Verified over two optimization rounds, and that the gains transfer to the out-of-domain SWE-Bench Multilingual and SWE-Bench Pro.
- Extensive analyses show that harness and policy improvements are complementary and retained in each component, and that the framework benefits further from a stronger external proposer.

2 Self-HPO: Self-Evolving via Harness–Policy Optimization

2.1 Problem Setup

Given an issue x, a software-engineering agent combines a policy P, parameterized by an LLM, with a harness H, which controls context, tools, feedback, and termination. The resulting trajectory $\tau \sim \pi_{H,P}(\cdot \mid x)$ is scored by an executable verifier $y(x, \tau) \in \{0, 1\}$. We measure a harness–policy pair by its expected resolve rate on the training data $\mathcal{D}$:

$$J(H, P) = \mathbb{E}_{x \sim \mathcal{D}} \, \mathbb{E}_{\tau \sim \pi_{H,P}(\cdot|x)} \big[y(x, \tau) \big]. \tag{1}$$

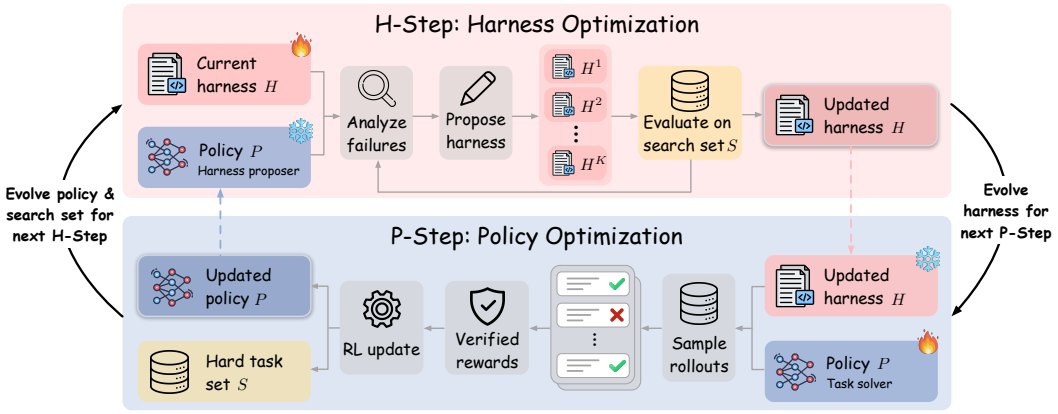


Figure 2: **Detailed overview of Self-HPO.** In the H-step, the current policy analyzes its failures and proposes harness revisions; verifier scores on the search set select the updated harness. In the P-step, the updated harness elicits verified trajectories for reinforcement learning; the resulting policy and hard-task set then condition the next H-step.

The harness and policy are coupled: a harness changes which behaviors the current policy can reliably express and which trajectories become training data, while a policy update changes the failures that the next harness should target. Optimizing either component once against a permanently frozen counterpart ignores this dependence. Self-HPO instead alternates harness and policy updates, rebuilding the data-generating agent around its current state.

Starting from a base policy P_0, a seed harness H_0, and a random initial search set $\mathcal{S}_0 \subset \mathcal{D}$ drawn from the training data $\mathcal{D}$, Self-HPO alternates two updates. At round t, the H-step freezes P_t: the policy analyzes failures of the current pair on $\mathcal{S}_t$, proposes harness revisions, and verifier outcomes select H_{t+1}. The P-step then freezes H_{t+1} and trains P_{t+1} starting from P_t on $\mathcal{D}$; its verified training trajectories also identify the next hard-task set $\mathcal{S}_{t+1} \subset \mathcal{D}$. Thus, only the first search set is sampled at random; later sets reuse P-step outcomes for task selection, while the next H-step generates fresh trajectories on those tasks.

We call this alternating framework Harness–Policy Optimization (HPO). In Self-HPO, the updated target checkpoint also proposes its own harness revisions in the next round, closing the loop without a separately trained proposer. For comparison, HPO (Opus 4.8) uses a fixed external proposer while retaining the same H/P loop. Table 1 reports results for both configurations.

Algorithm 1 summarizes the loop; our default P-step uses online reinforcement learning with Group Relative Policy Optimization (GRPO) (Shao et al., 2024). Here, RL optimizes the objective in Eq. (1) from P_t with H_{t+1} fixed, returning P_{t+1} and verified on-policy training trajectories; SelectHardTasks selects $\mathcal{S}_{t+1}$ from the task IDs represented in those trajectories. Within each H-step, ProposeHarness revises the previous candidate using its trajectories, and Eval returns the resolve rate and trajectories on the fixed search set; the highest-scoring harness is retained separately as H_{t+1}. We stop when the hard-task set stabilizes between rounds, a heuristic criterion without an optimality guarantee; our experiments evaluate two rounds. Like block-coordinate optimization and expectation–maximization (EM) (Dempster et al., 1977), Self-HPO alternates updates while holding the other component fixed. This resemblance is structural only: the two steps optimize different finite-sample criteria rather than a shared likelihood bound, so we claim neither monotonic improvement of J nor convergence. The formulation exposes the controlled factors studied later: whether the search set evolves, how the policy is optimized, and whether either optimized component transfers beyond its training pair.

2.2 H-STEP: FAILURE-CONDITIONED HARNESS SEARCH

Self-proposer for harness optimization. At round t, the H-step freezes P_t and uses the same checkpoint as both task solver and harness proposer, following the upper row of Figure 2. The current pair (H_t, P_t) first generates trajectories on $\mathcal{S}_t$; the proposer inspects the verified failures and produces targeted revisions to the prompt or executable runtime logic (its system prompt is given

Algorithm 1 Self-HPO

Require: Seed harness H_0, base policy P_0, training data $\mathcal{D}$, harness search budget K

$t \leftarrow 0$, randomly initialize harness search set $\mathcal{S}_t \subset \mathcal{D}$
while not converged **do** ▷ Stopping criterion (§2)
  /* H-step: search harness with the current policy as proposer */
  Score v_0, trajectories $\mathcal{T}_0 \leftarrow \mathrm{Eval}(H_t, P_t, \mathcal{S}_t)$, $v_{\mathrm{best}} \leftarrow v_0$, $H_{t+1} \leftarrow H_t$, $H_t^0 \leftarrow H_t$
  for $k = 1, \ldots, K$ **do**
    $H_t^k \leftarrow \mathrm{ProposeHarness}(H_t^{k-1}, P_t, \mathcal{T}_{k-1})$ ▷ Self-proposer (§2.2)
    $v_k, \mathcal{T}_k \leftarrow \mathrm{Eval}(H_t^k, P_t, \mathcal{S}_t)$
    if $v_k > v_{\mathrm{best}}$ **then**
      $H_{t+1} \leftarrow H_t^k$, $v_{\mathrm{best}} \leftarrow v_k$
  /* P-step: train the policy under the updated harness */
  $P_{t+1}, \mathcal{T}_{\mathrm{RL}} \leftarrow \mathrm{RL}(H_{t+1}, P_t, \mathcal{D})$
  $\mathcal{S}_{t+1} \leftarrow \mathrm{SelectHardTasks}(\mathcal{T}_{\mathrm{RL}})$ ▷ Evolving search set (§2.2)
  $t \leftarrow t + 1$
return H_t, P_t

in Appendix D). Each of the K search iterations revises the previous candidate using its evaluation trajectories, while separately retaining the highest-scoring harness seen so far, including the initial H_t. All candidates are evaluated on the same task IDs while the policy, environment, sampling configuration, interaction budget, and verifier remain fixed. The updated harness H_{t+1} is selected by verifier performance alone.

Evolving search set. The search set remains fixed within an H-step but evolves between rounds. We sample $\mathcal{S}_0$ uniformly from $\mathcal{D}$; subsequent search sets consist of hard tasks identified from the verified trajectories generated during the preceding P-step, where a task is considered hard if it is resolved at most once across its rollouts in that P-step. The next H-step regenerates trajectories on those selected task IDs with the updated harness–policy pair, producing fresh failure evidence for proposal while preserving paired comparisons among candidates. Harness search therefore follows the failures exposed by the preceding policy update instead of repeatedly optimizing against the initial tasks.

2.3 P-STEP: POLICY OPTIMIZATION UNDER THE SELECTED HARNESS

The P-step freezes the selected harness H_{t+1} and initializes training from P_t, collecting on-policy trajectories over $\mathcal{D}$ as the policy is updated, as shown in the lower row of Figure 2. An executable verifier assigns binary outcome rewards, and our default optimizer combines GRPO (Shao et al., 2024) with Balanced Adaptive Rollout (BAR) from Orchard (Peng et al., 2026). BAR adaptively draws additional trajectories to construct reward-balanced groups, reducing uninformative zero-variance groups under sparse outcome rewards; GRPO then updates the policy using the resulting group-relative advantages. The loss is applied only to model-generated tokens, while task text, environment observations, and tool outputs are masked.

The trajectories collected during the P-step support both outputs shown in Figure 2: the updated policy P_{t+1} and the next hard-task set $\mathcal{S}_{t+1}$. The next search set is constructed from hard tasks identified in these training trajectories, without a separate post-update evaluation to identify them. The next H-step uses P_{t+1} as both solver and proposer, regenerates failure evidence on those selected task IDs, and searches for the next harness. Appendix A describes the training objective and token masking.

