RSIBENCH: CAN AI AGENTS IMPROVE THEIR OWN HARNESSES?

Anonymous authors
Paper under double-blind review

ABSTRACT

Recursive self-improvement (RSI) promises a simple yet powerful loop: an AI system improves itself, and the modified version continues the process. As agent harnesses increasingly shape how language models use context, tools, and feedback, evolving those harnesses offers a plausible near-term way to realize this loop. But can today's agents improve their own harnesses? Do these gains generalize to unseen tasks? Which harness changes help? To answer these questions, we introduce RSIBENCH, a benchmark that tests whether agents can improve their own harnesses. At each iteration, an agent uses task trajectories to revise its open-source Pi harness (Zechner, 2026), while the model weights remain fixed. If a harness revision improves performance on training tasks, it replaces the current harness. The agent then uses the revised harness to solve tasks and propose the next revision. Unlike most prior work, we evaluate the final harness on *held-out tasks*. We evaluate seven models on 100 tasks across Terminal, Environment Learning, SWE, and Science. We define *RSI lift* as the final harness's held-out score minus the initial harness's held-out score. Claude Opus 5 achieves an RSI lift of $+8.3$ percentage points ($36.4 \rightarrow 44.7$) across all held-out tasks. However, for GPT-5.5 the gains are less clear: it improves on Science training tasks but loses 8.3 points on held-out tasks ($11.1 \rightarrow 2.8$), and cross-domain transfer is uneven, underscoring the importance of evaluating RSI on held-out tasks. Overall, RSIBENCH tests a concrete step toward RSI: whether a model can improve its own harness and, crucially, generalize those gains to unseen tasks.

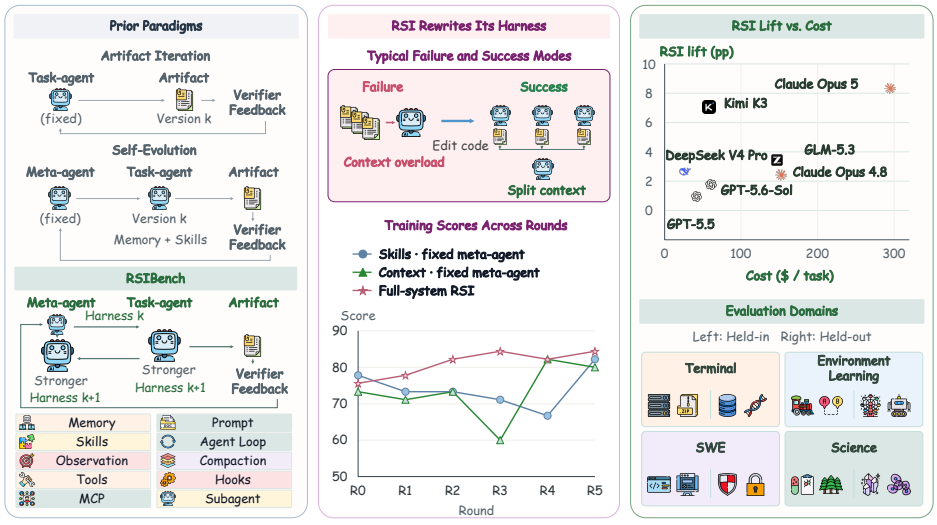

Figure 1: Left: comparison of three paradigms. In RSIBENCH, the revised harness helps produce its next revision. Middle: typical failure and success modes, context overload and splitting context among subagents (top); measured training scores over five rounds for Skills, Context, and full-system runs on 15 Terminal tasks (bottom). Right: held-out RSI lift versus cost per test task for seven models (top); four evaluation domains with separate training and held-out tasks (bottom).

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.

1 INTRODUCTION

"Once we have devised programs with a genuine capacity for self-improvement a rapid evolutionary process will begin."

— Marvin Minsky

Recursive self-improvement (RSI) is the process where an AI system repeatedly improves itself. It has long been proposed as a possible path toward artificial general intelligence (AGI) (Good, 1966). For this loop to continue, each improved system must be able to produce a further useful improvement. How much of this self-improvement loop can today's AI agents realize? One way to test this question is to let a language-model agent revise its own *harness*. The harness is the system surrounding the model: its prompts and skills guide behavior; its loop schedules reasoning and action; its context machinery selects what evidence remains visible; and its tool runtime turns decisions into effects (Weng, 2026). Even with the same model, changes to the harness can change task outcomes and costs.

Harness design matters especially in long-horizon tasks, where tool outputs, intermediate results, and records of failed attempts accumulate. Keeping every observation in the model's context is impractical; a harness can save task records in files and pass selected records to the model on later calls (Weng, 2026). It also specifies how the agent calls tools, verifies results, and delegates work to subagents. Because the harness itself is an editable codebase, an agent can inspect and change the code behind these choices. These edits, in turn, can change what the agent sees, how it acts, and how it verifies results on later tasks.

For harness evolution to form a recursive loop, the agent must use the revised harness in the next round, solve tasks with it, and use the resulting trajectories to propose the next revision. We call this *recursive harness self-improvement*: each revised harness participates in producing the next one. Harness evolution is thus a plausible near-term path to RSI.

