HARNESSPRO: RECURSIVE HARNESS SELF-IMPROVEMENT WITH PARETO OPTIMIZATION FOR PERFORMANCE-COST TRADE-OFFS

Anonymous authors
Paper under double-blind review

ABSTRACT

Harness development plays a crucial role in building capable AI agents. Recent approaches improve harnesses through recursive self-improvement (RSI), but their performance gains often come with higher execution costs. We introduce HARNESSPRO, a label-free method for evolving harnesses to improve task performance while reducing execution cost. In this setting, candidate harnesses are evaluated using LLM-generated tests without access to ground-truth solutions or official tests. Specialized agents diagnose execution trajectories, plan updates with different performance-cost priorities, and implement them as candidate harnesses. Rather than returning a single harness, HARNESSPRO uses normalized weighted squared-distance selection to provide **cost-focused** (lite), **balanced** (medium), and **performance-focused** (heavy) harnesses for different deployment budgets. Selected harnesses enter the next iteration, where execution feedback guides further updates. Our experiments cover Terminal-Bench 2, SWE-Bench Pro, and ProgramBench, with GPT-5.5 and Codex as well as GLM-5.2 and Claude Code. Across these benchmarks with GPT-5.5 and Codex, the heavy mode of HARNESSPRO averages a 2.0 percentage point gain in pass rate and a 26.1% reduction in per-task cost relative to the strongest prior label-free baseline. This advantage also holds for GLM-5.2 and Claude Code. Code is available at this link.

1 INTRODUCTION

An agent harness is the software infrastructure that manages interactions between a large language model (LLM) and its environment (Grace et al., 2026). Recent work on recursive self-improvement (RSI) (Zhang et al., 2026) uses past execution trajectories to diagnose problems and guide harness updates that improve performance on future tasks (Suzgun et al., 2026; Lin et al., 2026; Pan et al., 2026). However, these methods often pursue performance gains through more extensive exploration and stronger verification (Lin et al., 2026; Pan et al., 2026). These additional efforts improve performance, but often at the cost of higher token consumption and increased latency. This trade-off is less appealing in practical deployment, where the execution cost and latency also matter. We therefore ask: ***Can recursive harness self-improvement deliver better performance at lower execution cost?***

Toward this goal, we introduce HARNESSPRO, a label-free method for recursive harness self-improvement that uses both performance and cost metrics to guide harness updates and candidate selection. It orchestrates several specialized agents to turn execution feedback into successive harness updates. A trajectory analyzer examines repeated rollouts to identify failures and avoidable overhead. Guided by different **performance-cost priorities**, a planner turns these diagnoses into harness improvement plans, which an editor implements as executable harness candidates. A test-synthesis agent provides fixed tests for evaluating the candidates without access to ground-truth solutions or official tests. Candidates that underperform the initial harness or are Pareto-dominated are discarded. To accommodate different budgets, we then use *normalized weighted squared-distance selection* (Marler & Arora, 2004) to retain representative harnesses for **cost-focused** (lite), **balanced** (medium), and **performance-focused** (heavy) modes. The selected harnesses then enter the next epoch, where further updates pursue different performance-cost priorities.

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.

We evaluate HARNESSPRO on Terminal-Bench 2 (The Terminal-Bench Team, 2025), SWE-Bench Pro (Deng et al., 2026), and ProgramBench (Yang et al., 2026a), using GPT-5.5 with Codex and GLM-5.2 with Claude Code. With both backbone models, HARNESSPRO **improves task performance** while **reducing execution cost**. For example, on Terminal-Bench 2 with GPT-5.5 and Codex, the heavy mode of HARNESSPRO outperforms RHO (Pan et al., 2026) in pass rate by 4.5 percentage points while reducing mean cost per task by 33.9%. For more cost-sensitive deployments, the medium mode delivers performance comparable to RHO at substantially lower execution cost.

An analysis of the evolved harnesses shows that HARNESSPRO does not improve agents simply by compressing context or increasing reasoning effort. The evolved harnesses ***dynamically*** allocate ***reasoning effort*** and ***tool budgets*** based on task descriptions, while runtime hooks steer execution from exploration toward convergence based on the assigned budgets. They also adopt tool-output compression like RTK (Szymkowiak, 2026) and streamline instructions to reduce redundant prompt, while retaining necessary context and final verification. Together, these mechanisms curb ***avoidable overhead*** while preserving the reasoning and verification needed for task success, thereby advancing the performance-cost ***Pareto*** frontier.

Our contributions are threefold:

- We introduce HARNESSPRO, a multi-agent framework that simultaneously optimizes performance and cost by analyzing execution trajectories and recursively improving harnesses.
- We assess candidate artifacts with fixed LLM-generated tests rather than direct LLM judgments, improving failure detection without access to official tests.
- Across three benchmarks, HARNESSPRO improves performance while reducing cost and latency, yielding a series of harnesses with different performance-cost trade-offs.

2 RELATED WORK

Agent and harness optimization. Agent optimization has expanded from prompts and predefined workflows to the executable harness surrounding a model. AFlow searches graphs of model-invoking operators using execution feedback (Zhang et al., 2025). More recent systems directly modify harness components: agentic harness engineering makes edits traceable (Lin et al., 2026), HarnessFix diagnoses failures before applying repairs (Chen et al., 2026a), and HarnessX composes harnesses from typed primitives using trajectory evidence (Chen et al., 2026b). RHO uses LLMs to evaluate and compare execution trajectories without ground-truth labels, guiding harness evolution toward higher task success (Pan et al., 2026). In contrast, our method, HARNESSPRO, optimizes both task success rate and monetary cost. Rather than returning a single harness, it produces a set of harnesses with different performance-cost trade-offs, organized into lite, medium, and heavy tiers.

Component optimization and inference-time budget allocation. Some methods focus on specific components, such as skills or memory. SkillOpt optimizes skill documents (Yang et al., 2026b), while SkillMOO and FRAMES incorporate cost or regression constraints into skill evolution (Gong et al., 2026a; Wang et al., 2026). Dynamic Cheatsheet and ReasoningBank retain reusable strategies from past experience (Suzgun et al., 2026; Ouyang et al., 2026), while Sleep-time Compute builds reusable memory by reasoning over context before a query arrives (Lin et al., 2025). MemoryCPT jointly optimizes memory construction, retrieval, and summarization (Lei et al., 2026). Other work addresses budget allocation at inference time: BATS learns a budget-aware policy for allocating model tokens and external tool calls (Liu et al., 2025), while COMPAS jointly selects models, prompts, and decoding settings according to task difficulty (Gong et al., 2026b). HARNESSPRO instead uses past trajectories to evolve the entire harness, including instructions, skills, tools, hooks, and verification rules. Table 1 summarizes how HARNESSPRO differs from prior work.

3 PROBLEM SETTING

A *harness* $h \in \mathcal{H}$ is the software runtime layer that manages a foundation model's interactions and execution. It includes prompts, tools and skills, context and memory mechanisms, hooks, and verification policies (Yehudai et al., 2026; OpenAI, 2026). Given a task x, the harness determines what information the agent receives, which actions it can take, and how those actions are executed,

Table 1: Comparison of different harness self-improvement methods.

Method	Goal	Scope	Feedback	Iteration	Output
AFlow (Zhang et al., 2025)	Pass	Workflow	Ground truth	Recursive	Single
SkillMOO (Gong et al., 2026a)	Pass + Cost	Skills	Ground truth	Recursive	Pareto set
RHO (Pan et al., 2026)	Pass	Skills + tools	Label-free	Single round	Single
AHE (Lin et al., 2026)	Pass	Harness	Ground truth	Recursive	Single
HARNESSPRO	Pass + Cost	Harness	Label-free	Recursive	Pareto set

directly influencing both task success and resource consumption. An agent equipped with harness h solves task x, producing a trajectory $\tau_h(x) = \mathrm{Solve}(h, x)$ that records tool calls, environment feedback, and the final artifact. Recursive self-improvement uses trajectories from previous tasks to guide harness evolution and improve performance on future tasks. Below, we define the three evaluation metrics: pass rate, execution cost, and runtime latency.

Pass rate. To prevent test-set leakage, we partition tasks into training, validation, and test sets, denoted by $\mathcal{D}_{\mathrm{train}}$, $\mathcal{D}_{\mathrm{val}}$, and $\mathcal{D}_{\mathrm{test}}$. Outcomes on training set are used to update harnesses, while the validation set is used to evaluate and select candidate harnesses. The test set is used only for final evaluation after evolution. For a task $x \in \mathcal{D}_{\mathrm{test}}$, the official tests evaluate the final artifact produced by $\tau_h(x)$. We set $\mathrm{Pass}(\tau_h(x)) = 1$ if all official tests pass and $\mathrm{Pass}(\tau_h(x)) = 0$ otherwise. The pass rate is the fraction of test tasks that pass all official tests, $\mathrm{PR}(h) = \frac{1}{|\mathcal{D}_{\mathrm{test}}|} \sum_{x \in \mathcal{D}_{\mathrm{test}}} \mathrm{Pass}(\tau_h(x))$.

Cost. We compute execution cost from token usage, distinguishing uncached input, cached input, and output tokens because they are billed at different prices. For each task x, we aggregate these token counts over all model calls in $\tau_h(x)$ and compute its cost $c_h(x)$ using the prices in Table 4 (Appendix A). We report the average cost per test task as $C(h) = \frac{1}{|\mathcal{D}_{\mathrm{test}}|} \sum_{x \in \mathcal{D}_{\mathrm{test}}} c_h(x)$.

Latency. We define task latency as the wall-clock time from the start of agent execution to its completion: $\ell_h(x) = T_{\mathrm{end}}(\tau_h(x)) - T_{\mathrm{start}}(\tau_h(x))$ The time spent grading the final output is not included in task latency. We average task latencies over the test set to obtain the mean latency, $L(h) = \frac{1}{|\mathcal{D}_{\mathrm{test}}|} \sum_{x \in \mathcal{D}_{\mathrm{test}}} \ell_h(x)$. Because model response times fluctuate with server load and network conditions, we treat latency as an auxiliary metric rather than an optimization objective.

Optimization objective. We aim to evolve harnesses that improve performance while reducing cost. We formulate the optimization problem as

$$\min_{h \in \mathcal{H}} \left(-\mathrm{PR}(h),\, C(h)\right) \quad \text{s.t.} \quad \mathrm{PR}(h) \geq \mathrm{PR}(h_0). \tag{1}$$

Here, h_0 is the initial harness. We seek the ***Pareto*** frontier and select three ***representative*** operating points: cost-focused (*lite*), balanced (*medium*), and performance-focused (*heavy*). However, finding these trade-offs poses two challenges. **(1) A vast harness search space.** A harness comprises components such as instructions, tools, skills, hooks, and rules, all of which can be modified during evolution. Their possible combinations create a vast, discrete search space that cannot be optimized using gradients. **(2) Balancing performance and cost.** Additional reasoning, exploration, or verification can improve performance but also consumes more tokens, increasing cost. Conversely, indiscriminately reducing these activities may hurt performance. Advancing the Pareto frontier thus requires analyzing execution trajectories alongside task outcomes and token usage to distinguish necessary computation from avoidable overhead and find the "sweet spot" that reduces cost without compromising performance.

4 HARNESSPRO

To address these challenges, we introduce HARNESSPRO, a multi-agent framework that analyzes execution trajectories and iteratively modifies harnesses to advance the performance-cost Pareto frontier. We first outline the evolution process, then describe trajectory-based diagnosis and candidate generation, followed by test-based evaluation and selection.

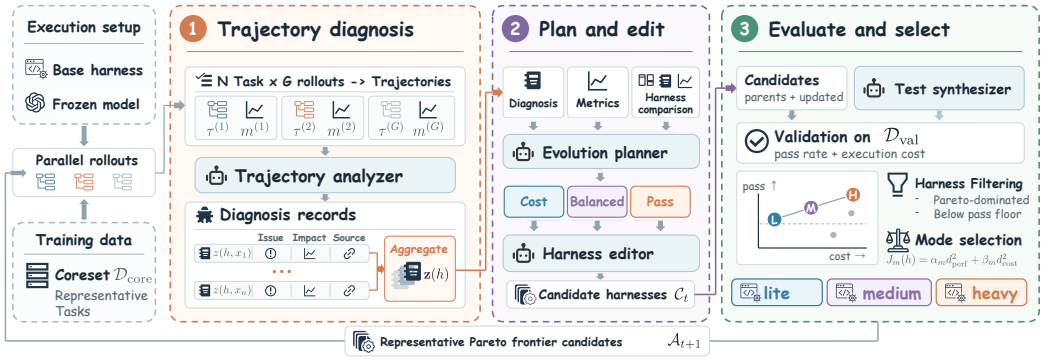


Figure 1: Overview of HARNESSPRO. In each epoch, it analyzes rollout trajectories, generates candidate harnesses, and selects representatives based on performance and cost.

4.1 OVERVIEW

To keep trajectory collection and analysis manageable, we begin by selecting a training coreset $\mathcal{D}_{\text{train}}$. Following RHO (Pan et al., 2026), we use the Determinantal Point Process (DPP) kernel (Kulesza & Taskar, 2012) to balance task difficulty and diversity (Appendix A.1). Starting from an executable harness h_0, we maintain a ***candidate set*** $\mathcal{A}_t$ of harnesses at epoch t, initialized as $\mathcal{A}_0 = \{h_0\}$. In each epoch, HARNESSPRO employs a multi-agent framework to analyze trajectories of these candidate harnesses on the coreset $\mathcal{D}_{\text{train}}$ to identify task failures, unproductive exploration, and redundant computation. The framework then plans and implements harness modifications to address these issues, producing new executable candidate harnesses as described in Section 4.2.