3 EXPERIMENTS

3.1 EXPERIMENTAL SETUP

Harness, model, and data. All experiments use Qwen3.5-35B-A3B (Qwen Team, 2026) as the base policy and the minimal ReAct-style mini-SWE-agent (Yao et al., 2023; Yang et al., 2024) as the seed harness H_0. Following Orchard (Peng et al., 2026), the training data $\mathcal{D}$ (2,484 tasks), used for both harness search and policy optimization, combine SWE-rebench (Badertdinov et al., 2025),

Table 1: **Main results on in-domain (SWE-Bench Verified) and out-of-domain evaluations (SWE-Bench Multilingual and SWE-Bench Pro).** The optimization columns indicate whether each method updates the harness or policy. Scores are resolve rates (%; mean ± standard deviation over three runs). Avg. is the unweighted mean across benchmarks. Bold marks the best result within each setup.

Method	Harness Opt	Policy Opt	SWE-Verified	SWE-Multi	SWE-Pro	Avg.
Base (Qwen3.5-35B-A3B)	✘	✘	61.0 ± 0.2	54.2 ± 0.3	58.3 ± 1.2	57.8 ± 0.4
Setting 1: With a stronger external model (Opus 4.8)						
SFT	✘	✓	65.0 ± 1.1	49.6 ± 0.8	49.1 ± 0.6	54.6 ± 0.5
Meta-Harness (Lee et al., 2026)	✓	✘	69.1 ± 1.9	56.5 ± 0.9	64.8 ± 1.4	63.5 ± 0.8
HPO (ours)	✓	✓	**71.6 ± 0.9**	**61.4 ± 0.7**	**66.0 ± 0.5**	**66.3 ± 0.4**
Setting 2: Self-Improving Setup						
RLVR (Peng et al., 2026)	✘	✓	67.5 ± 0.8	56.3 ± 1.0	61.8 ± 1.3	61.9 ± 0.6
Self-Harness (Zhang et al., 2026a)	✓	✘	65.2 ± 1.5	54.3 ± 0.7	60.7 ± 0.9	60.1 ± 0.6
Self-HPO (ours)	✓	✓	**69.5 ± 0.9**	**59.7 ± 0.8**	**65.2 ± 1.4**	**64.8 ± 0.6**

SWE-rebench V2 (Badertdinov et al., 2026), and Scale-SWE (Zhao et al., 2026). We perform two rounds of alternating harness and policy optimization, with $K = 6$ harness candidates per H-step and 50 RL updates per P-step. Appendices A and B describe the implementation and evaluation details.

Benchmarks and evaluation metrics. SWE-Bench Verified (SWE-Verified) contains 500 human-validated GitHub issues and is our primary benchmark (Jimenez et al., 2024; Chowdhury et al., 2024). We further evaluate out-of-domain performance on SWE-Bench Multilingual (SWE-Multi; 300 tasks) (SWE-bench Team, 2025) and the public V2 set of SWE-Bench Pro (SWE-Pro; 642 tasks) (Deng et al., 2025), covering different programming languages and more challenging repository-level tasks. We report resolve rate, the percentage of tasks whose generated patches pass the benchmark verifier. Main results are averaged over three evaluation runs (avg@3), with run-level standard deviations; Avg. denotes the unweighted mean across the three benchmarks.

Baselines. We compare joint optimization with policy-only and harness-only baselines under two setups, using the unmodified Base agent as a reference. In the *self-improving setup*, RLVR (Peng et al., 2026) updates the policy under the seed harness, whereas Self-Harness (Zhang et al., 2026a) optimizes the harness using the base policy as proposer without updating its weights. Self-HPO alternates both updates and uses the current policy to propose harness changes. In the *external-model setup*, supervised fine-tuning (SFT) learns from resolved Opus 4.8 trajectories on the same training tasks, while Meta-Harness (Lee et al., 2026) uses Opus 4.8 to optimize the harness with the target policy fixed. HPO combines external harness proposals with RL policy updates.

3.2 Main Results

Self-HPO consistently outperforms policy-only and harness-only optimization. As shown in Table 1, jointly optimizing the harness and policy achieves the best performance on all three benchmarks within each setup. Without an external proposer, Self-HPO reaches 69.5% on SWE-Verified, outperforming RLVR and Self-Harness by 2.0 and 4.3 points, respectively. Its average score improves from 57.8% for Base to 64.8%, exceeding the stronger single-component baseline, RLVR, by 2.9 points. The same pattern holds with external supervision: HPO surpasses Meta-Harness by 2.5 points on SWE-Verified and 2.8 points on average. These consistent gains support optimizing the harness and policy together. Appendix B further compares these results with frontier models on SWE-Verified.

The gains of Harness–Policy Optimization extend to out-of-domain benchmarks. Self-HPO achieves 59.7% on SWE-Multi and 65.2% on SWE-Pro, improving over Base by 5.5 and 6.9 points and over RLVR by 3.4 points on both benchmarks. HPO also obtains consistent gains, reaching 61.4% and 66.0%, respectively. The contrast with Self-Harness is particularly pronounced on SWE-Multi, where harness-only optimization improves over Base by only 0.1 points.

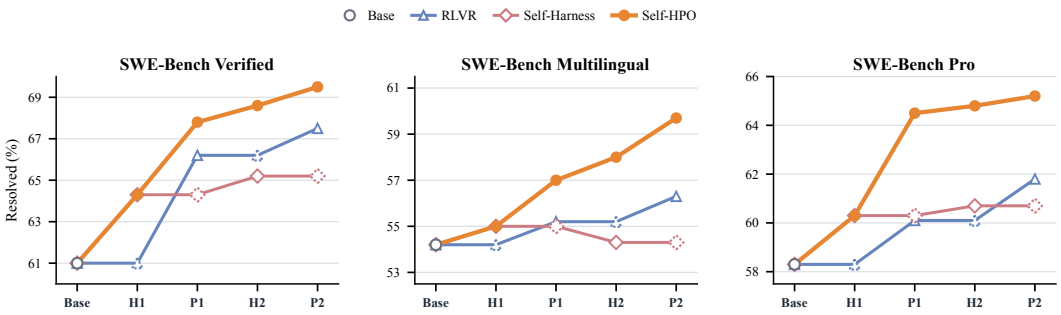


Figure 3: **Optimization trajectories across harness and policy updates.** Resolve rates (%) on SWE-Bench Verified, SWE-Bench Multilingual, and SWE-Bench Pro are shown on the common Base–H_1–P_1–H_2–P_2 schedule. Self-HPO alternates H- and P-steps, whereas Self-Harness and RLVR update only the harness and policy, respectively. Markers with dashed outlines indicate steps at which the corresponding method performs no update.

Harness-based distillation yields stronger transfer than trajectory imitation under our recipes. SFT and Meta-Harness represent two different approaches for transferring knowledge from Opus 4.8: SFT distills generated trajectories into the model weights, whereas Meta-Harness uses external proposals to optimize the harness while keeping the model fixed. Meta-Harness outperforms SFT by 4.1 points on SWE-Verified and 8.9 points on average. The gap is largest out of domain: SFT falls below Base on both SWE-Multi and SWE-Pro, while Meta-Harness improves on both. Under the evaluated recipes, external expertise is thus more effectively transferred through harness revisions than through trajectory imitation alone. HPO further improves on Meta-Harness, indicating that policy optimization remains beneficial when coupled with an optimized harness.

Alternating harness and policy optimization drives continued agent improvement. Figure 3 traces the optimization trajectories of Self-HPO and the two one-sided baselines. Self-HPO and Self-Harness begin with the same first H-step, but their trajectories diverge once Self-HPO updates its policy. Across all three benchmarks, the second H-step improves the updated Self-HPO policy, and the following P-step provides a further gain. In contrast, another harness-only update does not consistently improve performance: Self-Harness drops from 55.0% to 54.3% on SWE-Multi. The final Self-HPO pair surpasses both single-component baselines on every benchmark, supporting repeated adaptation of the harness to the evolving policy. Figure 6 in the appendix provides a candidate-level view of harness search on SWE-Verified.

3.3 Ablations

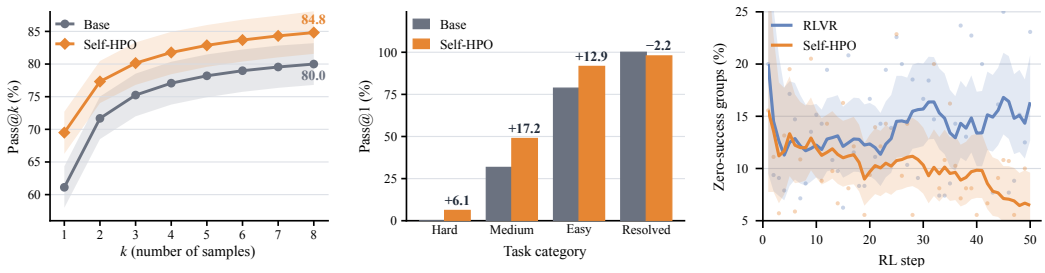
We examine the policy-update rule, search-set construction, and harness proposer on SWE-Verified. Table 2 uses HPO as the reference for ablating the shared H-step–P-step framework; replacing its external proposer with the current policy gives Self-HPO. Appendix B defines each variant.

RL yields stronger policy updates than imitation-based alternatives. We replace RL in the P-step with either SFT on trajectories distilled from Opus 4.8 or rejection-sampling fine-tuning (RFT) on the target policy's own successful trajectories. SFT reduces the final resolve rate from 71.6% to 67.2%, while RFT reaches 70.5%, trailing RL by 1.1 points. Learning from successful self-generated trajectories thus provides a stronger alternative than external trajectory distillation in this comparison, but RL yields the best final agent within the alternating framework.

Table 2: **Ablations on SWE-Bench Verified.** Resolve rates (%) for HPO variants; parentheses give changes from HPO in percentage points.

Configuration	Resolve rate (%)
HPO	**71.6**
w/ SFT in P-step	67.2 (−4.4)
w/ RFT in P-step	70.5 (−1.1)
w/ Fixed search set in H-step	70.1 (−1.5)
w/ Self-proposer in H-step	69.5 (−2.1)

Evolving the search set helps harness optimization keep pace with the policy. Reusing the initial search set lowers performance from 71.6% to 70.1%. Unlike the fixed-set variant, the evolving configuration directs the next H-step toward hard



(a) Pass@k (b) Pass@1 by task difficulty (c) Zero-success groups during RL

Figure 4: **Multi-attempt performance, task-difficulty breakdown, and RL training signal.** (a) and (b) compare Base with the final Self-HPO pair on SWE-Bench Verified using eight runs each. (a) Pass@k with 95% task-bootstrap confidence intervals. (b) Pass@1 by Base task success rate: Hard (0%), Medium ($(0\%, 50\%]$), Easy ($(50\%, 100\%)$), and Resolved (100%); bar annotations give Self-HPO's gains over Base in percentage points. (c) Fraction of zero-success GRPO groups (no rollout of the prompt succeeds), for the last 50 updates of RLVR and the 50 updates of the second Self-HPO P-step, on the training set (Appendix B). Dots show per-update values; curves show the ratio pooled over a trailing 10-update window, with 95% Wilson confidence bands.

tasks identified from the preceding P-step rollouts. The gain supports refreshing the search target as the agent improves: failures relevant to an earlier policy need not remain the most useful targets for its successor.

Self-proposal retains strong performance without an external proposer. Replacing Opus 4.8 with the current target policy yields 69.5%, a 2.1-point decrease from HPO. Although external proposals provide an advantage, Self-HPO still exceeds both self-improving single-component baselines in Table 1. An external model is therefore helpful but not required for effective Harness–Policy Optimization.

3.4 Analysis

Both optimized components retain benefits outside their final pairing. Table 3 cross-evaluates policies and harnesses from the same Self-HPO trajectory (Appendix B). Applying H_2 to the base policy improves resolve rate from 61.0% to 64.5%, while deploying P_2 under the seed harness reaches 64.4%. Thus, neither component's observed benefit is restricted to the final jointly optimized pair. Their combination is stronger still: $H_2 \times P_2$ reaches 69.5%, exceeding $H_2 \times P_0$ by 5.0 points and $H_0 \times P_2$ by 5.1 points. Interestingly, the first updated policy gains only 0.7 points under H_0, compared with 3.5 under H_1, suggesting that the harness influences how much of the policy update is expressed at evaluation time. Overall, the gains are partly retained in each component, but the optimized harness remains important for eliciting the updated policy's performance.

Table 3: **Cross-evaluating Self-HPO policies and harnesses on SWE-Bench Verified.** Rows vary the harness and columns vary the policy, using components from one Self-HPO trajectory. Scores are resolve rates (%; mean ± standard deviation).

	P_0	P_1	P_2
H_0	61.0 ± 0.2	61.7 ± 1.3	64.4 ± 1.1
H_1	64.3 ± 1.1	67.8 ± 1.6	68.1 ± 1.0
H_2	64.5 ± 1.5	68.6 ± 0.5	$\mathbf{69.5 \pm 0.9}$

Self-HPO improves multi-attempt coverage as well as single-attempt success. To examine whether the gains persist beyond pass@1, Figure 4 compares Base and the final Self-HPO pair using eight attempts per task on SWE-Verified (estimation details in Appendix B). Self-HPO remains above Base for every k from 1 to 8, so its advantage is not confined to a single attempt. As shown in Figure 4a, pass@8 increases from 80.0% to 84.8%, a gain of 4.8 points.

Figure 4b shows pass@1 gains of 17.2 and 12.9 points on Medium and Easy tasks, where Base sometimes succeeds but remains inconsistent, and a 6.1-point gain on Hard tasks. At the same time, its pass@1 decreases by 2.2 points on the Resolved group, showing that improvement is not uniform

Seed harness

```
class ProgressTrackingAgent(DefaultAgent):  # seed agent
  # default loop: query the model, then run its command
  def step(self):
    return self.execute_actions(self.query())

  # stop the episode once the step budget is used up
  def query(self):
    if 0 < self.config.step_limit <= self.n_calls:
      raise LimitsExceeded({..., "submission": ""})
      ✗ (1) budget runs out → empty result
    ...

class AzureModalEnvironment:  # sandbox
  # after each command: did the model submit?
  def _check_finished(self, output):
    if lines[0] == "COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT":
      submission = "".join(lines[1:])
      raise Submitted({..., "submission": submission})
      ✗ accepted as-is, even if the patch is:
      ✗ (2) empty (new file not staged)
      ✗ (3) built on a guessed API / contract
      ✗ (4) malformed and does not apply
```

Optimized harness

```
class AgentHarness(ProgressTrackingAgent):  # inherits seed
  # same loop; overrides only step(), adds checks
  def step(self):
    # budget valve: checked before every step
    if near_step_limit or near_time_limit:
      self.add_messages(rescue_text)
      ✓ (1) warn early → model submits a best-effort patch
    try:
      return super().step()  # = execute_actions(query())
    except Submitted as e:   # raised by _check_finished
      # empty-patch guard (fires once)
      if is_empty(sub) and not self._empty_guarded:
        self.add_messages(empty_guard_text)
        return []
        ✓ (2) reject empty patch; ask to git add -N new files
      # contract grounding on the first non-empty submit
      if not self._grounded:
        self.add_messages(ground_text)
        return []
        ✓ (3) hold once; check names / signatures vs. repo tests
      # final submit: patch fallback
      self._apply_fallback(e)
      raise
      ✓ (4) apply check fails → submit the sandbox's git diff
```

Figure 5: **Seed versus optimized harness (HPO, second H-step).** Left: the default agent loop, with the four failure modes it lets through (✗). Right: the optimized harness subclasses the seed agent and overrides only `step()`; each added check (✓) addresses the matching failure mode. Shared functions and classes use the same color on both sides; Figure 8 gives the full change.

across tasks. Together, the higher pass@1 on inconsistently solved tasks and the overall pass@8 gain indicate improvements in both execution reliability and task coverage within the evaluated sampling budget. During the second Self-HPO P-step (Figure 4c; Appendix B), the fraction of zero-success groups, in which no rollout succeeds and GRPO receives no gradient, declines to 6.5% over the final ten updates, whereas it remains mostly between 11% and 17% over the last 50 updates of RLVR. This is consistent with the co-evolved harness keeping more training prompts within reach of the policy, so that more groups provide a learning signal.

3.5 Case Studies

Harness evolution adds targeted controls around the existing agent loop. Figure 5 contrasts the seed harness with the harness obtained after HPO's second H-step. The optimized harness retains the underlying model-query and command-execution loop, but intervenes before budget exhaustion and at submission. First, it warns the model as the step or time limit approaches and prompts a best-effort submission before the episode ends with an empty result. Second, a one-time empty-patch guard asks the model to include newly created files via `git add -N`. Third, it delays the first non-empty submission to prompt a check of API names and signatures against the repository and tests. Finally, a fallback uses the sandbox's Git diff when the submitted patch fails the apply check. These changes target distinct ways in which useful repository edits can fail to become a valid submission. They combine explicit runtime checks, such as detecting an empty patch, with behavioral guidance, such as requesting an interface check. Harness optimization can therefore change execution and termination logic, in addition to prompts. Appendix D gives the full change as a diff against the seed harness together with the injected messages, and Appendix E walks through a task on which an evolved harness changes the outcome for the same policy.

4 Related Work

Harness optimization for LLM agents. Harnesses govern an agent's interaction with tools and environments (Yao et al., 2023; Yang et al., 2024). Optimization spans prompts and programs (Pryzant et al., 2023; Khattab et al., 2023; Fernando et al., 2023; Yuksekgonul et al., 2025; Agrawal et al., 2026), contexts (Zhang et al., 2026e; Ye et al., 2026a; Sun et al., 2026), and workflows (Hu et al., 2024; Zhang et al., 2024; Wang et al., 2025a; Gao et al., 2025). Agents reuse feedback through refinement and reflection (Madaan et al., 2023; Shinn et al., 2023), skills (Wang et al., 2023), and memory (Zhao et al., 2023; Wang et al., 2024); program evolution also improves

solutions and agents (Novikov et al., 2025; Lange et al., 2025; Zelikman et al., 2024; Zhang et al., 2025; 2026b). Recent systems revise harnesses from execution evidence (Lee et al., 2026; Lin et al., 2026a; Zhang et al., 2026a; Park et al., 2026), shared experience (Weng et al., 2026), demonstrations (Che et al., 2026; Jiang et al., 2026), or self-preferences (Pan et al., 2026). Other work studies continual adaptation (Liu et al., 2026), task selection (Miyai et al., 2026), generalization through search controls and modular edits (Zhang et al., 2026c; Xia et al., 2026b; Wu et al., 2026), and fixed-budget evaluation (Ursekar et al., 2026).