We evolve harnesses in a label-free setting, ***without*** access to ground-truth solutions, official test cases, or graders. Instead of using an LLM to assess and compare trajectories directly, as in RHO (Pan et al., 2026), we use it to generate tests from task descriptions and trajectories produced by the initial harness. In each epoch, we evaluate candidate harnesses on the validation set using the generated tests, measuring ***pass rate*** and ***execution cost***. After discarding candidates that fall below the performance floor or are Pareto-dominated, we use normalized weighted squared-distance selection to retain ***representatives*** for cost-focused (lite), balanced (medium), and performance-focused (heavy) modes. The selected harnesses form the candidate set for the next epoch. This process repeats until the epoch limit is reached, after which we fix the harnesses and evaluate them on the test set. Algorithm 1 summarizes the full procedure.

4.2 TRAJECTORY ANALYSIS AND HARNESS EVOLUTION

At epoch t, we run each harness in the current candidate set $\mathcal{A}_t = \{h_i^t\}_{i=1}^{|\mathcal{A}_t|}$ independently G times on every training task $x \in \mathcal{D}_{\text{train}}$. For candidate harness $h \in \mathcal{A}_t$, this rollout process yields a trajectory $\tau_h^{t,g}(x)$ and a metric vector $m_h^{t,g}(x) = \big(\text{Pass}_h^{t,g}(x),\, c_h^{t,g}(x),\, \ell_h^{t,g}(x)\big)$. Here, $\text{Pass}_h^{t,g}(x)$ is 1 if the artifact produced in $\tau_h^{t,g}(x)$ passes all generated tests in $\mathcal{S}(x)$, while $c_h^{t,g}(x)$ and $\ell_h^{t,g}(x)$ record its execution cost and latency. We also retain token usage and tool-call details for later diagnosis. These trajectories and artifacts guide harness self-improvement through a multi-agent framework comprising a ***trajectory analyzer***, an ***evolution planner***, a ***harness editor***, and a ***test synthesizer***. We describe each agent below.

Trajectory Analyzer. Execution trajectories can reveal harness weaknesses, but the relevant evidence is often dispersed across long sequences of model responses and tool outputs, making these weaknesses difficult to locate and interpret. We therefore implement a trajectory-analysis agent to diagnose these problems for each candidate h and task x at epoch t. To reduce the impact of randomness, the agent jointly examines the trajectories $\{\tau_h^{t,g}(x)\}_{g=1}^{G}$ from G independent rollouts and their metrics $\{m_h^{t,g}(x)\}_{g=1}^{G}$, rather than relying on a single attempt:

$$z(h,x) = \text{Analyzer}\Big(x, \big\{\big(\tau_h^{t,g}(x), m_h^{t,g}(x)\big)\big\}_{g=1}^{G}\Big). \tag{2}$$

Algorithm 1 Cost-aware recursive harness self-improvement with HARNESSPRO.

Input: Seed harness h_0; data $\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{val}}$; prompts I_1–I_3; rollouts G; epoch limit T
Output: Harnesses for the lite h^T_{lite}, medium h^T_{medium}, and heavy h^T_{heavy} modes.

Collect $\tau_{h_0}(x)$ for each $x \in \mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{val}}$
$\mathcal{S}(x) \leftarrow \mathbf{TestSynth}(x, \tau_{h_0}(x))$ for each x ▷ Generate once
Evaluate h_0 on $\mathcal{D}_{\text{val}}$ using $\mathcal{S}$; fix its initial pass rate $\text{PR}_{\text{val}}(h_0)$ as the performance floor
$\mathcal{A}_0 \leftarrow \{h_0\}$; $t \leftarrow 0$; $h^0_m \leftarrow h_0$ for $m \in \{\text{lite}, \text{medium}, \text{heavy}\}$
while $t < T$ **do**
  Stage 1 Trajectory diagnosis
  for each $h \in \mathcal{A}_t$ **do**
    Collect G independent rollouts per $x \in \mathcal{D}_{\text{train}}$, recording $\tau^{t,g}_h(x)$ and $m^{t,g}_h(x)$ using $\mathcal{S}(x)$
    $z(h,x) \leftarrow \mathbf{Analyzer}(x, \{(\tau^{t,g}_h(x), m^{t,g}_h(x))\}_{g=1}^G)$ for each $x \in \mathcal{D}_{\text{train}}$
    $\mathbf{z}(h) \leftarrow \bigcup_{x \in \mathcal{D}_{\text{train}}} z(h,x)$; $\mathbf{m}(h) \leftarrow \bigcup_{x \in \mathcal{D}_{\text{train}}} \{m^{t,g}_h(x)\}_{g=1}^G$
  end for
  Stage 2 Candidate generation
  for each $h^t_i \in \mathcal{A}_t$ and $j \in \{1,2,3\}$ **in parallel do**
    Generate a plan from I_j, diagnoses, metrics, and cross-candidate evidence:
    $\mathbf{p}_j(h^t_i) \leftarrow \mathbf{Plan}(I_j, h^t_i, \mathbf{z}(h^t_i), \mathbf{m}(h^t_i), \{(\text{Diff}(h^t_i, h^t_k), \mathbf{z}(h^t_k), \mathbf{m}(h^t_k))\}_{k \neq i})$
    $\widetilde{h}^t_{i,j} \leftarrow \mathbf{Edit}(h^t_i, \mathbf{z}(h^t_i), \mathbf{p}_j(h^t_i))$; commit $\widetilde{h}^t_{i,j}$ to Git ▷ Separate workspace
  end for
  $\mathcal{C}_t \leftarrow \{\widetilde{h}^t_{i,j}\}_{i,j}$; $\widetilde{\mathcal{C}}_t \leftarrow \mathcal{A}_t \cup \mathcal{C}_t$
  Stage 3 Evaluation and selection
  Evaluate each $h \in \widetilde{\mathcal{C}}_t$ once per $x \in \mathcal{D}_{\text{val}}$ using the fixed $\mathcal{S}(x)$; aggregate $\mathbf{m}_{\text{val}}(h)$
  Filter candidates: $\widetilde{\mathcal{A}}_{t+1} \leftarrow \{h \in \widetilde{\mathcal{C}}_t \mid \text{PR}_{\text{val}}(h) \geq \text{PR}_{\text{val}}(h_0) \wedge \nexists h' \in \widetilde{\mathcal{C}}_t : h' \prec h\}$
  Select representatives: $h^{t+1}_m \leftarrow \arg\min_{h \in \widetilde{\mathcal{A}}_{t+1}} J_m(h)$ for $m \in \{\text{lite}, \text{medium}, \text{heavy}\}$ ▷ Eq. 9
  Update candidates: $\mathcal{A}_{t+1} \leftarrow \{h^{t+1}_{\text{lite}}, h^{t+1}_{\text{medium}}, h^{t+1}_{\text{heavy}}\}$; $t \leftarrow t+1$
end while
return $(h^T_{\text{lite}}, h^T_{\text{medium}}, h^T_{\text{heavy}})$ ▷ Fixed before held-out test evaluation

The analyzer's output $z(h,x)$ reports (1) identified problems, (2) their impact on performance and cost, and (3) their sources and supporting evidence. The evidence includes a concise trajectory summary or relevant excerpts, together with the path of the trajectory file and the relevant event range. Two examples are provided in Appendix C.1. We combine diagnoses across different tasks as $\mathbf{z}(h) = \bigcup_{x \in \mathcal{D}_{\text{train}}} z(h,x)$. If $z(h)$ exceeds 64K tokens, the analyzer summarizes it. We similarly collect the metric vectors across training tasks as $\mathbf{m}(h) = \bigcup_{x \in \mathcal{D}_{\text{train}}} \{m^{t,g}_h(x)\}_{g=1}^G$. The planner uses these diagnoses and metrics to guide its proposals.

Evolution Planner. Building on these diagnoses, the planner proposes three improvement plans for each candidate. Each plan is guided by a distinct instruction: I_1 emphasizes cost reduction, I_2 balances performance and cost, and I_3 prioritizes performance, as detailed in Appendix B. The planner starts with the full implementation of h^t_i, using its diagnoses and metrics to identify where modifications are needed. It also compares harness h^t_i and the other candidates h^t_k by examining their code differences $\text{Diff}(h^t_i, h^t_k)$:

$$\mathbf{p}_j(h^t_i) = \text{Plan}(I_j, h^t_i, \mathbf{z}(h^t_i), \mathbf{m}(h^t_i), \{(\text{Diff}(h^t_i, h^t_k), \mathbf{z}(h^t_k), \mathbf{m}(h^t_k))\}_{k \neq i}). \tag{3}$$

By comparing candidates, the planner can identify effective mechanisms and remaining bottlenecks. Each plan explains which components to modify, how to modify them, and the expected effects on performance and cost, as exemplified in Appendix C.2. Full trajectories are not provided to the planner by default. When needed, it can inspect them using the file paths recorded in the diagnoses. The planner is instructed to propose generalizable, rather than task-specific, harness updates. Throughout evolution, the validation and test sets remain hidden from the planner to avoid benchmark leakage.

Harness Editor. We then design a harness editor to modify candidate h^t_i, producing a new, executable candidate harness using the proposed plan $\mathbf{p}_j(h^t_i)$ and the diagnoses $\mathbf{z}(h^t_i)$:

$$\widetilde{h}^t_{i,j} = \text{Edit}\big(h^t_i, \mathbf{z}(h^t_i), \mathbf{p}_j(h^t_i)\big), \qquad j = 1,2,3. \tag{4}$$

Each of the three plans is implemented in a separate workspace, and the resulting harness is committed to Git so that changes can be traced and reverted. Newly generated harnesses form $\mathcal{C}_t$, and

we combine them with the parent harnesses in $\mathcal{A}_t$ for evaluation:

$$\mathcal{C}_t = \bigcup_{i=1}^{|\mathcal{A}_t|} \left\{ \widetilde{h}_{i,1}^t, \widetilde{h}_{i,2}^t, \widetilde{h}_{i,3}^t \right\}, \qquad \widetilde{\mathcal{C}}_t = \mathcal{A}_t \cup \mathcal{C}_t. \tag{5}$$

All harnesses in $\widetilde{\mathcal{C}}_t$ then undergo the same evaluation and selection process, as described below.

4.3 Self-Supervised Evaluation and Iterative Selection

Test Synthesizer. During harness evolution, we follow RHO (Pan et al., 2026), adopting a label-free setting without access to ground-truth solutions, official tests, or grader feedback. RHO relies on LLM self-preference to select harnesses. However, we find that LLM judges tend to be overly optimistic when assessing the execution trajectories, often treating flawed implementations as successful. We therefore introduce a dedicated test-synthesis agent to generate test cases for evaluation; its instruction, skill, and validation tool are outlined in Appendix C.3. For each task $x \in \mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{val}}$, the agent generates tests from its description x and the initial trajectory $\tau_{h_0}(x)$:

$$\mathcal{S}(x) = \text{TestSynth}(x, \tau_{h_0}(x)). \tag{6}$$

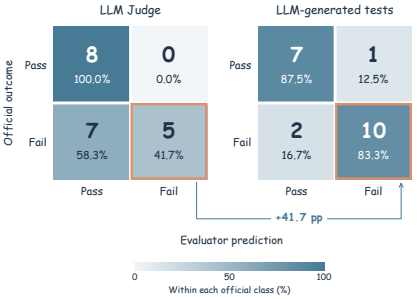


Figure 2: Confusion matrices pooled over our split training set of Terminal-Bench 2 and SWE-Bench Pro.

These generated tests remain fixed throughout the self-improvement process, ensuring a fair comparison among candidate harnesses. A preliminary experiment demonstrates the effectiveness of this approach, with failure recall increasing from 41.7% to 83.3% compared with direct LLM judgments (Figure 2). After each epoch, we run each candidate $h \in \widetilde{\mathcal{C}}_t$ once per validation task x and check its output against $\mathcal{S}(x)$. Task-level pass indicators, costs, and latencies are averaged to obtain the validation metrics $\mathbf{m}_{\text{val}}(h) = (\text{PR}_{\text{val}}(h), C_{\text{val}}(h), L_{\text{val}}(h))$.

Candidate filtering and selection. To prevent the candidate set from growing indefinitely, we retain at most three harnesses for the next epoch. We first discard candidates below the initial pass-rate floor $\text{PR}_{\text{val}}(h_0)$ or Pareto-dominated by another candidate:

$$\widetilde{\mathcal{A}}_{t+1} = \left\{ h \in \widetilde{\mathcal{C}}_t \;\middle|\; \text{PR}_{\text{val}}(h) \geq \text{PR}_{\text{val}}(h_0) \wedge \nexists h' \in \widetilde{\mathcal{C}}_t : h' \prec h \right\}. \tag{7}$$

Here, $h' \prec h$ denotes Pareto dominance: h' is no worse than h on both metrics and strictly better on at least one. We select representatives using a normalized weighted squared-distance criterion (Marler & Arora, 2004) that accounts for actual differences in performance and cost. We first measure each candidate's normalized deviations from the highest pass rate and lowest cost:

$$\begin{aligned} d_{\text{perf}}(h) &= (P_{\max} - \text{PR}_{\text{val}}(h))/(P_{\max} - P_{\min}), \\ d_{\text{cost}}(h) &= (C_{\text{val}}(h) - C_{\min})/(C_{\max} - C_{\min}). \end{aligned} \tag{8}$$

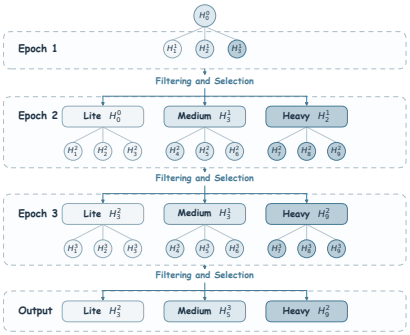


Figure 3: Schematic of recursive harness update, filtering, and selection.[2]

Here, $P_{\min}, P_{\max}$ and $C_{\min}, C_{\max}$ are the extrema of validation pass rate and mean cost over the current filtered set $\widetilde{\mathcal{A}}_{t+1}$. The selection objective for mode m is $J_m(h) = \alpha_m d_{\text{perf}}(h)^2 + \beta_m d_{\text{cost}}(h)^2$, where α_m and β_m weight performance and cost, respectively. Each mode $m \in \{\text{lite}, \text{medium}, \text{heavy}\}$ independently selects the candidate with the smallest weighted squared distance:

$$h_m^{t+1} = \arg\min_{h \in \widetilde{\mathcal{A}}_{t+1}} J_m(h). \tag{9}$$

The selected harnesses form $\mathcal{A}_{t+1}$ for the next round, as illustrated in Figure 3. Filtering and selection use only validation metrics, with no access to test-set results. The final harnesses are evaluated on the held-out test set $\mathcal{D}_{\text{test}}$ only after all rounds of recursive self-improvement are complete.