Policy learning and self-improvement. Iterative training uses filtered reasoning (Zelikman et al., 2022; Singh et al., 2024; Yuan et al., 2023), refreshed preferences (Chen et al., 2024; Yuan et al., 2024; Pang et al., 2024), self-play curricula (Zhao et al., 2025; Huang et al., 2025; 2026a), negative teachers (Pei et al., 2026), or evolving environments (Huang et al., 2026b). Software engineering agents learn from executable environments (Pan et al., 2024; Jain et al., 2025; Yang et al., 2025), software-evolution supervision (Wei et al., 2025a), and self-play code repair (Wei et al., 2025b; Choi et al., 2026). Policy improvement also draws on verifiable-reward RL (Shao et al., 2024; DeepSeek-AI et al., 2025; Peng et al., 2026), harness-guided distillation (Ye et al., 2026b), and test-time learning during solution search (Wang et al., 2025b; Yuksekgonul et al., 2026).

Joint harness–policy evolution. Joint adaptation spans prompts and weights (Soylu et al., 2024; Tiwari et al., 2026; Zhang et al., 2026d; Lu et al., 2026), executable harnesses and policies (Hebbar et al., 2026; Karten et al., 2026; Chen et al., 2026c; Kim et al., 2026; Luo et al., 2026; Chen et al., 2026b;a), and skills (Xia et al., 2026a; Fu et al., 2026; He et al., 2026), including safety-oriented co-evolution (Mao et al., 2026). Related work traces interventions (Fan & Huang, 2026), studies transfer and multi-harness training (Le et al., 2026; Wei et al., 2026), and distinguishes harness-updating from harness-benefit capabilities (Lin et al., 2026b). Unlike HASE (Luo et al., 2026), which places harness edits in the RL action space, Self-HPO alternates self-proposed harness search with RL and derives the next search set from RL trajectories (Table 5).

5 Limitations

Our experiments use one backbone and focus on repository-level software engineering. Although the gains extend to two out-of-domain benchmarks, transfer to other model families and agent domains remains to be evaluated. The component cross-evaluations establish reuse within one harness–policy lineage; portability to independently developed agent frameworks remains untested. We study two optimization rounds from a single training run; longer-run stability, variance across training seeds, and the trade-off between additional search, training cost, and performance remain open. In particular, harness search adds candidate-evaluation rollouts on top of RL, which the RLVR baseline does not receive. The proposer and distillation comparisons evaluate complete optimization recipes, not isolated proposal quality or a universal ranking of knowledge-transfer mechanisms under matched compute. Finally, the harness case illustrates mechanisms rather than attributing gains to individual code changes, and passing benchmark tests does not guarantee patch correctness beyond the test suite.

6 Conclusion

We introduced Self-HPO, a self-improving agent framework based on alternating Harness–Policy Optimization. The current policy proposes harness revisions from execution failures, learns through RL under the selected harness, and supplies hard tasks that guide the next H-step. Across three software-engineering benchmarks, Self-HPO outperforms policy-only and harness-only optimization, improving average resolve rate by 7.0 points over Base and 2.9 points over RLVR. Ablations favor RL updates and an evolving search set, while cross-evaluation shows that both optimized components retain benefits when paired with earlier counterparts. Together, these results support improving the policy and the harness as an interacting system, with each update shaping the next stage of agent improvement.

AI USE STATEMENT

We used generative AI tools to aid and polish the writing of this paper. Large language models are also components of the studied systems (Section 3.1). We did not use generative AI tools to formulate the research questions or to propose the method. We reviewed all AI-assisted content and take full responsibility for this work.

REPRODUCIBILITY STATEMENT

To support the reproducibility of our results, we provide:

- Implementation details of harness search and policy optimization, including training data and hyperparameters, in Appendix A;
- Evaluation protocols, ablation definitions, and analysis procedures in Appendix B;
- The harness proposer system prompt, the optimized harness as a full diff against the seed harness, and its injected messages in Appendix D;
- A complete example of how an evolved harness changes a trajectory in Appendix E.

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. In *International Conference on Learning Representations*, 2026.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint arXiv:2505.20411*, 2025.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. SWE-rebench V2: Language-agnostic SWE task collection at scale. *arXiv preprint arXiv:2602.23866*, 2026.

Lirong Che, Yuzhe Yang, Peiwen Lin, Chuang Wang, Xueqian Wang, and Jian Su. DemoEvolve: Overcoming sparse feedback in agentic harness evolution with demonstrations. *arXiv preprint arXiv:2605.24539*, 2026.

Mark Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Mingju Chen, Can Lv, Guibin Zhang, Heng Chang, and Shiji Zhou. HarnessForge: Joint harness and policy evolution for adaptive agent systems. *arXiv preprint arXiv:2606.01779*, 2026a.

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

Zhengyu Chen, Teng Xiao, Huaisheng Zhu, Yige Yuan, Luan Zhang, and Jingang Wang. Co-Harness: Co-evolving harnesses and model weights for LLM agents. *arXiv preprint arXiv:2607.22688*, 2026c.

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*, 2024.

Caroline Choi, Zeyneb Kaya, Shirley Wu, Tengyu Ma, Tatsunori Hashimoto, and Ludwig Schmidt. Anchored self-play for code repair. *arXiv preprint arXiv:2607.03523*, 2026.

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

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Arthur P. Dempster, Nan M. Laird, and Donald B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

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? *arXiv preprint arXiv:2509.16941*, 2025.

Tianyu Fan and Chao Huang. HELIX: Model-harness co-evolution for recursive self-improvement. *arXiv preprint arXiv:2608.13951*, 2026.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

Zenghuang Fu, Zhaoyang Li, Qiuyuan Ai, Haoyu Wu, Minghui Wu, Chenxu Zhao, Ante Wang, Guannan He, and Changwei Wang. Self-play meets skill evolution: Self-evolving search agents that pose, solve, and remember. *arXiv preprint arXiv:2607.29468*, 2026.

Hongcheng Gao, Yue Liu, Yufei He, Longxu Dou, Chao Du, Zhijie Deng, Bryan Hooi, Min Lin, and Tianyu Pang. FlowReasoner: Reinforcing query-level meta-agents. *arXiv preprint arXiv:2504.15257*, 2025.

Zelin He, Haotian Lin, Boran Han, Wei Zhu, Haoyang Fang, Bernie Wang, Xuan Zhu, Runze Li, and Matthew Reimherr. ReSkill: Reconciling skill creation with policy optimization in agentic RL. *arXiv preprint arXiv:2606.01619*, 2026.

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

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-Zero: Self-evolving reasoning LLM from zero data. *arXiv preprint arXiv:2508.05004*, 2025.

Chengsong Huang, Haolin Liu, Tong Zheng, Runpeng Dai, Langlin Huang, Jinyuan Li, Zongxia Li, Zhepei Wei, Yu Meng, and Jiaxin Huang. G-Zero: Self-play for open-ended generation from zero data. *arXiv preprint arXiv:2605.09959*, 2026a.

Chengsong Huang, Zifeng Wang, Rujun Han, Jun Yan, Yanfei Chen, Zoey CuiZhu, Ke Jiang, Peng Xia, Han Yu, Yufan Zhuang, Yifei Ming, Jiaqi Pan, Bhavana Dalvi Mishra, Jiaxin Huang, Burak Gokturk, Tomas Pfister, and Chen-Yu Lee. EnvHarness: Awakening static worlds for agent learning. *arXiv preprint arXiv:2608.19880*, 2026b.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. *arXiv preprint arXiv:2504.07164*, 2025.

Wen Jiang, Mingmin Chu, Yimeng Tian, Qianxin Zhang, Haofei Yang, Rui Yang, Yang Liu, Tao Lv, and Fangming Li. HarnessEvolve: Learning from reference trajectories for reliable agent self-evolution. *arXiv preprint arXiv:2609.00829*, 2026.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world Github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

Seth Karten, Joel Zhang, Tersoo Upaa, Jr, Ruirong Feng, Wenzhe Li, Chengshuai Shi, Chi Jin, and Kiran Vodrahalli. Continual Harness: Online adaptation for self-improving foundation agents. *arXiv preprint arXiv:2605.09998*, 2026.

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. *arXiv preprint arXiv:2310.03714*, 2023.

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

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

Chenqian Le, Jiayi Cheng, Qijia He, Runhao Li, Yinghao Li, and Xupeng Chen. What does multi-harness RL learn? Credit assignment and portability in coding agents. *arXiv preprint arXiv:2609.04518*, 2026.

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

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

Minhua Lin, 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. *arXiv preprint arXiv:2605.30621*, 2026b.

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

Xinyu Lu, Kaiqi Zhang, Jinglin Yang, Boxi Cao, Yaojie Lu, Hongyu Lin, Min He, Xianpei Han, and Le Sun. P^2O: Joint policy and prompt optimization. *arXiv preprint arXiv:2603.21877*, 2026.