[2] In practice, filtering and selection may retain fewer than three distinct harnesses, see Figure 4.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Data and implementation. We evaluate HARNESSPRO on Terminal-Bench 2 (The Terminal-Bench Team, 2025), SWE-Bench Pro (Deng et al., 2026), and ProgramBench (Yang et al., 2026a), covering terminal tasks, repository-level issue resolution, and program reconstruction. Following the dataset splitting protocol of RHO (Pan et al., 2026), we hold out 59, 100, and 100 tasks, respectively, for testing. From the remaining tasks in each benchmark, we select a 10-task training set $\mathcal{D}_{\text{train}}$ using the DPP-based coreset selection method (Kulesza & Taskar, 2012) described in Section 4.1. The remaining tasks form the validation set. We pair GPT-5.5 with `Codex 0.147.0` and GLM-5.2 with `Claude Code 2.1.231`. Each candidate harness produces $G = 3$ rollouts per training task in each epoch. Evolution runs for three epochs on Terminal-Bench 2 and SWE-Bench Pro, and four on ProgramBench, retaining up to three harnesses after each epoch. All methods use the same data splits and grading settings. Further implementation details are in Appendix A.

Baselines. We compare HARNESSPRO against several strong harness optimization methods. The original Codex or Claude Code harness is *Vanilla*, the starting point for optimization. *Dynamic Cheatsheet* (Suzgun et al., 2026) maintains reusable strategies and code snippets, *ReasoningBank* (Ouyang et al., 2026) retrieves strategies distilled from past trajectories, and *Sleep-time Compute* (Lin et al., 2025) prepares reusable memory before new tasks. *RHO* (Pan et al., 2026) revises harness components through trajectory analysis and selects candidates using LLM self-preference. To separate the effects of cost-aware and recursive optimization, we include two RHO variants. *Balance* changes only RHO's optimization instruction, asking for better task performance at lower cost. *RHO+Iteration* runs *Balance* for the same number of epochs as HARNESSPRO. Each subsequent epoch starts from the harness selected in the preceding epoch by the best-of-N procedure of RHO.

Evaluation metrics. On Terminal-Bench 2 and SWE-Bench Pro, a task passes only if all its official tests pass. For ProgramBench, we follow its official evaluation protocol (Yang et al., 2026a) and report pass rate as the macro-average of the fraction of test cases passed for each task. We also report mean cost $C(h)$ and latency $L(h)$. For HARNESSPRO, all metrics are given as mean $\pm$ sample standard deviation over three runs on Terminal-Bench 2 and SWE-Bench Pro, and two on ProgramBench. The token prices used to calculate cost are listed in Table 4 in the appendix.

5.2 MAIN RESULTS

Table 2 compares HARNESSPRO with several baseline methods across three benchmarks. With GPT-5.5 and Codex, RHO improves performance over Vanilla on all three benchmarks, but also increases execution cost. On SWE-Bench Pro, for example, its mean cost per task rises by 42.6%, from \$1.88 to \$2.68. In contrast, the heavy mode of HARNESSPRO achieves the ***highest*** pass rate across all three benchmarks ***at lower cost than RHO***. On Terminal-Bench 2, the heavy mode of HARNESSPRO reaches a mean pass rate of 80.8% across three runs, 4.5 percentage points above RHO, with 33.9% lower mean cost. The advantage extends to ProgramBench, where its **69.6%** pass rate exceeds RHO's 68.8% at **17.8%** lower mean cost (\$2.95 vs. \$3.59). These results show that better task performance need not come at a higher execution cost. For the medium mode, we retain strong task performance while reducing cost further. On SWE-Bench Pro, it achieves a 79.0% pass rate compared with RHO's 80.0%, while reducing the average cost per task by 53.0%, from \$2.68 to \$1.26. The lite mode lowers cost further, outperforming Vanilla on all three benchmarks while **spending 17.7–57.4% less**, making it an economical choice for deployment under strict budget constraints. Even with comparable iteration budgets, HARNESSPRO achieves a better performance–cost trade-off than *RHO+Iteration*. Our medium mode matches the performance of *RHO+Iteration* on Terminal-Bench 2 at 31.7% lower cost. Moreover, on ProgramBench, our heavy mode raises pass rate by 1.3 percentage points while lowering mean cost by 5.8%.

The results with GLM-5.2 and Claude Code follow a similar pattern: the heavy mode of HARNESSPRO surpasses RHO on all three benchmarks at lower cost, while the medium and lite modes outperform Vanilla in pass rate, often at ***substantially lower cost***. With GLM-5.2 and Claude Code, the heavy mode also ***reduces mean latency*** on SWE-Bench Pro from RHO's 520 seconds to 412 seconds. Although latency is not an objective, cost optimization also yields faster execution.

Table 2: Performance and efficiency across three benchmarks. P denotes pass rate (%), while C and T denote mean cost ($) and execution time (s). We report mean and standard deviation for HARNESSPRO. On ProgramBench, P is the macro-average of the fraction of tests passed per task.

Method	Terminal-Bench 2			SWE-Bench Pro			ProgramBench		
	P(%) ↑	C($) ↓	T(s) ↓	P(%) ↑	C($) ↓	T(s) ↓	P(%) ↑	C($) ↓	T(s) ↓
GPT-5.5 × Codex									
Vanilla	69.5	0.89	260	59.0	1.88	337	60.3	2.65	669
Dynamic Cheatsheet	72.9	1.06	421	62.0	2.19	335	68.5	3.58	742
ReasoningBank	74.6	0.99	369	61.0	2.13	341	62.5	3.06	661
Sleep-time Compute	74.6	1.04	331	62.0	2.26	337	67.7	3.38	**637**
RHO	76.3	1.09	354	80.0	2.68	437	68.8	3.59	928
Balance	74.6	0.79	355	78.0	2.08	399	65.7	3.29	865
RHO+Iteration	76.3	0.82	328	80.0	1.90	375	68.3	3.13	789
HarnessPro-Lite	72.9± 1.7	**0.43**± 0.01	**230**± 32	72.3± 0.6	**0.80**± 0.02	**203**± 19	64.3± 0.04	**2.18**± 0.06	985± 70
HarnessPro-Medium	76.3± 1.7	0.56± 0.04	253± 35	79.0± 1.0	1.26± 0.03	326± 25	67.9± 0.47	2.90± 0.13	745± 61
HarnessPro-Heavy	**80.8**± 1.0	0.72± 0.05	310± 31	**80.7**± 0.6	1.97± 0.02	333± 10	**69.6**± 1.03	2.95± 0.13	888± 32
GLM-5.2 × Claude Code									
Vanilla	74.6	0.92	963	56.0	1.17	402	59.7	11.34	**3847**
RHO	78.0	1.00	1318	75.0	2.17	520	63.3	17.00	6434
HarnessPro-Lite	75.1± 1.0	**0.43**± 0.03	**665**± 8	70.7± 0.6	**0.70**± 0.02	**289**± 3	60.1± 0.6	**11.10**± 0.28	3907± 32
HarnessPro-Medium	76.3± 0.0	0.50± 0.01	968± 17	75.0± 1.0	0.87± 0.01	332± 10	63.0± 0.4	13.74± 0.16	5893± 132
HarnessPro-Heavy	**81.4**± 1.7	0.62± 0.02	1080± 15	**77.3**± 0.6	1.01± 0.03	412± 11	**64.4**± 0.8	14.95± 0.08	5925± 110

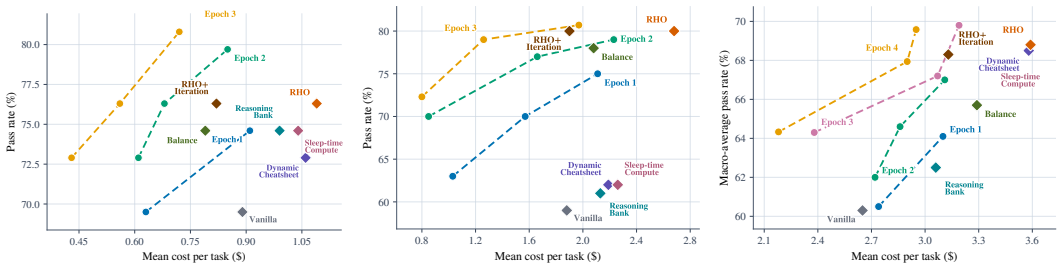
5.3 ANALYSIS OF EVOLVED HARNESSES

To understand how HARNESSPRO improves the performance-cost trade-off, we inspect its evolved harnesses. On Terminal-Bench 2 and SWE-Bench Pro, they select an execution profile from each task description. Each profile specifies a reasoning effort level and a tool-call budget. The lite, medium, and heavy modes use **different rubrics** for profile selection. Runtime hooks **enforce** the selected budget. For example, on Terminal-Bench 2, when HARNESSPRO-Medium selects the standard profile, the hooks remind the agent to produce parallel tool calls and avoid redundant checks after 14 tool calls, focus on completing and verifying the task after 26, and block further calls after 44. The reminders are appended to tool results, preserving the prompt prefix for cache reuse. Alongside these dynamic mechanisms, the evolved harnesses also reduce static **context redundancy**. On Terminal-Bench 2, for instance, tool-output compression like RTK (Szymkowiak, 2026) folds repeated logs and condenses long outputs, while streamlined developer instructions reduce redundant input. This combination curbs recurring prompt overhead and limits the context growth caused by verbose tool feedback, thereby reducing execution cost and latency. Implementation details of these harnesses are in Appendix D.1.

5.4 EVOLUTION DYNAMICS

Figure 4 tracks test-set performance and cost across epochs. Across all three benchmarks, HARNESSPRO advances the performance-cost Pareto frontier through recursive harness self-improvement. On Terminal-Bench 2, each round of evolution ***improves the retained Pareto frontier***, yielding higher pass rates at lower costs. For example, the pass rate of the heavy mode rises from **74.6%** in epoch 1 to a mean of **80.8%** in epoch 3, while mean cost per task falls from **$0.91** to **$0.72**. The lite and medium modes also maintain their pass rates from epoch 2 to 3, while their mean costs fall from $0.61 to $0.43 and from $0.68 to $0.56, respectively. The necessity of recursive harness self-improvement is even clearer on SWE-Bench Pro, where a single update may raise pass rate ***at the expense of higher cost***. In the medium mode, pass rate rises from 70.0% in epoch 1 to 77.0% in epoch 2, but mean cost also rises from $1.57 to $1.66. In epoch 3, pass rate improves further to 79.0% as cost falls to $1.26 per task. On ProgramBench, the fourth epoch likewise lowers costs across all three modes while maintaining similar pass rates. These patterns highlight an advantage of **recursive** self-improvement over single-round harness evolution.

5.5 ABLATION STUDY



(a) Terminal-Bench 2 (b) SWE-Bench Pro (c) ProgramBench

Figure 4: Evolution dynamics of HARNESSPRO with GPT-5.5 and Codex. Dashed lines connect the harnesses retained in each epoch, and diamonds denote baselines. Intermediate-epoch test-set metrics are reported for visualization only and were not used for filtering or selection. Final-epoch points show means across repeated test runs.

Table 3 examines the contributions of main components of HARNESSPRO. We also report Cost-of-Pass (CoP) (Erol et al., 2026), the total execution cost divided by the number of successful tasks. Removing the planner reduces performance and **increases cost** in most experiments. For example, on SWE-Bench Pro, the pass rate of the medium mode drops from 79.0% to 73.0%, while mean cost rises from $1.26 to $1.42 per task. Removing the test synthesizer has a **larger impact**, with pass rates dropping by **3.4–10.0** percentage points across both benchmarks and all three modes. These performance drops highlight the value of our multi-agent framework. Beyond these component ablations, we also examine whether RHO can achieve similar gains by simply adopting individual strategies from our evolved harnesses. As shown in Table 3, adding either dynamic routing or budget hooks to RHO reduces cost, but can also **degrade** task performance. On Terminal-Bench 2, both variants retain RHO's 76.3% pass rate while lowering mean cost from $1.09 to $0.92; on SWE-Bench Pro, their pass rates all fall below RHO's 80.0%. Uniformly using low reasoning effort yields larger cost savings but also substantially degrades performance. Overall, all three RHO variants are Pareto-dominated by harnesses evolved by HARNESSPRO, which achieve higher pass rates at lower cost.

Table 3: Ablations with GPT-5.5 and Codex. In the RHO variants, + *dynamic routing* selects reasoning effort from task descriptions, + *budget hooks* append budget reminders to tool results and + *low reasoning effort* fixes effort at low.

Method	Terminal-Bench 2			SWE-Bench Pro		
	P(%) ↑	C($) ↓	CoP($) ↓	P(%) ↑	C($) ↓	CoP($) ↓
RHO	76.3	1.09	1.43	80.0	2.68	3.35
RHO + dynamic routing	76.3	0.92	1.21	76.0	2.13	2.80
RHO + budget hooks	76.3	0.92	1.21	75.0	1.75	2.33
RHO + low reasoning effort	62.7	0.45	0.72	59.0	1.08	1.83
HarnessPro-Lite	72.9	0.43	0.59	72.3	0.80	**1.11**
w/o planner	72.9	**0.41**	**0.56**	70.0	0.98	1.40
w/o test synthesizer	67.8	0.67	0.99	64.0	**0.78**	1.22
HarnessPro-Medium	76.3	0.56	0.73	79.0	1.26	1.59
w/o planner	74.6	0.61	0.82	73.0	1.42	1.95
w/o test synthesizer	72.9	0.82	1.12	69.0	1.37	1.99
HarnessPro-Heavy	**80.8**	0.72	0.89	**80.7**	1.97	2.44
w/o planner	79.7	1.04	1.30	79.0	2.12	2.68
w/o test synthesizer	74.6	0.88	1.18	72.0	1.62	2.25

6 CONCLUSION

We introduced HARNESSPRO, a label-free approach to recursive harness self-improvement that improves task performance while reducing execution cost. To this end, it coordinates trajectory diagnosis and harness modification through a multi-agent framework, with a dedicated planner proposing update plans and LLM-generated tests guiding candidate evaluation. Rather than producing a single harness, HARNESSPRO offers **cost-focused** (lite), **balanced** (medium), and **performance-focused** (heavy) alternatives to meet different deployment budgets. Experiments across three benchmarks show that HARNESSPRO can ***improve task performance*** while ***reducing execution cost***, with additional benefits in **latency**. Further analysis attributes these gains to the **synergy** of multiple mechanisms and shows sustained progress through recursive self-improvement. These findings show that performance and efficiency can improve together, making cost-aware evolution a promising direction for practical recursive self-improvement in harness optimization.

AI USE STATEMENT

Generative AI tools were used to assist with software implementation and the drafting and editing of this manuscript. Separately, LLMs are integral to the method studied in this work: they analyze execution trajectories, propose and implement harness updates, and synthesize tests during harness evolution. All AI-assisted code and text were manually reviewed, and all reported results were checked by the authors.

REPRODUCIBILITY STATEMENT

The full evolution procedure and optimization objective are described in Sections 3 and 4, with pseudocode in Algorithm 1. Section 5.1 specifies the benchmarks, model and agent versions, rollout counts, evolution epochs, selection hyperparameters, and evaluation metrics. The appendix provides the evolution instructions, token-pricing assumptions, and representative artifacts from the frozen evolved harnesses. We will release the implementation, complete run configurations, generated tests, task-split manifests, and evolved harnesses to support independent reproduction of our results.

REFERENCES

Mengzhuo Chen, Junjie Wang, Zhe Liu, Yawen Wang, Haiming Zheng, and Qing Wang. From failed trajectories to reliable LLM agents: Diagnosing and repairing harness flaws. *arXiv preprint arXiv:2606.06324*, 2026a. URL https://arxiv.org/abs/2606.06324.

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. URL https://arxiv.org/abs/2606.14249.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/61047.

Mehmet Hamza Erol, Batu El, Mirac Suzgun, Mert Yuksekgonul, and James Zou. Cost-of-pass: An economic framework for evaluating language models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=vC9S20zsgN.

Jingzhi Gong, Ruizhen Gu, Zhiwei Fei, Yazhuo Cao, Lukas Twist, Alina Geiger, Shuo Han, Dominik Sobania, Federica Sarro, and Jie M. Zhang. SkillMOO: Multi-objective optimization of agent skills for software engineering. *arXiv preprint arXiv:2604.09297*, 2026a. URL https://arxiv.org/abs/2604.09297.

Jingzhi Gong, Jie M. Zhang, Gunel Jahangirova, Dong Huang, Mohammad Reza Mousavi, and Mark Harman. COMPAS: Difficulty-aware joint search for optimizing code generation. *arXiv preprint arXiv:2608.04336*, 2026b. URL https://arxiv.org/abs/2608.04336.

Mikaela Grace, Jeremy Hadfield, Rodrigo Olivares, and Jiri De Jonghe. Demystifying evals for AI agents. Anthropic Engineering, January 2026. URL https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents. Published January 9, 2026.

Alex Kulesza and Ben Taskar. Determinantal point processes for machine learning. *Foundations and Trends in Machine Learning*, 5(2–3):123–286, 2012.

Songxin Lei, Kun Ouyang, Weilin Ruan, Yuqian Wu, Zhijiang Guo, Yushi Sun, and Fugee Tsung. MemoryCPT: An end-to-end agent memory framework for cost-performance trade-off. *arXiv preprint arXiv:2608.04843*, 2026. URL https://arxiv.org/abs/2608.04843.

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

Kevin Lin, Charlie Snell, Yu Wang, Charles Packer, Sarah Wooders, Ion Stoica, and Joseph E. Gonzalez. Sleep-time compute: Beyond inference scaling at test-time. *arXiv preprint arXiv:2504.13171*, 2025. URL https://arxiv.org/abs/2504.13171.

Tengxiao Liu, Zifeng Wang, Jin Miao, I-Hung Hsu, Jun Yan, Jiefeng Chen, Rujun Han, Fangyuan Xu, Yanfei Chen, Ke Jiang, Samira Daruki, Yi Liang, William Yang Wang, Tomas Pfister, and Chen-Yu Lee. Budget-aware tool use enables effective agent scaling. *arXiv preprint arXiv:2511.17006*, 2025. URL https://arxiv.org/abs/2511.17006.

R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 2004. doi: 10.1007/s00158-003-0368-6.

OpenAI. Agents API architecture. https://developers.openai.com/api/docs/guides/agents-api/architecture/, 2026. Accessed: 2026-09-12.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=jL7fwchScm.

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. URL https://arxiv.org/abs/2606.05922.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.eacl-long.333. URL https://aclanthology.org/2026.eacl-long.333/.

Patrick Szymkowiak. RTK: Rust Token Killer. GitHub repository, 2026. URL https://github.com/rtk-ai/rtk. Accessed September 20, 2026.

The Terminal-Bench Team. Terminal-Bench: A benchmark for AI agents in terminal environments, April 2025. URL https://github.com/laude-institute/terminal-bench.

Xuhui Wang, Ruoqi Shu, Chen Dan, Tianhua Xu, Mengxi Luo, Yanming Mai, and Bo Wan. FRAMES: Guarded and dual-objective skill evolution for agents in policy-governed enterprise workflows. *arXiv preprint arXiv:2608.01772*, 2026. URL https://arxiv.org/abs/2608.01772.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026a. URL https://arxiv.org/abs/2605.03546.

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

Asaf Yehudai, Lilach Eden, Alan Li, Guy Uziel, Yilun Zhao, Roy Bar-Haim, Arman Cohan, and Michal Shmueli-Scheuer. A survey on evaluation of LLM-based agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 26690–26714, 2026. doi: 10.18653/v1/2026.findings-acl.1330. URL https://aclanthology.org/2026.findings-acl.1330/.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Godel machine: Open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=43n76kz3F5.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2410.10762.

A Implementation Details

A.1 DPP-Based Coreset Selection

We select the training coreset once, before harness evolution. Following RHO (Pan et al., 2026), let $\mathcal{U} = \{1, \ldots, M\}$ index the candidate tasks. For each task x_i, we collect a trajectory under the initial harness. A language-model judge reads the task description and trajectory, then assigns a difficulty score $r_i \in [0, 10]$ and an abstract failure-mode fingerprint ϕ_i. The fingerprint omits task-specific names so that similarity reflects shared challenges rather than superficial project details.

Let u_i be the unit-normalized embedding of ϕ_i. We compute cosine similarity and construct a difficulty-weighted DPP kernel:

$$\begin{aligned} S_{ij} &= u_i^\top u_j, \qquad q_i = \max(r_i/10, \varepsilon), \\ w_i &= \left(q_i / \max_j q_j \right)^\alpha, \qquad \alpha = \frac{\theta}{2(1-\theta)}, \\ K &= \mathrm{diag}(w)\, S\, \mathrm{diag}(w). \end{aligned} \tag{10}$$

We use $\varepsilon = 0.1$ and $\theta = 0.7$, as in RHO. The difficulty weights favor challenging tasks, while the determinant of a subset's kernel penalizes redundant fingerprints. For a fixed coreset size k, the corresponding k-DPP assigns each k-element subset $Y \subseteq \mathcal{U}$ probability (Kulesza & Taskar, 2012)

$$\Pr_k(Y) = \frac{\det(K_Y)}{\sum_{Z \subseteq \mathcal{U},\, |Z|=k} \det(K_Z)}, \tag{11}$$

where K_Y is the principal submatrix indexed by Y. We do not sample from this distribution. Instead, we approximate its most probable subset by greedily adding the task that maximizes the resulting determinant:

$$\begin{aligned} i_s &= \arg\max_{i \in \mathcal{U} \setminus Y_s} \det(K_{Y_s \cup \{i\}}), \\ Y_{s+1} &= Y_s \cup \{i_s\}, \qquad Y_0 = \emptyset. \end{aligned} \tag{12}$$

We stop after $k = 10$ tasks and keep this coreset fixed across all evolution epochs. Neither official test results nor validation outcomes enter coreset selection.

A.2 Additional Implementation Details

Reasoning effort and mode weights. The default reasoning effort for GPT-5.5 is high on all three benchmarks. For GLM-5.2, the default is max on Terminal-Bench 2 and SWE-Bench Pro, but high on ProgramBench, where max tends to induce extensive reasoning and tool use, substantially lengthening execution. For the selection rule in Equation 9, the performance-cost weights (α_m, β_m) are $(1, 2)$ for lite, $(1, 1)$ for medium, and $(2, 1)$ for heavy. These weights remain fixed across epochs.

Generation settings. Our Codex configurations use the Responses API without explicitly setting temperature, top_p, or max_output_tokens. GPT-5.5 and Codex thus uses its default sampling behavior. The model supports up to 128,000 output tokens per response.[3] For GLM-5.2, Terminal-Bench 2 and SWE-Bench Pro use temperature 1.0, top-p 0.95, and a per-request `max_tokens` of 131,072. On ProgramBench, GLM-5.2 runtime uses temperature 1.0, top-p 1.0, and `max_tokens` of 32,000, with a 2,000-turn limit.

Solver and grading budgets. On Terminal-Bench 2, GPT-5.5 uses each task's native solver limit (600–3,600 seconds on the test set). On SWE-Bench Pro, the solver time limit is 1200 seconds for GPT-5.5 and 7,200 for GLM-5.2, and the grader limit is 3,600 seconds for both models. On ProgramBench, the solver budget is 7,200 seconds, separate from the 14,400 second outer limit for each grading branch. These wall-clock limits are distinct from harness-level tool-call budgets.

[3] `https://developers.openai.com/api/docs/models/gpt-5.5`

Table 4: Token prices used to compute execution costs.

Model	Uncached input	Cached input	Output
GPT-5.5 ($<$ 272K context)	5.00 \$/M	0.50 \$/M	30.00 \$/M
GLM-5.2	1.40 \$/M	0.26 \$/M	4.40 \$/M

Task environments. Terminal-Bench 2 and ProgramBench allocate 8 CPUs and 32 GiB of memory per solver. SWE-Bench Pro runs solvers in host workspaces without an explicit per-task CPU or memory limit. Candidate evaluation uses one rollout per validation task, and final testing is performed only after evolution. Infrastructure failures are excluded and retried to avoid inaccurate results.

Token pricing. Table 4 reports the public API prices used to convert token usage into execution cost. We use the short-context price tier for GPT-5.5 because all requests remain below 272K context tokens. Prices follow the official OpenAI[4] and Z.ai[5] pricing pages.