Haochen Luo, Yi Huang, Sichun Luo, Fengyuan Liu, Lei Li, Zefa Hu, Junlan Feng, and Qi Liu. Harness-aware self-evolving: Co-evolving model weights, harness, and task solutions. *arXiv preprint arXiv:2607.03935*, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

Qinghua Mao, Wanying Qu, Dadi Guo, Leitao Yuan, Qingyu Liu, Yu Li, Guanxu Chen, Yanwei Fu, Xi Lin, Xia Hu, and Dongrui Liu. SafeEvolve: Harness-policy co-evolution from agent experience for safety alignment. *arXiv preprint arXiv:2609.02786*, 2026.

Atsuyuki Miyai, Kiyoharu Aizawa, and Toshihiko Yamasaki. Task-CoEvolve: Efficient harness optimization via adaptive validation task selection. *arXiv preprint arXiv:2608.20169*, 2026.

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. *arXiv preprint arXiv:2506.13131*, 2025.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. *arXiv preprint arXiv:2412.21139*, 2024.

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

Richard Yuanzhe Pang, Weizhe Yuan, Kyunghyun Cho, He He, Sainbayar Sukhbaatar, and Jason Weston. Iterative reasoning preference optimization. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Sungho Park, Wonjoong Kim, Rongyuan Tan, Jue Zhang, Wook-Shin Han, Pengfei Gao, Chanyoung Park, Yongqiang Yao, Rao Fu, Elsie Nallipogu, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. AutoSaddler: Automatic harness optimization with durable updates from agent execution traces. *arXiv preprint arXiv:2608.23041*, 2026.

Rongcan Pei, Zhepei Wei, Shuyao Xu, Xinyu Zhu, Wei-Lin Chen, and Yu Meng. Negative self-distillation: Learning to reason by avoiding flaws. *arXiv preprint arXiv:2609.11699*, 2026.

Baolin Peng, Wenlin Yao, Qianhui Wu, Hao Cheng, Xiao Yu, Rui Yang, Tao Ge, Alessandro Sordoni, Xingdi Yuan, Yelong Shen, Pengcheng He, Tong Zhang, Zhou Yu, and Jianfeng Gao. Orchard: An open-source agentic modeling framework. *arXiv preprint arXiv:2605.15040*, 2026.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7957–7968, 2023.

Qwen Team. Qwen3.5-35B-A3B model card, 2026. URL https://huggingface.co/Qwen/Qwen3.5-35B-A3B.

Zhihong Shao, Peiyi Wang, Qihao Zhu, et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.

Avi Singh, John D. Co-Reyes, Rishabh Agarwal, et al. Beyond human data: Scaling self-training for problem-solving with language models. *Transactions on Machine Learning Research*, 2024.

Dilara Soylu, Christopher Potts, and Omar Khattab. Fine-tuning and prompt optimization: Two great steps that work better together. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024.

Qi Sun, Stefan Nielsen, Rio Yokota, and Yujin Tang. Evolutionary context search for automated skill acquisition. *arXiv preprint arXiv:2602.16113*, 2026.

SWE-bench Team. SWE-bench Multilingual. Hugging Face dataset, 2025. URL https://huggingface.co/datasets/SWE-bench/SWE-bench_Multilingual.

SWE-bench Team. SWE-bench leaderboards. https://www.swebench.com, 2026. Verified leaderboard, mini-SWE-agent entries; accessed 2026-09-26.

Rishabh Tiwari, Kusha Sareen, Lakshya A Agrawal, Joseph E. Gonzalez, Matei Zaharia, Kurt Keutzer, Inderjit S Dhillon, Rishabh Agarwal, and Devvrit Khatri. Learning, fast and slow: Towards LLMs that adapt continually. *arXiv preprint arXiv:2605.12484*, 2026.

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

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Yinjie Wang, Ling Yang, Guohao Li, Mengdi Wang, and Bryon Aragam. ScoreFlow: Mastering LLM agent workflows via score-based preference optimization. *arXiv preprint arXiv:2502.04306*, 2025a.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, Hao Cheng, Pengcheng He, Weizhu Chen, Shuohang Wang, Simon Shaolei Du, and Yelong Shen. ThetaEvolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*, 2025b.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Hongliang Wei, Xiaobing Tu, Yinggui Wang, Zhengxi Liu, Rongkun Xue, Jinkui Ren, Xiantao Zhang, Debin Zhao, and Xiaopeng Fan. HarnessBandit: Joint learnability-transferability scheduling for multi-harness agentic reinforcement learning. *arXiv preprint arXiv:2609.13739*, 2026.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025a.

Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida Wang. Toward training superintelligent software agents through self-play SWE-RL. *arXiv preprint arXiv:2512.18552*, 2025b.

Zhaotian Weng, Antonis Antoniades, Deepak Nathani, Zhen Zhang, Xiao Pu, and Xin Eric Wang. Group-evolving agents: Open-ended self-improvement via experience sharing. *arXiv preprint arXiv:2602.04837*, 2026.

Siwei Wu, Jincheng Ren, Yizhi Li, Haau-Sing Li, Chengran Yang, Yuxuan Zhang, Weicheng Gu, Jian Yang, Riza Batista-Navarro, Chuanyi Zhang, Xianglong Liu, Ming Zhou, Bryan Dai, and Chenghua Lin. ModularRSI: Modular and generalizable recursive harness self-improvement. *arXiv preprint arXiv:2609.14857*, 2026.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. SkillRL: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*, 2026a.

Peng Xia, Rujun Han, Zifeng Wang, Yanfei Chen, Yufan Zhuang, Yoonho Lee, Chengsong Huang, Han Yu, Zhongying CuiZhu, Yifei Ming, Huaxiu Yao, Burak Gokturk, Tomas Pfister, and Chen-Yu Lee. RRSI: Regularized recursive self-improvement of agent harnesses. *arXiv preprint arXiv:2609.24972*, 2026b.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025.

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

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

Haoran Ye, Yuxing Lu, Haonan Dong, Zhaochen Su, and Guojie Song. Harness-Zero: Harness distillation via agent-as-harness. *arXiv preprint arXiv:2609.24974*, 2026b.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models. In *International Conference on Machine Learning*, 2024.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. Scaling relationship on learning mathematical reasoning with large language models. *arXiv preprint arXiv:2308.01825*, 2023.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639:609–616, 2025.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping reasoning with reasoning. *arXiv preprint arXiv:2203.14465*, 2022.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *Conference on Language Modeling*, 2024.

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

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Godel Machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025.

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

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. *arXiv preprint arXiv:2410.10762*, 2024.

Luan Zhang, Ruochen Zhou, Dandan Song, Zhengyu Chen, Yuhang Tian, Jun Yang, Huipeng Ma, Chenhao Li, Guangyuan Feng, Xudong Li, Yizhou Jin, and Yan Xu. HarnessCompass: Guiding automatic harness evolution toward generalizable and effective agent harnesses. *arXiv preprint arXiv:2608.01918*, 2026c.

Lunjun Zhang, Ryan Chen, and Bradly C. Stadie. Evolutionary system prompt learning for reinforcement learning in LLMs. *arXiv preprint arXiv:2602.14697*, 2026d.

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

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. *arXiv preprint arXiv:2308.10144*, 2023.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. *arXiv preprint arXiv:2505.03335*, 2025.

Jiale Zhao, Guoxin Chen, Fanzhe Meng, Minghao Li, Jie Chen, Hui Xu, Yongshuai Sun, Wayne Xin Zhao, Ruihua Song, Yuan Zhang, Peng Wang, Cheng Chen, Jirong Wen, and Kai Jia. Immersion in the GitHub universe: Scaling coding agents to mastery. *arXiv preprint arXiv:2602.09892*, 2026.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, 2024.

A Implementation Details

Agent implementation. We use Qwen3.5-35B-A3B (Qwen Team, 2026) as the backbone and SGLang (Zheng et al., 2024) for model serving; RL and SFT use the slime framework on one node with eight B200 GPUs. The seed harness follows the mini-SWE-agent ReAct interface (Yao et al., 2023; Yang et al., 2024), through which the model executes bash commands in a task-specific repository environment. Harness optimization modifies the prompts or runtime logic surrounding the model, while policy optimization updates its parameters under the selected harness.

Optimization data. The initial search set $\mathcal{S}_0$ consists of 300 tasks randomly sampled from the training data $\mathcal{D}$. For subsequent H-steps, the search set consists of hard tasks from $\mathcal{D}$, identified using the verified trajectories collected during the preceding P-step: a task is hard if it is resolved at most once across its rollouts in that P-step. These task IDs are reused to generate fresh failure evidence with the updated policy at the next H-step. The search set remains fixed while candidates within one H-step are compared. This construction reuses model-training experience to guide harness search without requiring a separate rollout stage to identify hard tasks.

Harness optimization. Self-HPO uses the current policy checkpoint as both task solver and harness proposer. The proposer examines failed trajectories and suggests revisions to the current harness; candidate harnesses are evaluated on the round-specific search set with the target policy frozen. HPO instead uses Opus 4.8 to propose revisions, while retaining the same alternating H-step–P-step structure. In both configurations, each H-step evaluates $K = 6$ candidates, each run once per search task, and the selected harness is held fixed during the next policy update. The Self-Harness and Meta-Harness baselines use the same search budget.