[4] https://developers.openai.com/api/docs/pricing
[5] https://docs.z.ai/guides/overview/pricing

B EVOLUTION INSTRUCTIONS

Listing 1 Cost-focused evolution instruction (I_1)

```
Prioritize reducing average execution cost while preserving task correctness and
reliable completion. Seek genuine efficiency gains, not lower bills obtained by
skipping required work, abandoning difficult tasks, or weakening validation.

Use the diagnostics, trajectories, and candidate implementations to distinguish
necessary work from avoidable expense. Look for recurring sources of redundant
exploration, repeated context, or low-value verification. Consider the total cost of
execution, including unsuccessful attempts and any overhead introduced by the proposed
change; fewer steps or shorter outputs alone do not establish an improvement.

Possible directions include adapting reasoning effort to individual public tasks,
managing context and tool feedback economically, and improving convergence or recovery.
Preserve the information and checks needed for dependable results. These are optional
search directions, not a checklist: derive any mechanism from the available evidence,
without assuming a routing rubric, fixed thresholds, profile taxonomy, or per-task rules.

Propose a focused, generalizable plan rather than accumulating optimizations. Explain
which observed expense each change addresses, why correctness should be preserved,
and what regression risks remain. Distinguish expected savings from measured results;
the fixed external evaluator, not the plan's claims, determines whether quality is
maintained and cost is reduced.
```

Listing 2 Balanced evolution instruction (I_2)

```
Seek a balanced improvement in task reliability and average execution cost. Treat
correctness and efficiency as joint objectives: do not pursue savings by neglecting
required behavior, or assume that additional computation necessarily improves quality.
A balanced plan should improve the trade-off, not simply average the other directions.

Use the diagnostics, trajectories, and candidate implementations to identify where
effort contributes to successful completion and where it is redundant. Compare peer
candidates to understand their strengths, weaknesses, and interactions, rather than
copying their mechanisms wholesale. Prefer changes that remove low-value work while
preserving or strengthening the reasoning and validation that matter.

Possible directions include task-adaptive reasoning effort, concise but recoverable
context, effective use of tool feedback, and verification or recovery guided by observed
uncertainty. These are optional hypotheses, not prescribed designs. Infer an appropriate
approach from the evidence; no routing rubric, fixed thresholds, profile names, or
per-task rules are supplied. Account for the overhead and complexity of combining
changes.

Propose a coherent, generalizable plan and explain its expected quality and cost effects
together. Make uncertain trade-offs and regression risks explicit, including why any
added work is worthwhile. Keep predicted benefits separate from established findings
and let the fixed external evaluator determine whether the joint improvement is real.
```

Listing 3 Performance-focused evolution instruction (I_3)

```
Prioritize improving task success while continuing to eliminate avoidable expense.
Focus on reliable completion and correct behavior, not more elaborate execution for
its own sake. Performance-focused evolution should invest effort where it can address
meaningful failures, rather than increase computation uniformly across tasks.

Use the diagnostics, trajectories, and candidate implementations to identify
consequential failure patterns and the limitations behind them. Examine both successful
and unsuccessful executions to distinguish useful reasoning or validation from activity
that merely adds cost. Draw on peer candidates when their behavior provides relevant
evidence, but do not assume that a more complex harness is a more capable one.

Possible directions include adapting reasoning effort to the public task, selective
deeper investigation or independent verification, and stronger context or recovery
management. These are optional hypotheses, not an implementation recipe. Derive the
actual design from evidence; no routing rubric, fixed thresholds, mandatory review
schedule, or per-task rules are supplied. Preserve efficient behavior where additional
work has no demonstrated purpose.

Propose a focused, generalizable plan that explains which failures it aims to prevent
and why the changes should help. If additional work is necessary, state the expected
quality benefit, cost implications, and remaining risks without presenting predictions
as measured gains. The fixed external evaluator determines whether the improvement
justifies the resulting quality-cost trade-off.
```

C Representative Multi-Agent Artifacts

C.1 Trajectory Diagnoses

The following condensed analyzer outputs summarize three independent rollouts per task for two Terminal-Bench 2 training tasks. Pass outcomes are determined by fixed LLM-generated tests, and the evidence pointers refer to event ranges in the recorded trajectories.

Caffe–CIFAR-10. This case illustrates a recurring completion failure: all three attempts spent their execution budget building Caffe or downloading data, without reaching the required training run. The diagnosis identifies both the shared resource bottleneck and the resulting wasted time.

Case 1 Caffe–CIFAR-10: incomplete training

```
Task: Build CPU-only Caffe and train CIFAR-10 for 500 iterations.
Outcome: None of the three rollouts produced the training log or model.
Problem: All three used high build parallelism and hit out-of-memory
  (OOM) failures during compilation. Lower-parallelism retries took time;
  one rollout completed the build but was delayed by a slow data download.
Performance impact: Training and accuracy verification were not reached.
Cost impact: Failed builds, retries, and download attempts consumed
  substantial execution time before the task ended.
Evidence:
  evidence/run-0-trace-0.jsonl:71-89
  evidence/run-1-trace-0.jsonl:90-111
  evidence/run-2-trace-0.jsonl:61-80
```

Mailman. In contrast, all three attempts to configure a Postfix/Mailman 3 mailing list passed the generated tests. Here the analyzer diagnoses avoidable overhead rather than a correctness failure: broad configuration and source inspection, followed by repeated checks with overlapping coverage.

Case 2 Mailman: successful but costly execution

```
Task: Configure a mailing list with join, leave, and post delivery.
Outcome: All three rollouts passed the generated tests.
Problem: Broad configuration and package-source dumps expanded the
  traces. Agents also reran /app/eval.py after passing evaluations and
  manual smoke checks had already covered the required behavior.
Performance impact: No task failure was observed; useful end-to-end
  checks should be preserved.
Cost impact: Large tool outputs and redundant evaluations increased
  context processing and execution time without clear new coverage.
Evidence:
  evidence/run-0-trace-0.jsonl:36-69,149-153
  evidence/run-1-trace-0.jsonl:158-173
  evidence/run-2-trace-0.jsonl:102-122,167-170
```

C.2 Evolution Plans

The following Terminal-Bench 2 plans illustrate three performance-cost priorities. The medium and heavy plans start from the same initial harness and trajectory diagnoses. A later lite proposal for an edited descendant responds to a router that often chose high or xhigh. The listings retain the Planner's wording and structured fields, with line breaks added for readability. Some passages are omitted to save space.

Case 3 Medium plan: introduce task-adaptive routing (excerpt)

```
objective: Improve the empty baseline harness by adding general solver
  guidance and lightweight execution support that preserves task-relevant
  verification while cutting repeated broad exploration, noisy command
  output, and overconfident local-success conclusions.
changes:
  README.md: Add concise solver guidance: start from the public task
    contract and required artifacts;
    ...  (content is excerpted to save space) ...
  runtime.json: Add a small runtime configuration referencing
    router_prompt.txt and setting tool_output_token_limit to a moderate
    value such as 12000, without adding a review session or replacing
    the model instructions.
  router_prompt.txt: Add a general cost-aware routing prompt that
    chooses the lowest reasoning_effort likely to complete the public
    task: low for bounded recovery or simple one-artifact tasks, medium
    for routine code/configuration with clear tests, high for multi-file
    implementations, builds, interpreters, protocol/integration work,
    numerical algorithms, or weak local oracles, and xhigh only for
    unusually open-ended tasks where failure would likely come from
    deep reasoning rather than environment setup. Require JSON with
    reasoning_effort and a short rationale.
  tools/guarded-run: Add an executable POSIX shell helper for long or
    noisy commands. It should run a command under a timeout, save full
    stdout/stderr to a log file, and print only exit status, elapsed
    time, log path, and bounded head/tail excerpts.
    ...  (content is excerpted to save space) ...
  tools/quick-probe: Add an executable POSIX shell helper that prints a
    compact one-time environment snapshot: cwd, top-level files, whether
    the directory is a git repo, and availability of common tools.
    ...  (content is excerpted to save space) ...
quality_impact: Established findings show failures and regressions came
  less from insufficient raw deliberation and more from shallow or
  misread validation, unfinished long-running workflows, environment
  assumptions, and task-integrity problems.
  ...  (content is excerpted to save space) ...
cost_impact:
  ...  (content is excerpted to save space) ...
  The router adds a small high-effort classification call, but should
  repay that overhead on simpler tasks by avoiding default high-effort
  solving while preserving higher effort for tasks where it is likely
  useful.
  ...  (content is excerpted to save space) ...
risks: The router may under-allocate effort for deceptively simple tasks,
  and the routing call can be net-negative on very small tasks. Output
  caps and guarded summaries can hide important diagnostics unless the
  solver follows the log-path recovery guidance.
  ...  (content is excerpted to save space) ...
```

Case 4 Heavy plan: verify task artifacts (excerpt)

```
objective: Add a small, general harness that steers solvers toward
  public-contract verification, bounded exploration, and cheaper command
  output while preserving flexibility for varied tasks.
  ...  (content is excerpted to save space) ...
changes:
  README.md: Create concise solver guidance: derive an acceptance
    checklist from the public task first; use only public task files and
    legitimate upstream documentation, never benchmark
    answers/graders/solutions; inspect files in narrow slices; check
    tool availability once; prefer Python fallback inspection in minimal
    containers; avoid dumping large logs, matrices, generated files, or
    whole source files; complete all running commands before final;
    verify required artifacts with semantic/end-to-end checks rather
    than only syntax, existence, or nonblank outputs.
  runtime.json: Create with a conservative tool_output_token_limit, for
    example 16000, to cap accidental giant outputs from package managers,
    build logs, tracebacks, git usage text, and file dumps while still
    leaving room for meaningful compiler or test failures.
  tools/run_logged.py: Add an executable Python helper with a shebang
    that runs a command with an optional timeout, writes the full
    stdout/stderr stream to a log file, and prints only command, exit
    code, elapsed time, and the final bounded tail. README should
    recommend it for apt/build/train/long-test commands so expensive
    workflows can preserve diagnostics without flooding context.
  tools/probe.py: Add an executable Python helper with a shebang for
    common low-dependency inspections: command-exists checks, limited
    head/tail display, hex/ascii byte previews, JSON validation, CSV
    column/row summaries, file size/hash, and simple BMP/PNG metadata
    where possible. This replaces repeated assumptions about
    unavailable utilities such as file, sqlite3, xxd, strings, and git.
quality_impact: Expected to improve reliability by making solvers
  explicitly map public requirements to final artifacts, require
  semantic verification before success claims, and avoid continuing
  from misleading checks such as command-not-found text hidden inside
  successful pipelines.
  ...  (content is excerpted to save space) ...
cost_impact: Expected to reduce avoidable token and latency costs from
  repeated broad file reads, package/build log dumps, missing-tool
  probes, noisy git failures, and redundant post-success inspection.
  ...  (content is excerpted to save space) ...
  There is no added second solver session or uniform extra reasoning
  pass.
risks: Guidance-only changes depend on solver compliance, and an output
  cap can hide useful detail if commands are not logged through the
  helper. The helper scripts must remain dependency-free and generic.
  More rigorous semantic verification can add some task-local execution
  cost, so the README should emphasize checks tied to public
  requirements rather than broad extra exploration.
```

Case 5 Lite plan: make routing more cost-sensitive (excerpt)

```
objective: Reduce solver and exploratory cost by correcting a router that
  was observed to choose high or xhigh for most sampled tasks, while
  retaining task-specific reasoning-effort selection and public-contract
  correctness.
changes:
  runtime.json: Keep router_prompt connected so each task still receives
    a dynamically selected reasoning_effort.
    ...  (content is excerpted to save space) ...
    Reduce tool_output_token_limit from 12000 to about 8000 to limit recurring
    transcript growth without removing useful diagnostics.
  router_prompt.txt: Rewrite the rubric to counter the observed
    high/xhigh bias: make medium the preferred starting point for bounded
    tasks with a plausible implementation path and practical checks,
    even when they mention several files, a build, data processing, or
    an unfamiliar domain. Choose low more readily when the requested
    change is concrete and its result can be checked directly. Escalate
    to high only for substantial unresolved reasoning or weak
    verification, and to xhigh only for exceptional cases where deeper
    reasoning is likely to change the solution.
    ...  (content is excerpted to save space) ...
  README.md: Revise guidance to reference helpers by absolute harness
    path, such as $HARNESSPRO_HARNESS/tools/guarded-run and
    $HARNESSPRO_HARNESS/tools/quick-probe, because relative tools/ paths
    failed from task working directories.
    ...  (content is excerpted to save space) ...
    stop extra stress testing
    once public-contract coverage plus one meaningful semantic check
    has passed.
  tools/guarded-run: Tighten defaults and output shaping: reduce the
    default excerpt size, add a per-line character cap while preserving
    the full log path, and keep timeout/status/elapsed reporting.
    ...  (content is excerpted to save space) ...
  tools/quick-probe: Extend the probe output to include
    HARNESSPRO_HARNESS, the absolute helper paths, and a compact
    availability table only.
    ...  (content is excerpted to save space) ...
quality_impact: Dynamic routing remains available for tasks that
  genuinely need high or xhigh reasoning, while the solver guidance
  strengthens public-contract and semantic validation. Favoring medium
  and, for directly checkable work, low should not sacrifice quality
  solely to save tokens; the router must still escalate when the
  approach or verification is genuinely uncertain.
cost_impact: The observed router usually chose high or xhigh despite
  adding its own high-reasoning turn, so it incurred fixed routing cost
  without much solver-effort reduction.
  ...  (content is excerpted to save space) ...
  Savings remain a hypothesis until measured on task rollouts.
risks: The observed high/xhigh choices may include genuinely difficult
  tasks; shifting toward medium or low could hurt quality if public
  descriptions hide hard reasoning or weak oracles.
  ...  (content is excerpted to save space) ...
  Keep high and xhigh available for substantial uncertainty, and
  compare per-task outcomes and costs before treating the revised
  rubric as an improvement.
```

C.3 TEST SYNTHESIZER CONFIGURATION

Our test-synthesis agent uses an instruction and skill together with an MCP-based validation tool. The first two listings present selected passages from the deployed text, with output-schema fields condensed; the third summarizes the tool interface. Together, they show how the agent designs tests from public evidence, checks them against a frozen artifact, and reviews the suite without access to official tests or labels.

Listing 4 Test-synthesis instruction (excerpt)

```
Use $empty-test-authoring. In this one Codex invocation, write ten distinct
executable cases, validate them on the frozen Empty product, inspect the results,
and self-review the final suite. No separate reviewer is part of this run.

## Evidence boundary
Read /work/public-task.json, relevant /public_inputs, and the read-only
/empty_product. The task description and public files define the contract;
Empty is only an implementation and execution-diagnostic context, never an
answer key. Public sample programs and evaluation helpers may be used.
...
Do not seek or use the target's hidden official tests, grader, solution,
grades, or earlier target suites, locally or online. Do not modify Empty.

## Test design
Inventory the public requirements and the actual input and output interfaces
before testing Empty. Cover distinct documented behaviors, valid alternatives,
boundaries, and justified combinations; do not invent extra obligations to
fill the fixed count.
...
For each central requirement, invoke the real delivered product and assert a
meaningful relation between its output and independent public evidence.
A file existing, successful process, shape, nonempty value, or plausible range
alone does not establish the required behavior.
...
Choose the strongest justified oracle available: compatible public reference,
known answer, necessary content property, or discriminating metamorphic
relation. Never derive expected constants, hashes, tolerated values, or
execution paths from Empty.

## Execute and self-review
Write the complete suite, then call validate_generated_tests on it. A tool
result with status=ok, passed=false is a valid execution with possible product
failures; making Empty pass is not the objective. infra_error is inconclusive:
inspect the traceback and public evidence to distinguish test/setup defects
from product exceptions under valid invocation. Repair only demonstrated test
defects, preserving the public requirement and legitimate Empty failures. Do
not skip cases, catch all exceptions, or tune expectations toward Empty.
After every executable edit, rerun the tool; the final executable files must
match a completed execution without infrastructure error. Run tests serially.
Self-review every case, expected value, and shared helper after the final run.
...
## Output contract
Write UTF-8 files under /work/output, at most 1 MiB per file and 4 MiB total:
- test_generated.py: exactly test_case_0001 through test_case_0010 as
  unittest.TestCase methods; no skipped or unconditional padding cases.
- case_manifest.json: expected_count: 10; each case has a meaningful
  description and kind behavioral, format, or unavailable_oracle.
- coverage.json: public evidence, requirements mapped to case IDs, checks
  performed, what was not checked, and limitations.
- repair-log.json: demonstrated defects, public evidence, edits, and rerun
```

Listing 4 Test-synthesis instruction (excerpt) (continued)

```
  outcomes; use [] if no repair was needed.
- self-review.json: per-case assessment, reason, public evidence, suggested
  repair, and any essential or supplemental coverage gaps.
```

Listing 5 Empty-test-authoring skill (excerpt)

```
Read /work/public-task.json and relevant /public_inputs first. Inspect
/empty_product only to understand the frozen candidate's interfaces or to
diagnose execution. Its files correspond to original paths such as /app, but
the snapshot is not an oracle. The available public dependencies and fixture
mounts are listed in validation_environment in the task JSON. Disjoint
examples are read-only under /examples.

Write the suite in /work/output. Invoke the frozen_empty MCP tool
validate_generated_tests after writing executable tests and after each edit;
the tool executes the current suite in a fresh Empty container and returns
only synthetic diagnostics. If MCP discovery fails, send a JSON-RPC tools/call
request to python3 /opt/harnesspro/empty-tools/tools/mcp_server.py on stdin.
Keep tests and validation calls serial. The model has no Docker socket or
official grader.

Interpret status=ok, passed=false as a completed execution, not a request
to make Empty pass. Treat infra_error as unresolved until its cause is
identified. Repair only test defects supported by public evidence, record
the change in repair-log.json, and rerun the changed suite. Preserve valid
product failures and report any unresolved oracle or setup problem. Review
the final test design in the same invocation; do not launch another reviewer.

Public reference tools, if needed, are available in the authoring image;
version manifests are under /opt/harnesspro/reference-tools. Check that a
reference is compatible with the public runtime before using exact outputs.
The main instruction defines the ten-case output and review schema.
```

Listing 6 Validation tool interface and response (summary)

```
Tool: frozen_empty.validate_generated_tests
MCP discovery: tools/list advertises validate_generated_tests
Arguments: {} (no task-specific parameters; extra properties rejected)
Input: the current test suite in /work/output
Request: a validate_generated_tests operation sent through /empty-tool-rpc
Execution: a fresh container with the frozen Empty product
Response: JSON diagnostics from the suite execution, not official tests,
  grader feedback, or labels
Completed execution: status == ok, whether passed is true or false
Infrastructure failure: status == infra_error; MCP isError is true
Timeout: 7,200 seconds; an absent response raises an error rather than
  inventing a validation result
```

D Evolved Harness Artifacts

This appendix presents the evolved harnesses for all three benchmarks: the medium configurations for Terminal-Bench 2, SWE-Bench Pro, and ProgramBench, plus the differences with the lite and heavy modes. Long lines wrap, JSON whitespace is normalized, and closing Markdown inline-code backticks are typeset as apostrophes where needed. The bundled frozen files retain their original bytes. The directory listings show the evolved harness files.

D.1 TERMINAL-BENCH 2

Listing 7 shows the runtime files of the medium-mode harness. The evaluator supplies the rubric to a separate profile classifier, reads the policy settings, and generates a Codex configuration that loads the instruction file and registers the hook.

Listing 7 Terminal-Bench 2 medium-mode harness runtime files

```
runtime/
    model_instructions.md
    profile_rubric.md
    policy.json
    hook.py
```

D.1.1 MEDIUM MODE

The medium mode uses GPT-5.5 throughout, with five execution profiles spanning low, medium, and high reasoning effort. The listings below present its instructions, routing rubric, runtime hooks, and policy settings.

Listing 8 Medium: `runtime/model_instructions.md`

```
You are Codex, a non-interactive software agent. Work autonomously until the
user's task is complete. Follow the task's integrity, safety, path, and output
requirements exactly. Use the available tools to inspect, edit, execute, and
verify; batch independent checks and avoid repeating unchanged reads. Preserve
unrelated files and avoid destructive operations outside the requested scope.
Do not replace a requested implementation or service with a mock, proxy, or
look-alike unless the task explicitly permits that substitution.
Do not spawn other agents or ask the user to perform routine steps. Keep
progress messages brief. Before finishing, leave every requested artifact at
its exact path and run the strongest allowed task-shaped verification. Report
the outcome concisely, including any genuine remaining uncertainty.
```

Listing 9 Medium: `policy.json`: `session_context`

```
Harness guidance is already loaded; do not inventory harness/. Extract exact artifacts,
paths, behavior, integrity restrictions, public interfaces, and final state from
task/prompt.md. Use its host-to-container pattern. For black-box work use only public
interfaces and implement general behavior. When the task names an API, wrapper, package,
revision, runtime, or service, exercise that named interface first and do not silently
replace it with a lower-level dependency, mock, proxy, or look-alike behavior; inspect
the public signature or implementation after an interface error. A source-only or
single-artifact deliverable should not acquire undeclared runtime helpers. Start with
the smallest discriminating probe and batch independent checks. Bound downloads,
external interaction, and potentially blocking commands well below the task watchdog;
reserve time for final repair and verification. Validate requested behavior, not only
superficial readiness signals. Run mutating workflow checks on disposable fixtures when
possible; if the user is meant to perform the same workflow later, do not leave test
data or state that collides with its documented preconditions. Preserve the real
requested implementation and durable service state while removing only temporary
verification artifacts. For quantitative work, identify units, variables, invariants,
and the comparison objective before optimizing; exercise every publicly stated
correctness dimension using a direct reference or invariant when one can be constructed.
Keep temporary verification artifacts separate from final deliverables. Before
finishing, compare exact contents and state with every explicit requirement. Avoid
narration and unchanged rereads; do not skip necessary validation.
```

Listing 10 Medium: `runtime/profile_rubric.md`

```
Choose exactly one profile from observable task characteristics. Optimize
expected success per dollar. Never use historical outcomes remembered from prior
runs or evaluations.

Profiles:

- 'compact': low reasoning; converge near 4 tool calls and stop near 8.
- 'lean': low reasoning; converge near 10 calls and stop near 16.
- 'standard': medium reasoning; converge near 26 calls and stop near 38.
- 'deep': high reasoning; converge near 30 calls and stop near 48.
- 'research': high reasoning with about 40 calls for source reconstruction.

Evidence rules

- Use only validation, interaction, dependencies, artifacts, and constraints
  explicitly described in the task text. Never assume hidden-test behavior,
  previous attempts, tool output, runtime results, or historical scores.
- An undescribed validator has unknown quality, not weak quality. Unknown decisive
  evidence defaults to 'standard'; missing information alone cannot justify
  another profile.
- A strong oracle must be explicitly local and repeatable and provide localized
  failure, quantitative error, candidate ranking, deterministic simulation, or a
  directly inspectable target. Compilation, launch, an open port, one end-to-end
  request, or whole-task pass/fail is weak only when the text explicitly limits
  feedback to that undifferentiated signal.
- Count distinct evidence-resolvable hypotheses, not files, prose, commands, or
  retries. Extra iterations are valuable only when described probes can
  distinguish hypotheses and change the plan.
- Include the stated watchdog and expected probe cost in the value of
  iteration. A precise oracle is not useful enough for 'deep' when only a few
  full evaluations can fit and high reasoning does not create more evidence.

Apply the following gates in order and stop at the first match.

Gate 1: 'compact'

Choose 'compact' only when low marginal value is explicit and no faithful
decomposable oracle is described or structurally possible:

1. Long serial graphical, remote, virtualized, or manual interaction whose
   correctness depends on visual or hidden state.
2. Unavailable accounts, devices, services, credentials, or changing external
   state without a faithful local substitute.
3. Broad hidden or effectively unbounded behavior plus exact preservation of
   unrelated details, with explicitly end-to-end accept/reject feedback.
4. One unchanged artifact must satisfy independent grammars, parsers, runtimes,
   or environments, while checks expose only a joint accept/reject objective.
5. Bulk media or visual interpretation with no described numerical comparator
   and correctness resting on visual judgment.

Difficulty, visualization, specialization, or virtualization alone is not enough.
A small fixed visual artifact, scripted headless setup with direct checks, or
local scorer that distinguishes candidates should continue to later gates.
File-existence or format checks do not make long visual/media search a strong
oracle when the decisive event must still be found by inspecting many frames.

Gate 2: 'research'

Choose 'research' only when the primary deliverable is factual reconstruction
```

Listing 10 Medium: `runtime/profile_rubric.md` (continued)

```
requiring multiple accessible sources, version/date alignment, provenance, or a
historical snapshot. A multi-record date/version ranking qualifies; one lookup or
research used only to build an artifact does not. If decisive sources are
explicitly inaccessible, choose 'compact'.

Gate 3: 'deep'

Choose 'deep' only when a strong local oracle is explicitly described or can be
constructed from an exact deterministic reference, transformation, simulator, or
runtime supplied by the task, and at least one shape applies:

1. Latent structure or parameter recovery plus exact compatibility or a
   quantitative objective.
2. Correctness spans several interacting components, stages, or states and can be
   checked through exact invariants, reference behavior, or round trips.
3. Damaged structured-state recovery requiring competing format, ordering, or
   transformation hypotheses.
4. Constrained synthesis, compression, size reduction, or candidate search with
   coupled hard constraints and an executable scorer.
5. Nontrivial recovery, inference, or attack derived from many observations with
   exact candidate validation.
6. An inverse problem with a deterministic simulator or quantitative objective.
7. Candidate construction against a deterministic transformation or runtime,
   with multiple materially different mechanisms that can be executed and observed.
8. A severely constrained reimplementation of deterministic reference behavior,
   where exact candidate outputs can be compared locally despite language, size,
   compatibility, or dependency restrictions.
9. One source artifact must implement behavior while remaining valid under two
   or more independent grammars, compilers, or runtimes, each of which can be
   exercised separately for localized feedback.

Do not choose 'deep' for terminology, long specifications, ordinary security or
reverse engineering, or hidden tests alone. Ordinary performance tuning and
ordinary state propagation through services, hooks, or deployments are
'standard'. A binary validator alone is insufficient, but a deterministic local
transformation plus executable behavior can support meaningful candidate
iteration even when the final verdict is binary. Prefer 'standard' when each
candidate evaluation is described as expensive enough to consume a material
part of the watchdog and the work is mainly bounded tuning against one supplied
evaluator. Modifying an existing artifact against one supplied evaluator remains
'standard' merely because it combines a quantitative objective with preservation
constraints; 'deep' additionally requires latent recovery, combinatorial search,
or several materially different candidate mechanisms. 'deep' requires room for
several evidence-changing iterations.

Gate 4: 'lean'

Choose 'lean' only when all are explicit:

1. Bounded, mostly deterministic work.
2. At most two meaningful uncertainties.
3. A credible plan before broad exploration.
4. A direct local check covering nearly all decisive behavior.
5. About eight or fewer evidence-producing calls for inspection, implementation,
   and final validation.

Typical cases are a localized transformation; enumerable mechanical cleanup with
a deterministic final scan; a simple build or single-service setup with one main
uncertainty and a direct test; or bounded numerical work with direct value, shape,
```

Listing 10 Medium: `runtime/profile_rubric.md` (continued)

```
or residual validation.

Do not choose 'lean' when several coupled uncertainties require iterative
candidate comparison, or when recovering latent state requires materially
different hypotheses. Apply the 'deep' gate first when a constructible exact or
quantitative reference makes those iterations useful. Also do not choose 'lean'
for broad cleanup where unaffected content must be preserved exactly and the
set of true matches is not explicitly bounded; false-positive scope is a
separate uncertainty even when a final scan is deterministic.

Use 'standard' when coordinating several schemas, formats, subsystems,
integration boundaries, correctness/performance constraints, or hidden edge-case
families. Also use it for multi-service state propagation, broad security
properties, builds combining source provenance with feature exclusion and
multiple runtime modes, corpus-wide cleanup with exact preservation of unaffected
content, or unstructured recovery with several evidence paths but only final
candidate checks.

Gate 5: 'standard'

Choose 'standard' for everything else, including unknown oracle quality. It is
the default for ordinary implementation, debugging, data processing, service
setup, bounded numerical work, and analysis with several actionable
uncertainties, and should normally be the plurality.
```

Listing 11 reproduces the budget-denial branch of `_pre_tool_use` and the complete functions that produce stage reminders and compact text outputs. Dispatch, storage, audit logging, and other helper routines are omitted from the listing but retained in the accompanying full files.