Policy optimization. Each P-step initializes from the policy produced by the preceding round and collects on-policy trajectories under the updated harness. The executable verifier supplies outcome rewards, and the policy is optimized with GRPO and Balanced Adaptive Rollout (BAR) (Shao et al., 2024; Peng et al., 2026). Let $\mathcal{A}(\tau_i)$ denote the assistant-generated token positions in trajectory τ_i, $\widehat{A}_i$ its group-relative advantage, and $\rho_{i,u}(\theta)$ the ratio of updated-policy to rollout-policy token probabilities. The clipped objective is

$$\mathcal{L}_P(\theta) = -\mathbb{E}_i \left[\frac{1}{|\mathcal{A}(\tau_i)|} \sum_{u \in \mathcal{A}(\tau_i)} \min \left(\rho_{i,u}(\theta)\widehat{A}_i, \mathrm{clip}(\rho_{i,u}(\theta), 1-\varepsilon_{\mathrm{low}}, 1+\varepsilon_{\mathrm{high}})\widehat{A}_i \right) \right], \quad (2)$$

where $\varepsilon_{\mathrm{low}} = 0.2$ and $\varepsilon_{\mathrm{high}} = 0.28$ are the clipping thresholds. The loss is applied only to model-generated tokens; task descriptions and environment observations are masked. Each P-step performs 50 updates. Each update samples 32 training tasks, and BAR starts with 4 rollouts per task and draws up to 8 to form groups whose fraction of successful rollouts lies between 0.25 and 0.75; the global batch size is 128. We use Adam with a constant learning rate of 10^{-6}, $(\beta_1, \beta_2) = (0.9, 0.98)$, weight decay 0.01, an entropy coefficient of 10^{-4}, and no KL penalty. Rollouts use temperature 0.95 and top-p 0.95. RLVR uses the same configuration under the seed harness, with 50 updates per step.

B Additional Evaluation Details

Resolve rate and avg@3. For a benchmark containing N tasks, a run's resolve rate is $100N^{-1}\sum_{i=1}^{N} y_i$, where y_i indicates whether the submitted patch passes the verifier. Table 1 reports the mean resolve rate over three runs and their standard deviation. All evaluations use a step limit of 250 and temperature and top-p of 0.95; SWE-Pro uses the 642-task public V2 set. Runs that fail because of sandbox or API errors are rerun, and resolve rates are computed over all tasks in each benchmark.

Ablation variants. The P-step variants in Table 2 replace RL with SFT or RFT within the alternating optimization loop. SFT trains on trajectories distilled from Opus 4.8. RFT instead samples trajectories from the target policy itself, retains those that pass the verifier, and fine-tunes on the accepted trajectories. The fixed-search-set variant retains the initial search tasks instead of selecting new hard tasks from P-step experience. The self-proposer variant uses the current target policy in

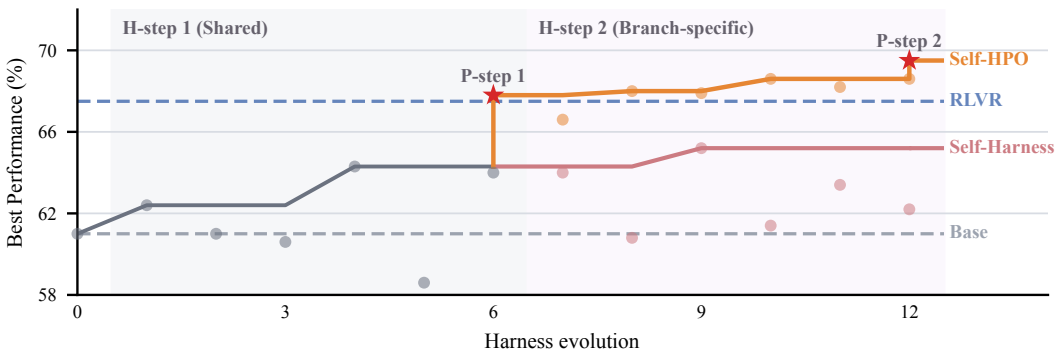


Figure 6: **Best-so-far performance during harness evolution on SWE-Bench Verified.** Circles show individual candidate scores; solid curves track the running best. Candidates 1–6 are shared, while 7–12 belong to the separate Self-Harness and Self-HPO branches. Vertical increases and stars mark Self-HPO's two P-steps. Dashed horizontal lines show Base and final RLVR performance.

place of Opus 4.8. The SFT baseline in Table 1 fine-tunes the base policy on Opus 4.8 trajectories collected on the RL training tasks, keeping only resolved trajectories; it trains for two epochs with batch size 64 and a constant learning rate of 5×10^{-6} after a 10% warmup. The SFT row in Table 1 is a policy-only baseline, whereas SFT in Table 2 is a replacement for the P-step within HPO; the two rows therefore describe different agent configurations.

Search-frontier curves. Figure 6 shows individual harness-candidate scores and their running best on SWE-Verified. The first six candidates are shared; subsequent candidates belong to the separate Self-Harness and Self-HPO branches. Self-HPO's P-steps do not add a harness candidate and therefore appear as vertical updates at the same horizontal coordinate. The RLVR line marks its final score rather than its training trajectory. This view complements Figure 3, which reports the agent pairs at the selected H-step and P-step endpoints.

After the shared first H-step reaches 64.3%, continued harness-only search improves the best score to 65.2%. Self-HPO instead updates the policy to 67.8% before resuming search; its second H-step then reaches 68.6%, followed by 69.5% after the second policy update. The separation between the two branches shows the benefit of policy updates alongside continued harness search.

Pass@k estimation. Figure 4 compares Base and the final Self-HPO pair with eight attempts per task. Following Chen et al. (2021), for a task with c successes among $n = 8$ attempts, we estimate

$$\text{pass@}\, k = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}, \qquad k = 1, \ldots, 8,$$

and average over tasks. Pass@1 averages the success frequency, whereas pass@8 records whether any of the eight attempts succeeds. The shaded regions show 95% bootstrap confidence intervals over tasks.

Task-difficulty groups. Tasks are grouped by Base success rate p: Hard ($p = 0$), Medium ($0 < p \leq 0.5$), Easy ($0.5 < p < 1$), and Resolved ($p = 1$). Figure 4b reports pass@1 for each group.

Zero-success groups during RL. Figure 4c uses the SWE-rebench V1 and V2 tasks in the training mixture and excludes any rollout group containing a `TimeExceeded` outcome, applying the same filter to both methods. Each group contains all rollouts that BAR draws for one prompt, counted before GRPO filters out zero-variance groups; a zero-success group contains no successful rollout, and its frequency is computed over all retained groups, including all-success groups. We compare the last 50 RLVR updates with the 50 updates of Self-HPO's second P-step. Dots show per-update ratios; each curve pools zero-success and total group counts over the trailing ten updates, with a shorter window at the start. The shaded regions are 95% Wilson intervals for these pooled proportions.

Table 4: **Comparison with frontier models on SWE-Bench Verified.** Resolve rates (%) of frontier models on the SWE-bench leaderboards (SWE-bench Team, 2026) and of Qwen3.5-35B-A3B with the methods in Table 1. Bold marks Self-HPO, the reference for Δ. Parameter counts are those stated in the developers' model cards ("–" if not stated).

Model	Harness	Total	Active	Resolve Rate	Δ (ours − model)
Frontier models					
Gemini 3 Flash	mini-SWE-agent	–	–	75.8	−6.3
MiniMax-M2.5	mini-SWE-agent	–	–	75.8	−6.3
Claude Opus 4.6	mini-SWE-agent	–	–	75.6	−6.1
GLM-5	mini-SWE-agent	744B	40B	72.8	−3.3
GPT-5.2	mini-SWE-agent	–	–	72.8	−3.3
Kimi K2.5	mini-SWE-agent	1T	32B	70.8	−1.3
DeepSeek-V3.2	mini-SWE-agent	–	–	70.0	−0.5
Gemini 3 Pro	mini-SWE-agent	–	–	69.6	−0.1
Claude Haiku 4.5	mini-SWE-agent	–	–	66.6	+2.9
Kimi K2 Thinking	mini-SWE-agent	1T	32B	63.4	+6.1
MiniMax-M2	mini-SWE-agent	230B	10B	61.0	+8.5
Devstral Small 2	mini-SWE-agent	24B	24B	56.4	+13.1
GPT-5 mini	mini-SWE-agent	–	–	56.2	+13.3
Qwen3-Coder-480B-A35B	mini-SWE-agent	480B	35B	55.4	+14.1
GLM-4.6	mini-SWE-agent	–	–	55.4	+14.1
Devstral 2	mini-SWE-agent	123B	123B	53.8	+15.7
Qwen3.5-35B-A3B (Table 1)					
Base	mini-SWE-agent	35B	3B	61.0	+8.5
SFT	mini-SWE-agent	35B	3B	65.0	+4.5
Self-Harness	mini-SWE-agent (optimized)	35B	3B	65.2	+4.3
RLVR	mini-SWE-agent	35B	3B	67.5	+2.0
Self-HPO (ours)	mini-SWE-agent (optimized)	35B	3B	**69.5**	0.0

Harness–policy cross-evaluation. Table 3 recombines policies and harnesses from one Self-HPO trajectory. The P_0 column evaluates evolved harnesses with the base policy, while the H_0 row evaluates updated policies with the seed harness.

Comparison with frontier models. Table 4 places our results next to frontier models on SWE-Bench Verified (SWE-bench Team, 2026). All frontier results are mini-SWE-agent entries on the leaderboard (scaffold versions vary); our seed harness is also a mini-SWE-agent loop, but our step limit, sampling settings, and sandbox differ, so the comparison indicates the remaining gap rather than a controlled comparison.