Listing 11 Medium: `runtime/hook.py` (excerpts)

```
# Excerpt from _pre_tool_use: counter, budget, and denial branch
    tool_name = str(payload.get("tool_name") or "")
    state = _increment_tool_count(control_state)
    count = int(state["tool_count"])
    finalize = int(selection.get("finalize_after_tools") or 0)
    if finalize and count > finalize:
        _audit(
            "pre_tool_use",
            cwd=str(cwd),
            tool_name=tool_name,
            tool_count=count,
            decision="deny",
        )
        print(
            json.dumps(
                {
                    "hookSpecificOutput": {
                        "hookEventName": "PreToolUse",
                        "permissionDecision": "deny",
                        "permissionDecisionReason": (
                            "The task tool budget is exhausted. Preserve the current
files, "
                            "stop calling tools, and give the best final answer from
existing evidence."
                        ),
                    }
                }
```

Listing 11 Medium: `runtime/hook.py` (excerpts) (continued)

```
            )
        )
        return 0

def _stage_context(count: int, selection: dict[str, Any]) -> str:
    economy = int(selection.get("economy_after_tools") or 0)
    strict = int(selection.get("strict_after_tools") or 0)
    reminder = int(selection.get("strict_reminder_every") or 8)
    if count == economy:
        return (
            "Economy phase starts now. Batch independent ls/rg/read/test checks,
suppress "
            "process narration, and do not repeat unchanged checks. Do not trade away
the "
            "behavioral validation or exact final-state checks required by the task."
        )
    if count >= strict and (count == strict or (count - strict) % reminder == 0):
        return (
            "Convergence phase: stop broad exploration. Only inspect evidence directly
tied to the "
            "current failure or deliverable. If a concrete failure, unexplained
instability, or "
            "mutated deliverable remains, repair it rather than finalizing because the
phase changed. "
            "Avoid irreversible cleanup or broad state changes not required by the task,
and keep "
            "temporary verification artifacts separate from final deliverables. "
            "Then recheck exact paths, interfaces, behavior, and final state once and
finish."
        )
    return ""

def _compact_text(text: str, *, head_chars: int, tail_chars: int) -> str:
    lines = text.splitlines()
    deduped: list[str] = []
    index = 0
    while index < len(lines):
        next_index = index + 1
        while next_index < len(lines) and lines[next_index] == lines[index]:
            next_index += 1
        repeats = next_index - index
        deduped.append(lines[index])
        if repeats > 2:
            deduped.append(f"[previous line repeated {repeats - 1} more times]")
        elif repeats == 2:
            deduped.append(lines[index])
        index = next_index
    compact = "\n".join(deduped)
    if len(compact) <= head_chars + tail_chars:
        return compact
    omitted = len(compact) - head_chars - tail_chars
    return (
        compact[:head_chars]
        + f"\n... [{omitted} characters omitted; use the raw path for a targeted range]
...\n"
        + compact[-tail_chars:]
    )
```

The policy below specifies the execution profiles, routing rules, and output-retention settings. CLI feature settings remain in the full files. `history_token_limit` is a per-output history limit, not the model context-window size. The rubric's "stop near" figures are approximate guidance for profile choice; the policy's `finalize_after_tools` values set the enforced limits. Tool thresholds count attempted calls.

Listing 12 Medium: `policy.json` (selected fields)


```
{
  "default_profile": "standard",
  "tool_output": {
    "history_token_limit": 900,
    "max_chars": 4000,
    "head_chars": 2400,
    "tail_chars": 900,
    "rtk_enabled": true
  },
  "profiles": {
    "compact": {
      "reasoning_effort": "low",
      "economy_after_tools": 2,
      "strict_after_tools": 4,
      "finalize_after_tools": 14,
      "strict_reminder_every": 2
    },
    "lean": {
      "reasoning_effort": "low",
      "economy_after_tools": 6,
      "strict_after_tools": 10,
      "finalize_after_tools": 22,
      "strict_reminder_every": 4
    },
    "standard": {
      "reasoning_effort": "medium",
      "economy_after_tools": 14,
      "strict_after_tools": 26,
      "finalize_after_tools": 44,
      "strict_reminder_every": 8
    },
    "deep": {
      "reasoning_effort": "high",
      "economy_after_tools": 16,
      "strict_after_tools": 30,
      "finalize_after_tools": 54,
      "strict_reminder_every": 8
    },
    "research": {
      "reasoning_effort": "high",
      "economy_after_tools": 8,
      "strict_after_tools": 20,
      "finalize_after_tools": 46,
      "strict_reminder_every": 6
    }
  },
  "classifier": {
    "type": "model",
    "model": "gpt-5.5",
    "reasoning_effort": "medium",
    "transport": "responses",
    "fallback_profile": "standard",
```

Listing 12 Medium: `policy.json` (selected fields) (continued)

```
    "rubric_file": "profile_rubric.md",
    "timeout_s": 120,
    "max_attempts": 3,
    "max_output_tokens": 2048,
    "samples": 1,
    "max_task_chars": 24000
  },
  "confidence_overrides": [],
  "rules": []
}
```

D.1.2 HEAVY MODE (V1)

The heavy mode uses GPT-5.5 with low, medium, or high reasoning effort, depending on the selected profile, and allows larger tool budgets than medium. Its frozen evaluation snapshot has the same instruction and hook files as medium. Only the routing rubric and the policy's `profiles` and `session_context` fields differ. The rubric diff below is relative to Listing 10; the changed policy fields follow.

Listing 13 Heavy: `profile_rubric.md` (diff vs. medium)

```
--- medium/runtime/profile_rubric.md
+++ heavy/runtime/profile_rubric.md
@@ -7,5 +7,5 @@
-- 'compact': low reasoning; converge near 4 tool calls and stop near 8.
-- 'lean': low reasoning; converge near 10 calls and stop near 16.
-- 'standard': medium reasoning; converge near 26 calls and stop near 38.
-- 'deep': high reasoning; converge near 30 calls and stop near 48.
-- 'research': high reasoning with about 40 calls for source reconstruction.
+- 'compact': low reasoning; converge near 8 tool calls and stop near 14.
+- 'lean': medium reasoning; converge near 22 calls and stop near 36.
+- 'standard': high reasoning; converge near 38 calls and stop near 60.
+- 'deep': high reasoning; converge near 42 calls and stop near 68.
+- 'research': high reasoning with up to about 60 calls for source reconstruction.
@@ -113 +113 @@
-5. About eight or fewer evidence-producing calls for inspection, implementation,
+5. About twelve or fewer evidence-producing calls for inspection, implementation,
```

Listing 14 Heavy: `policy.json`: `session_context`

```
Harness guidance is already loaded; do not inventory harness/. Extract exact artifacts,
paths, behavior, integrity restrictions, public interfaces, and final state from
task/prompt.md. Use its host-to-container pattern. For black-box work use only public
interfaces and implement general behavior. When the task names an API, wrapper, package,
revision, runtime, or service, exercise that named interface first and do not silently
replace it with a lower-level dependency, mock, proxy, or look-alike behavior; inspect
the public signature or implementation after an interface error. A source-only or
single-artifact deliverable should not acquire undeclared runtime helpers. Start with
the smallest discriminating probe and batch independent checks. Bound downloads,
external interaction, and potentially blocking commands well below the task watchdog;
reserve time for final repair and verification. Validate requested behavior, not only
superficial readiness signals. For multi-stage services and persistent state, test from
a fresh external entry point and check repeated-operation behavior, defaults,
permissions, and final ownership instead of trusting internal readiness. For numerical,
transformation, or recovery work, validate against an independent reference, invariant,
round trip, or residual rather than only reusing the implementation's own assumptions.
Run mutating workflow checks on disposable fixtures when possible; if the user is meant
```

Listing 14 Heavy: `policy.json: session_context` (continued)

```
to perform the same workflow later, do not leave test data or state that collides with
its documented preconditions. Preserve the real requested implementation and durable
service state while removing only temporary verification artifacts. Keep temporary
verification artifacts separate from final deliverables. Before finishing, reread the
public task once, inspect the actual final artifact or durable state, and compare every
explicit requirement against independent evidence. Avoid narration and unchanged
rereads; do not skip necessary validation.
```

Listing 15 Heavy: `policy.json: profiles`

```
{
  "profiles": {
    "compact": {
      "reasoning_effort": "low",
      "economy_after_tools": 4,
      "strict_after_tools": 8,
      "finalize_after_tools": 14,
      "strict_reminder_every": 3
    },
    "lean": {
      "reasoning_effort": "medium",
      "economy_after_tools": 10,
      "strict_after_tools": 22,
      "finalize_after_tools": 36,
      "strict_reminder_every": 4
    },
    "standard": {
      "reasoning_effort": "high",
      "economy_after_tools": 18,
      "strict_after_tools": 38,
      "finalize_after_tools": 60,
      "strict_reminder_every": 8
    },
    "deep": {
      "reasoning_effort": "high",
      "economy_after_tools": 20,
      "strict_after_tools": 42,
      "finalize_after_tools": 68,
      "strict_reminder_every": 8
    },
    "research": {
      "reasoning_effort": "high",
      "economy_after_tools": 16,
      "strict_after_tools": 36,
      "finalize_after_tools": 60,
      "strict_reminder_every": 8
    }
  }
}
```

D.2 SWE-Bench Pro

The medium mode harness uses GPT-5.5 and Codex. GPT-5.5 selects one of five reasoning/tool-budget profiles from the public task description. Its repository-repair workflow first extracts the contract, then checks the implementation and available references, makes a focused patch, and verifies task-shaped behavior. Listing 16 shows the stored harness files. The evaluator passes the rubric to a separate classifier and configures Codex to load the instructions and register the hook. The verification helper and checklist remain available to the solver.

Listing 16 SWE-Bench Pro medium-mode harness files

```
README.md
runtime/
    model_instructions.md
    profile_rubric.md
    policy.json
    hook.py
bin/
    repair-verify
checklists/
    contract-verification.md
```

Listing 17 SWE-Bench Pro medium: `model_instructions.md`

```
You are Codex, a non-interactive software agent. Work autonomously until the
user's task is complete. Follow the task's integrity, safety, path, and output
requirements exactly. Use the available tools to inspect, edit, execute, and
verify; batch independent checks and avoid repeating unchanged reads. Preserve
unrelated files and avoid destructive operations outside the requested scope.
Do not replace a requested implementation or service with a mock, proxy, or
look-alike unless the task explicitly permits that substitution.
Do not spawn other agents or ask the user to perform routine steps. Keep
progress messages brief. Before finishing, leave every requested artifact at
its exact path and run the strongest allowed task-shaped verification. Report
the outcome concisely, including any genuine remaining uncertainty.
```

Listing 18 SWE-Bench Pro medium: `policy.json`: `session_context`

```
The repair workflow is already loaded; do not spend tool calls inventorying harness/.
First establish the repository root under task/ and keep later commands rooted there.
Read task/prompt.md as a contract and keep a concise ledger of exact public names,
files, payloads, defaults, error behavior, compatibility requirements, and stated edge
cases. Inspect the local implementation and nearby tests first. Look for an existing
reference when the task sounds like a known fix: local Git history, tags, upstream
commits, tests, fixtures, or nearby implementation patterns. Prefer adapting proven
behavior over guessing. Trace the real execution and data path before editing, then make
the smallest production-source change that satisfies the contract. Start with narrow
discriminating probes, batch independent reads and searches, and avoid unchanged
rereads. Reserve enough budget for a post-change task-shaped test, investigation of any
concrete failure, and one final diff/contract review; compilation alone is insufficient.
Use harness/bin/repair-verify only when its general checks are useful. Keep generated
files, caches, and temporary tests out of the final patch. Before finishing, inspect the
diff and re-check every ledger item. Preserve unrelated behavior and do not replace
requested behavior with a mock, proxy, or look-alike implementation.
```

Relative to the Terminal-Bench 2 medium rubric in Listing 10, the SWE-Bench Pro rubric changes the profile budgets and the `lean` selection gate. The diff below shows these changes.

Listing 19 SWE-Bench Pro medium: `profile_rubric.md` (diff vs. Terminal-Bench 2)

```
--- tb2/medium/runtime/profile_rubric.md
+++ swebench_pro/medium/runtime/profile_rubric.md
@@ -7,6 +7,6 @@
 - ‘compact’: low reasoning; converge near 4 tool calls and stop near 8.
-- ‘lean’: low reasoning; converge near 10 calls and stop near 16.
-- ‘standard’: medium reasoning; converge near 26 calls and stop near 38.
-- ‘deep’: high reasoning; converge near 30 calls and stop near 48.
-- ‘research’: high reasoning with about 40 calls for source reconstruction.
+- ‘lean’: low reasoning; converge near 16 calls and stop near 24.
+- ‘standard’: medium reasoning; converge near 42 calls and stop near 64.
+- ‘deep’: high reasoning; converge near 48 calls and stop near 72.
+- ‘research’: high reasoning with up to about 64 calls for source reconstruction.

@@ -112,4 +112,4 @@
 4. A direct local check covering nearly all decisive behavior.
-5. About eight or fewer evidence-producing calls for inspection, implementation,
-   and final validation.
+5. About twelve or fewer evidence-producing calls for inspection,
+   implementation, failure diagnosis, and final validation.

@@ -120,2 +120,8 @@

+For an existing-code repair, use ‘lean’ only when the described behavior is
+confined to one narrow implementation surface and one focused local check can
+cover both the requested success path and its stated edge cases. A short prompt
+or a small-looking diff is not enough when compatibility, lifecycle, persistence,
+concurrency, or cross-layer behavior remains coupled.
+
 Do not choose ‘lean’ when several coupled uncertainties require iterative
```

The selected policy fields below specify routing, profile effort and tool stages, confidence handling, and output retention. The full policy also contains Codex CLI feature settings. Tool thresholds count attempted calls, while `history_token_limit` limits each retained tool output, not the model context window.