Table 5: **Conceptual comparison of harness–policy optimization methods.** Coding marks evaluation on software-engineering or terminal-agent benchmarks (✓: yes; ✘: no), excluding tool-assisted mathematics. Self-proposers use the target policy, frozen or updated across rounds. SFT and RL denote supervised fine-tuning and reinforcement learning, respectively. Harness search set denotes the task set used to collect diagnostic trajectories for harness optimization; Self-HPO selects hard tasks from preceding RL trajectories.

Method	Coding	Policy Opt	Harness proposer	Harness search set
Meta-Harness (Lee et al., 2026)	✓	Frozen	External model	Fixed
Self-Harness (Zhang et al., 2026a)	✓	Frozen	Frozen self-proposer	Fixed
Task-CoEvolve (Miyai et al., 2026)	✓	Frozen	External model	Evolving (variance-weighted resampling)
WHALE (Kim et al., 2026)	✘	SFT	External model	Fixed
HarnessForge (Chen et al., 2026a)	✘	SFT	External model	Evolving (fresh filtering subsets)
HASE (Luo et al., 2026)	✘	RL	Evolving self-proposer	Fixed
Self-HPO	✓	**RL**	**Evolving self-proposer**	**Evolving (RL-oriented search set)**

C Conceptual Comparison with Related Methods

Table 5 compares coverage of the coding domain, how the policy learns, who proposes harness changes, and how harness-search tasks are selected. We focus on executable harnesses and their components; complementary prompt–weight methods include BetterTogether, FST, E-SPL, and P^{2}O (Soylu et al., 2024; Tiwari et al., 2026; Zhang et al., 2026d; Lu et al., 2026).

The trained solver becomes the next harness proposer. Self-Harness revises its harness with a fixed policy, while WHALE and HarnessForge use external proposers alongside policy learning. Self-HPO instead reuses the updated task-solving policy as the proposer in the next round: the H-step holds policy weights fixed, and the P-step performs RL under the selected harness. HASE also trains a self-proposing policy, but places task solving and harness editing in a joint RL action space, with persistent edits reviewed at phase boundaries. Our distinction is therefore the combination of self-proposal with separate harness search and task-solving RL, not self-proposal alone.

Policy training also determines the next harness-search tasks. The P-step supplies two inputs to the next H-step: the updated proposer P_{t+1} and a search set $\mathcal{S}_{t+1}$ drawn from hard tasks encountered during RL. Thus, policy learning changes both *who proposes* and *which tasks guide the search*. Task-CoEvolve also adapts task selection, but uses outcome variance across candidate harnesses within search rather than difficulty exposed during policy training. Self-HPO updates task IDs between H-steps and keeps them fixed within each H-step, so candidates are compared under the same policy and search set.

D Harness Details and Prompt Templates

Seed harness. The seed harness H_0 is the mini-SWE-agent ReAct loop (Yao et al., 2023; Yang et al., 2024): the agent issues bash tool calls in a task-specific sandbox and ends the episode by printing the patch after a fixed submission command. Its task prompt contains the issue description followed by fixed instructions, including the exact command used to submit the patch.

Harness search space. A candidate harness is a Python class that subclasses the seed agent, together with its prompt templates. It may override the agent step, for example to inspect the agent's state, return feedback to the agent before accepting a submission, replace the submitted patch, or act at any step, and all repository access goes through the sandbox. The policy, sampling configuration (temperature and top-p of 0.95), maximum response length, sandbox resources, and verifier are fixed across candidates.

Harness proposer. Figure 7 shows the system prompt of the harness proposer. The proposer runs as a coding agent with shell access to the parent harness, its trajectories on the search set, reference fixes for search-set tasks, and the outputs of earlier candidates; reference fixes are used only for diagnosis and are never passed to the agent.

Harness Proposer System Prompt

```
You are the harness optimizer for a fixed policy {P_t}. The search set contains {N}
hard training tasks that {P_t} solved at most once during the preceding P-step. The parent
harness is {H_t}, which solves {s}% of the search set under the frozen sampling
configuration. Your goal is to design ONE new harness mechanism, of your own design,
that is added on top of the parent harness and solves more search-set tasks than the
parent.

CONSTRAINTS
- Keep every method of the parent harness verbatim and add exactly one new mechanism.
- The policy, sampling configuration (temperature 0.95, top-p 0.95), step limit,
  maximum tokens, and verifier are frozen.
- A mechanism that fires on every task and adds turns pushes the agent into the step
  limit and hurts performance. Target a detectable failure pattern with a tight
  trigger, and keep the mechanism turn-cheap.
- Never inject reference patches or task-specific facts into the agent's context; the
  mechanism must compute its own signal at runtime.

EVIDENCE (computed from the parent harness's own runs; treat it as evidence, not a
mandate)
- Coverage gap: the source files required by the reference fix versus the files the
  agent touched.
- Precision gap: failures in which every required file was edited but the fix was
  incorrect.
- Submission loss: failures in which a real source edit appears in the trajectory but
  the final submitted patch is empty or malformed.
- Reference-fix archetypes (multi-file, signature change, new import, guard, error
  handling, ...) versus what the agent produced.
Decide for yourself which failure is the largest and most reachable by a harness, or
find another one.

WORKFLOW
0. If a previous candidate exists, read the tasks it made worse and better than the
   parent (trajectories and reference fixes), work out why, and build on it.
1. Read several failing trajectories and compare a few with the reference fix to form
   your own hypothesis about the dominant failure cause.
2. Design a mechanism with a detectable trigger in the agent's own state and a
   concrete action: for example, intercept a submission and return feedback so that
   the agent gets another turn to act on it, replace the submitted patch, or act at
   any step.
3. Implement it in agent.py as `class AgentHarness(ProgressTrackingAgent)`. All
   repository access must go through the sandbox environment.
4. Check that the candidate preserves every parent method and fires on real sandbox
   tasks.
5. Write a one-line purpose (diagnosed cause -> mechanism) and report the candidate
   name.
```

Figure 7: **Harness proposer system prompt.** Condensed from the prompt used in Self-HPO's second H-step. Values in braces change between rounds; the first round starts from the seed harness. Implementation-specific instructions, such as file paths, the sandbox interface, and validation scripts, are omitted.

Optimized harness. Figure 5 shows a condensed view of the harness selected in HPO's second H-step. Figure 8 shows the same harness as a diff against the seed harness. The optimized harness keeps the seed prompts unchanged and inherits the seed agent loop; its only change is an override of `step()` that adds four checks. Three of these checks add a message to the conversation, listed in full in Figure 9.

HPO's second H-step: diff against the seed harness

```
--- seed harness H0:     ProgressTrackingAgent.step()   (inherited agent loop)
+++ optimized harness:    AgentHarness(ProgressTrackingAgent).step()
@@ class AgentHarness(ProgressTrackingAgent) @@
 def step(self):
+    # (1) budget valve: once, shortly before the step or time limit, if nothing was submitted
+    if not self._rescued and not self._submitted_once and (near_step_limit() or near_time_limit()):
+        self._rescued = True
+        self.add_messages(user(RESCUE_TEXT))           # the step then continues as usual
-    return self.execute_actions(self.query())
+    try:
+        return self.execute_actions(self.query())      # unchanged seed step
+    except Submitted as e:                              # the policy issued the submit command
+        self._submitted_once = True
+        sub = e.submission
+        # (2) empty-patch guard: hold an empty submission once
+        if is_empty(sub) and not self._empty_guarded and has_runway():
+            self._empty_guarded = True
+            self.add_messages(user(EMPTY_GUARD_TEXT))
+            return []                                   # not submitted; the policy continues
+        # (3) contract grounding: hold the first non-empty submission once
+        if not is_empty(sub) and not self._grounded and has_runway():
+            self._grounded = True
+            self.add_messages(user(GROUND_TEXT))
+            return []
+        # (4) patch fallback on the final submission
+        if not is_empty(sub) and not applies_cleanly(sub):     # `git apply --check` on the base commit
+            auth = sandbox("git add -N <untracked files> && git diff")
+            if auth: e.submission = auth                # submit the sandbox's own diff instead
+        raise                                           # accept the submission
```

Figure 8: **Harness changes from HPO's second H-step.** Shown as a diff against the seed agent's `step()` (+ added, − removed). Helper calls are condensed into short names; the message texts are listed in Figure 9, and the seed prompts are unchanged.

(1) Budget valve: RESCUE_TEXT

```
[BUDGET ALMOST EXHAUSTED -- STOP EXPLORING AND SUBMIT NOW]

You are nearly out of steps/time and have not submitted a patch. An empty result scores
↪  zero, but a focused best-effort edit can pass. Right now:
1. Decide the single most likely fix location from what you already know -- do NOT start
↪  new open-ended exploration.
2. Make that concrete edit to the source file (the smallest change that matches the PR's
↪  literal request).
3. Immediately create and submit the patch, in SEPARATE commands:
   git diff -- <the source files you modified> > patch.txt
   echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && cat patch.txt
Do this in your next one or two commands.
```

(2) Empty-patch guard: EMPTY_GUARD_TEXT

```
[YOUR SUBMISSION IS EMPTY -- it contains no source-code changes, so it cannot fix anything.
↪  Not finalized.]

You have not edited any source file yet (patch.txt is empty or has no diff hunks). Do NOT
↪  give up. Re-localize the bug:
1. Re-read the PR and grep for the exact symbol/message/flag it names.
2. Open the file where that symbol lives and make the concrete edit the PR requires.
3. IMPORTANT -- did your fix CREATE a NEW file (a brand-new module)? A plain `git diff` does
↪  NOT show untracked new files, so your patch looks empty even though the file exists.
↪  Stage new files first so they appear as additions:
   cd /testbed && git add -N <the new file(s) you created>
4. Verify with `cd /testbed && git --no-pager diff` that a real change (or a `new file mode`
↪  hunk) now shows.
5. Then `cd /testbed && git diff -- <files> > patch.txt` and submit with:
   echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && cat patch.txt
```

(3) Contract grounding: GROUND_TEXT

```
[GROUND YOUR DIFF IN THE REPO'S EXISTING TESTS -- one-time check; your submission is NOT
↪  finalized yet]

The most common way this task is failed is submitting a patch that APPLIES cleanly but
↪  implements the WRONG contract -- a guessed function/option name, a signature the
↪  codebase never uses, a rename or new spelling that no caller expects, or an
↪  error/return shape different from what the code around it uses. Re-reading the PR will
↪  NOT catch this, because your reading of the PR is what produced the wrong contract. The
↪  codebase itself is the authority. The repo ships EXISTING tests that already exercise
↪  the code you touched and the API next to it -- they show the EXACT names, signatures,
↪  argument spellings, and error/return types the code is expected to have. Use them as
↪  ground truth now.

1. Print your changes and identify the concrete symbols you added or changed:
   git --no-pager diff
   List every NEW or RENAMED symbol/option/message string in your diff.
2. For each such symbol, search the repo's tests and existing call sites for how that
↪  symbol -- and the nearest EXISTING sibling API -- is actually used:
   grep -rn --include='*.py' -e '<symbol>' -e '<sibling_api>' . | head -40
   Then OPEN and READ the specific test files that reference the area you changed (e.g.
   ↪  tests/ or test_*.py near your file) and note the exact spelling, call signature, and
   ↪  expected result/exception the codebase uses.
3. Confront your diff against what the tests/call sites actually expect. If you introduced
↪  a name, option spelling, signature, or error shape that NO existing test or caller uses
↪  -- while an existing convention does -- change your diff to MATCH the codebase's
↪  convention. Do not invent a new API; conform to the one already there. Also confirm you
↪  did not MISS a behavior the tests imply is required.
4. If you changed anything, regenerate the patch:
   git diff -- <the source files you modified> > patch.txt
   and re-inspect patch.txt so it contains ONLY your intended source changes.
5. When your diff conforms to the codebase's real contract, re-issue the EXACT submit
↪  command:
   echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && cat patch.txt
This check happens only once -- after you re-submit, it goes through.
```

Figure 9: **Messages added by the optimized harness.** The messages that the harness in Figure 8 adds to the conversation, shown in full. Each is added at most once per task and is followed by another policy turn; the numbers match the checks in Figures 5 and 8.

E Case Study: Where the Harness Intervenes

To illustrate how an evolved harness changes the outcome for a fixed policy, we compare two runs of the final Self-HPO policy P_2 on the SWE-Bench Verified task `sphinx-doc__sphinx-9658`: one under the seed harness H_0 and one under the evolved harness H_2. We show the final steps of each run, labeling the task, the agent's reasoning, its actions, the resulting observations, and any message injected by the harness; "[...]" marks omitted content. Purple notes summarize the key steps, so the example can be followed by reading them alone.

The two runs reach the same point: the policy has edited the source, the existing tests pass, and it prepares to finish with a summary of its changes. Under H_0, this summary becomes the final submission, so no code change is evaluated. Under H_2, the harness holds the first submission, reports that the submitted patch contains no files, and returns control to the policy, which writes the diff, checks the fix, and submits a valid patch. The PR-intent reminder in this example is a false positive, since its keywords come from unrelated parts of the issue text.

Case 1 (Failure Case): Policy P_2 with the seed harness H_0

Task

```
<pr_description>
Consider the following PR description:
Inherited classes not correctly documented when mocked
### Describe the bug
We're experiencing an issue when documenting classes that inherit mocked classes. [...]
The base class is listed as "Bases: `torch.nn.`" instead of "Bases: `torch.nn.Module`".
[...]
</pr_description>
[task instructions of the seed harness]
```

[131 earlier steps omitted]

Agent

```
Good, I only modified the source files, not test files. Let me now run a final
↪  comprehensive test to make sure the fix works correctly:
```

Action

```
cd /testbed && python -m pytest tests/test_util_typing.py tests/test_ext_autodoc.py \
  tests/test_ext_autodoc_automodule.py tests/test_ext_inheritance_diagram.py -x --tb=short
  ↪  2>&1 | tail -20
```

Observation

```
[...]
================== 99 passed, 7 skipped, 7 warnings in 2.00s ===================
```

► (1) The agent's edits are in place and the existing tests pass.

Agent

```
All tests pass. The fix is complete. Let me summarize what was done:
## Summary
[...]
```

Action

```
cd /testbed && echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && cat << 'EOF'
## Summary of Changes
### Issue
When documenting classes that inherit from mocked classes, the "Bases" section
incorrectly showed `torch.nn.` instead of `torch.nn.Module`.
[...]
All existing tests pass with these changes.
EOF
```

► (2) To finish, the agent prints a prose summary of its changes instead of the patch file.

Final submission

```
## Summary of Changes
### Issue
[...]
All existing tests pass with these changes.

=> Prose instead of a patch; no code change is scored (unresolved).
```

► (3) The seed harness accepts this output as the submission. No code change reaches the grader, so the task is unresolved.

Case 2 (Success Case): Policy P_2 with the evolved harness H_2

Task

```
[... same task as above ...]
```

[114 earlier steps omitted]

Agent

```
The fix looks correct. Let me run a broader test to make sure I haven't broken anything:
```

Action

```
cd /testbed && python3 -m pytest tests/test_ext_autodoc*.py tests/test_util_typing.py \
  tests/test_ext_inheritance_diagram.py -x 2>&1 | tail -20
```

Observation

```
[...]
================== 195 passed, 7 skipped, 7 warnings in 3.53s ==================
```

► (1) Same situation as under H_0: the edits are in place and the tests pass.

Agent

```
All 195 tests pass (7 skipped). The fix is complete and working.
Let me write a summary of the fix:
## Summary
[...]
```

Action

```
cd /testbed && echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && cat sphinx/util/typing.py |
↪  head -130 \
  | tail -20 && echo "---" && cat sphinx/util/typing.py | tail -10
```

► (2) The agent again tries to finish without submitting a patch.

Harness

```
[Full Files Auditor - Coverage Check]
----------------------------------------
You touched 1 files during exploration: ['sphinx/util/typing.py']
Your submitted patch has 0 files: (none)

FOUND GAP: 1 touched files NOT in your patch:

Potential missing files:
  - sphinx/util/typing.py

[PR Intent Reminder]
----------------------------------------
Your patch may not address the core PR intent. Key terms from PR:
  - sphinx.ext.viewcode
  - sphinx_autodoc_typehints
  - version
  - sphinxcontrib.apidoc
  - sphinx.ext.napoleon

Your submitted patch does not mention these key terms.
Consider: Does your edit actually address the issue described?
```

► (3) H_2 holds this submission once and reports that the submitted patch contains no files. (The PR-intent reminder is a false positive.)

Agent

```
The task mentioned a patch file was created. Let me verify my changes are correct and
↪  submit them properly.
```

Action

```
cd /testbed && git diff > patch.txt && cat patch.txt
```

Observation

```
diff --git a/sphinx/util/typing.py b/sphinx/util/typing.py
index 87707d48f..142503422 100644
--- a/sphinx/util/typing.py
+++ b/sphinx/util/typing.py
@@ -117,6 +117,12 @@ def restify(cls: Optional[Type]) -> str:
             return cls
         elif cls in INVALID_BUILTIN_CLASSES:
             return ':py:class:`%s`' % INVALID_BUILTIN_CLASSES[cls]
+        elif _get_ismock()(cls):
+            # This is a mock object. Use its type's attributes.
+            if sys.version_info >= (3, 7):
+                return _restify_py37(type(cls))
+            else:
+                return _restify_py36(type(cls))
         elif inspect.isNewType(cls):
[...]
```

► (4) Prompted by the feedback, the agent writes its working-tree changes into a patch.

Agent

```
The patch shows my changes correctly. Let me verify the fix works correctly and then
↪  submit it.
```

Action

```
cd /testbed && python3 -c "
from sphinx.ext.autodoc.mock import _MockModule
from sphinx.util.typing import restify
mock_mod = _MockModule('torch.nn')
class DeepKernel(mock_mod.Module):
    pass
result = restify(DeepKernel.__orig_bases__[0])
print('Result:', result)
[...]"
```

Observation

```
Result: :py:class:`torch.nn.Module`
Expected: :py:class:`torch.nn.Module`
Fix works: True
```

► (5) It checks the fix on the example from the issue.

Agent

```
The fix is working correctly. Now let me submit the final output.
```

Action

```
echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && cat /testbed/patch.txt
```

Final submission

```
diff --git a/sphinx/util/typing.py b/sphinx/util/typing.py
[...]

=> Valid patch (resolved).
```

► (6) The final submission is a valid patch, which resolves the task.