Listing 20 SWE-Bench Pro medium: `policy.json` (selected fields)

```
{
  "default_profile": "standard",
  "tool_output": {
    "history_token_limit": 1400,
    "max_chars": 6000,
    "head_chars": 3600,
    "tail_chars": 1400,
    "rtk_enabled": true
  },
  "profiles": {
    "compact": {
      "reasoning_effort": "low",
      "economy_after_tools": 2,
      "strict_after_tools": 4,
      "finalize_after_tools": 8,
      "strict_reminder_every": 2
    },
    "lean": {
```

Listing 20 SWE-Bench Pro medium: `policy.json` (selected fields) (continued)

```
      "reasoning_effort": "low",
      "economy_after_tools": 8,
      "strict_after_tools": 16,
      "finalize_after_tools": 24,
      "strict_reminder_every": 4
    },
    "standard": {
      "reasoning_effort": "medium",
      "economy_after_tools": 20,
      "strict_after_tools": 42,
      "finalize_after_tools": 64,
      "strict_reminder_every": 8
    },
    "deep": {
      "reasoning_effort": "high",
      "economy_after_tools": 24,
      "strict_after_tools": 48,
      "finalize_after_tools": 72,
      "strict_reminder_every": 8
    },
    "research": {
      "reasoning_effort": "high",
      "economy_after_tools": 16,
      "strict_after_tools": 40,
      "finalize_after_tools": 64,
      "strict_reminder_every": 8
    }
  },
  "classifier": {
    "type": "model",
    "model": "gpt-5.5",
    "reasoning_effort": "medium",
    "fallback_profile": "standard",
    "rubric_file": "profile_rubric.md",
    "timeout_s": 120,
    "max_attempts": 3,
    "max_task_chars": 24000
  },
  "confidence_overrides": [
    {
      "name": "uncertain_lean_to_standard",
      "source_profile": "lean",
      "below": 0.9,
      "reasoning_effort": "medium",
      "economy_after_tools": 20,
      "strict_after_tools": 42,
      "finalize_after_tools": 64,
      "strict_reminder_every": 8
    }
  ],
  "rules": []
}
```

The hook excerpts reproduce the complete budget gate, stage-reminder, and text-compaction functions. Dispatch, audit, storage, and the remaining helper routines are retained in the accompanying complete `hook.py`.

Listing 21 SWE-Bench Pro medium: `hook.py` (excerpts)

```
def _pre_tool_use(
    payload: dict[str, Any],
    cwd: Path,
    control_state: Path,
    selection: dict[str, Any],
) -> int:
    tool_name = str(payload.get("tool_name") or "")
    state = _increment_tool_count(control_state)
    count = int(state["tool_count"])
    finalize = int(selection.get("finalize_after_tools") or 0)
    if finalize and count > finalize:
        _audit(
            "pre_tool_use",
            cwd=str(cwd),
            tool_name=tool_name,
            tool_count=count,
            decision="deny",
        )
        print(
            json.dumps(
                {
                    "hookSpecificOutput": {
                        "hookEventName": "PreToolUse",
                        "permissionDecision": "deny",
                        "permissionDecisionReason": (
                            "The task tool budget is exhausted. Preserve the current
files, "
                            "stop calling tools, and give the best final answer from
existing evidence."
                        ),
                    }
                }
            )
        )
        return 0

    context = _stage_context(count, selection)
    updated_input = _rtk_rewrite(payload, selection) if tool_name == "Bash" else None
    _audit(
        "pre_tool_use",
        cwd=str(cwd),
        tool_name=tool_name,
        tool_count=count,
        stage=_stage_name(count, selection),
        input_rewritten=updated_input is not None,
    )
    if context or updated_input is not None:
        specific: dict[str, Any] = {"hookEventName": "PreToolUse"}
        if context:
            specific["additionalContext"] = context
        if updated_input is not None:
            specific["permissionDecision"] = "allow"
            specific["updatedInput"] = updated_input
        print(json.dumps({"hookSpecificOutput": specific}))
    return 0

def _stage_context(count: int, selection: dict[str, Any]) -> str:
    economy = int(selection.get("economy_after_tools") or 0)
    strict = int(selection.get("strict_after_tools") or 0)
```

Listing 21 SWE-Bench Pro medium: `hook.py` (excerpts) (continued)

```
    reminder = int(selection.get("strict_reminder_every") or 8)
    if count == economy:
        return (
            "Economy phase starts now. Batch independent ls/rg/read/test checks,
suppress "
            "process narration, and do not repeat unchanged checks. Do not trade away
the "
            "behavioral validation or exact final-state checks required by the task."
        )
    if count >= strict and (count == strict or (count - strict) % reminder == 0):
        return (
            "Convergence phase: stop broad exploration. Only inspect evidence directly
tied to the "
            "current failure or deliverable. If a concrete failure, unexplained
instability, or "
            "mutated deliverable remains, repair it rather than finalizing because the
phase changed. "
            "Avoid irreversible cleanup or broad state changes not required by the task,
and keep "
            "temporary verification artifacts separate from final deliverables. "
            "Then recheck exact paths, interfaces, behavior, and final state once and
finish."
        )
    return ""

def _compact_text(text: str, *, head_chars: int, tail_chars: int) -> str:
    lines = text.splitlines()
    deduped: list[str] = []
    index = 0
    while index < len(lines):
        next_index = index + 1
        while next_index < len(lines) and lines[next_index] == lines[index]:
            next_index += 1
        repeats = next_index - index
        deduped.append(lines[index])
        if repeats > 2:
            deduped.append(f"[previous line repeated {repeats - 1} more times]")
        elif repeats == 2:
            deduped.append(lines[index])
        index = next_index
    compact = "\n".join(deduped)
    if len(compact) <= head_chars + tail_chars:
        return compact
    omitted = len(compact) - head_chars - tail_chars
    return (
        compact[:head_chars]
        + f"\n... [{omitted} characters omitted; use the raw path for a targeted range]
...\n"
        + compact[-tail_chars:]
    )
```

D.3 PROGRAMBENCH

The ProgramBench medium-mode harness uses a fixed GPT-5.5 high-effort profile instead of task-adaptive routing. The workflow builds an independent program from bundled documentation and permitted black-box probes, validates it against the public reference, and reserves time for a clean offline rebuild. Listing 22 shows the stored harness files. This fixed-profile harness has no routing rubric. The evaluator configures Codex to load the instruction file and execute the budget hook.

Listing 22 ProgramBench medium-mode harness files

```
README.md
runtime/
    model_instructions.md
    policy.json
    hook.py
```

Listing 23 ProgramBench medium: `model_instructions.md`

```
# ProgramBench Reconstruction Harness

Use this harness as a task-agnostic control loop for rebuilding a program from
documentation and black-box observations. It supplements the benchmark rules;
it never permits source retrieval, internet access, decompilation, tracing, or
reuse of the reference executable.

## Evidence-first workflow

1. Read all bundled documentation and turn it into a compact behavior ledger:
   commands, flags, inputs, outputs, exit status, file effects, defaults, error
   behavior, formatting, and explicitly documented edge cases.
2. Establish a runnable skeleton early. Create 'compile.sh', build a minimal
   independent executable, and confirm the complete build/run cycle before
   investing in secondary behavior.
3. Probe the reference through its normal public interface. Use small,
   discriminating experiments that separate plausible behaviors; record the
   observed input, output, status, and state change instead of relying on
   assumptions based on the program name.
4. Prioritize breadth before polish. Implement the documented happy paths and
   common input classes, then error handling, boundary cases, formatting, and
   low-frequency options. Keep a list of remaining unsupported behavior.
5. Compare your implementation and the reference on the same locally generated
   cases. When they differ, reduce the case and fix the underlying general rule
   rather than adding a one-off special case.
6. Rebuild from a clean state periodically. A solution that works only because
   of cached artifacts, undeclared tools, or files outside the submission is
   not complete.
7. Before finalizing, verify that 'compile.sh' is executable, produces
   './executable' without network access, and that representative success and
   failure probes agree with the reference.

## Efficiency rules

- Batch independent documentation reads and related probes when practical.
- Prefer high-information probes over broad random exploration.
- Keep outputs bounded and summarize findings in the behavior ledger.
- Stop repeating probes once a stable rule has been established.
- Reserve the final portion of the budget for a clean rebuild and end-to-end
  validation rather than continued exploration.

## Reference lifecycle

Keep the black-box reference available until differential validation is finished.
Develop a separately built candidate at another path, with the same executable
basename where relevant. Do not overwrite the reference during implementation.
If a permitted move is necessary, retain the execute-only reference on the same
filesystem without reading/copying its bytes or changing its permissions.
The reference is only a development oracle: never include it in the submission
or invoke it from the submitted implementation. Self-authored expected outputs
```

Listing 23 ProgramBench medium: `model_instructions.md` (continued)

```
do not replace an actual reference-versus-candidate comparison. Build the final
deliverable from the validated independent source after the last comparison.
```

The policy's `history_token_limit` is a per-output history-retention limit, not the model context size. The hook applies staged tool budgets; its hard-stop threshold is the finalization threshold plus the larger of eight and the reminder interval.

Listing 24 ProgramBench medium: `policy.json`

```
{
  "version": 1,
  "default_profile": "heavy",
  "session_context": "Follow the reconstruction workflow: read bundled documentation,
preserve the reference for permitted black-box comparisons, build an independent
implementation, and reserve time for an offline rebuild and end-to-end validation.",
  "codex": {
    "model_instructions_file": "model_instructions.md",
    "model_verbosity": "low",
    "disabled_features": [
      "apps",
      "auth_elicitation",
      "browser_use",
      "browser_use_external",
      "browser_use_full_cdp_access",
      "computer_use",
      "fast_mode",
      "goals",
      "image_generation",
      "in_app_browser",
      "multi_agent",
      "personality",
      "plugins",
      "remote_plugin",
      "shell_snapshot",
      "skill_mcp_dependency_install",
      "tool_call_mcp_elicitation",
      "tool_suggest",
      "workspace_dependencies"
    ]
  },
  "tool_output": {
    "history_token_limit": 3000,
    "max_chars": 4500,
    "head_chars": 2700,
    "tail_chars": 1100,
    "rtk_enabled": false
  },
  "profiles": {
    "heavy": {
      "reasoning_effort": "high",
      "economy_after_tools": 36,
      "strict_after_tools": 60,
      "finalize_after_tools": 76,
      "strict_reminder_every": 8
    }
  },
  "rules": []
}
```

Listing 25 ProgramBench medium: `hook.py`

```
#!/usr/bin/env python3
"""Task-agnostic staged budget hook for ProgramBench Codex runs."""

from __future__ import annotations

import fcntl
import json
import os
from pathlib import Path
import sys


def _selection() -> dict:
    path = os.environ.get("RHO_RUNTIME_SELECTION_PATH", "").strip()
    if not path:
        return {}
    try:
        value = json.loads(Path(path).read_text(encoding="utf-8"))
    except (OSError, json.JSONDecodeError):
        return {}
    return value if isinstance(value, dict) else {}


def _next_count(cwd: Path) -> int:
    configured = os.environ.get("RHO_RUNTIME_STATE_ROOT", "").strip()
    state_dir = Path(configured).resolve() if configured else cwd / ".rho_runtime"
    state_dir.mkdir(parents=True, exist_ok=True)
    path = state_dir / "tool-count.json"
    lock_path = state_dir / "tool-count.lock"
    with lock_path.open("a+") as lock:
        fcntl.flock(lock, fcntl.LOCK_EX)
        try:
            value = json.loads(path.read_text(encoding="utf-8"))
            count = int(value.get("tool_count") or 0) + 1
        except (OSError, ValueError, TypeError, json.JSONDecodeError):
            count = 1
        temp = path.with_suffix(".tmp")
        temp.write_text(json.dumps({"tool_count": count}) + "\n", encoding="utf-8")
        temp.replace(path)
        return count


def _context(event: str, text: str) -> None:
    print(json.dumps({"hookSpecificOutput": {"hookEventName": event,
"additionalContext": text}}))


def main() -> int:
    payload = json.load(sys.stdin)
    event = str(payload.get("hook_event_name") or "")
    selection = _selection()
    if event == "SessionStart":
        context = str(selection.get("session_context") or "").strip()
        if context:
            _context("SessionStart", context)
        return 0
    if event != "PreToolUse":
        return 0
    cwd = Path(str(payload.get("cwd") or os.getcwd())).resolve()
```

Listing 25 ProgramBench medium: `hook.py` (continued)

```
    count = _next_count(cwd)
    economy = int(selection.get("economy_after_tools") or 0)
    strict = int(selection.get("strict_after_tools") or 0)
    finalize = int(selection.get("finalize_after_tools") or 0)
    reminder = int(selection.get("strict_reminder_every") or 1)
    hard_stop = finalize + max(8, reminder) if finalize else 0
    if hard_stop and count > hard_stop:
        print(
            json.dumps(
                {
                    "hookSpecificOutput": {
                        "hookEventName": "PreToolUse",
                        "permissionDecision": "deny",
                        "permissionDecisionReason": (
                            "The exploration budget is exhausted. Preserve the current
files, "
                            "finish from existing evidence, and do not call more tools."
                        ),
                    }
                }
            )
        )
        return 0
    if finalize and count >= finalize and (
        count == finalize or (count - finalize) % max(4, reminder) == 0
    ):
        _context(
            "PreToolUse",
            "Finalization stage: stop adding semantic scope. Use the remaining small
grace "
            "window only for implementation completion, a clean build, a focused check, "
            "artifact cleanup, or submission. Then finish.",
        )
    elif strict and count >= strict and (count == strict or (count - strict) % reminder
== 0):
        _context(
            "PreToolUse",
            "Convergence stage: stop broad exploration. Revisit the public behavior
ledger, "
            "use only probes that distinguish a live hypothesis or repair a concrete
mismatch, "
            "and preserve enough budget for an offline rebuild and final contract
check.",
        )
    elif economy and count == economy:
        _context(
            "PreToolUse",
            "Economy stage: consolidate the behavior ledger, batch related checks, and
cover "
            "remaining documented semantic families before low-frequency polish.",
        )
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```