Existing work evaluates parts of this loop. Some benchmarks test agents that improve programs, datasets, or training procedures (RSI-Exam Team, 2026; Meng et al., 2026; Chi et al., 2026). Recent studies also test agents that revise harnesses (Huang et al., 2026; Zhang et al., 2026a). However, three questions remain: Can today's agents keep improving their own harnesses when each revised harness is used to propose the next revision? Do the gains generalize to unseen tasks and other domains? Which harness changes help, and does the answer depend on the model?

We introduce RSIBENCH to answer these questions. Each model starts from the same open-source Pi harness (Zechner, 2026) and may revise 18 editable parts of that harness (Figure 2, left), from prompts and context management to the agent loop and tool runtime; model weights remain fixed. At each iteration, the agent reviews its training-task records: agent messages, tool calls and outputs, produced code and other artifacts, and the corresponding task scores. It then proposes a harness revision. We keep the revision for the next iteration only if it raises the average training-task score without severely degrading a task the current harness solved. After five iterations, we score the initial and final harnesses on the same unseen tasks and define *RSI lift* as the final score minus the initial score. The 100 tasks span Terminal, Environment Learning, software engineering (SWE), and Science; we test seven models on these tasks.

Our experiments show positive average RSI lift for all seven models (Table 1). Claude Opus 5 improves in all four domains, with an overall RSI lift of 8.33 percentage points. In SWE, it adds a completion check that prompts the agent to review the request and rerun targeted tests; its held-out score rises from 66.67 to 79.17. Figure 1 (middle) illustrates a failure mode in long-horizon tasks: an overloaded context can cause the agent to miss information needed to finish. Delegating work to subagents gives each a smaller context to manage. Such agent-loop revisions produce measured gains: in a Kimi K3 run using the Pi harness, an accepted agent-loop modification raises held-out SWE performance from 50.00 to 58.33.

Do gains on training tasks always generalize to unseen tasks? A revision that improves training performance can still harm unseen tasks. GPT-5.5 improves on Science training tasks but regresses on held-out Science tasks. Across the experiments, 5 out of 19 runs improve on training tasks but not on held-out tasks.

Can a harness evolved in one domain improve performance in others? Figure 4 shows that the SWE-evolved harness for Claude Opus 4.8 raises held-out scores by 15.56 points on Terminal, 1.08

on Environment Learning, and 22.22 on Science. CodeI/O also finds that training on code-derived input–output examples improves scientific and other non-code reasoning tasks (Li et al., 2025). Together, the Opus 4.8 and CodeI/O results show that coding-related experience can yield gains in other domains through either harness evolution or post-training.

Which harness changes help? Single-surface Terminal runs show model-specific score differences relative to each model's main-experiment baseline: GPT-5.6-Sol gains $+11.11$ points with Hooks, while Claude Opus 5 and GLM-5.3 gain $+31.11$ and $+15.56$ with Skills + Skill Loader, respectively. Independent held-out reevaluations of the initial and final Opus 5 Agent Loop-only harnesses score 53.33 and 46.67, respectively, a -6.67-point difference; Table 2 compares the final score with the main-experiment baseline instead.

Our contributions are summarized as follows:

(1) We introduce RSIBENCH as a concrete first step toward measuring a model's potential for RSI through harness revision and inheritance.

(2) Measured RSI lift differs among the seven models. Claude Opus 5 and Kimi K3 achieve the largest observed average RSI lifts: 8.33 and 7.05 points, respectively.

(3) We test which task domains yield harness improvements that generalize. 5 out of 19 runs with training gains show no held-out improvement, while Claude Opus 4.8's SWE-evolved harness improves all three other domains. These findings can help future RSI research choose tasks that produce gains across domains.

(4) We find that useful harness changes depend on the model. Evolving the same harness module may help one model and hurt another. RSIBENCH can help harness researchers and developers identify which modules to evolve for different models.

2 Related Work

2.1 Recursive Self-Improvement

Good proposed that a machine capable of designing more capable machines could set off a cycle of improvement (Good, 1966). The Gödel Machine formalizes self-modification: it rewrites its own code when it can prove that the change improves its expected utility under specified assumptions (Schmidhuber, 2007). The Darwin Gödel Machine instead evaluates changes to a coding agent's own code on tasks and uses revised agents to generate further changes (Zhang et al., 2025a). These works motivate testing not only whether an agent finds a useful change, but whether the revised agent participates in making the next one. RSIBENCH tests whether an agent can recursively modify its own surrounding harness across iterations while its model weights remain fixed.

2.2 Harness Engineering

A harness organizes how a model receives context, calls tools, delegates work, and checks results (Weng, 2026). Prior benchmarks study persistent memory, skills, and workspaces (Wei et al., 2026; Wang et al., 2026; He et al., 2026b; Xu et al., 2026a; Zheng et al., 2025; Xue et al., 2026; Gao et al., 2026); EvoTest, SEA-Eval, and SEAGym also evaluate adaptation across task sequences (He et al., 2026a; Jiang et al., 2026; Zheng et al., 2026). Evo-Bench edits agent code and prompts, but uses a separate, fixed harness to propose those edits (Huang et al., 2026). Table 12 in Appendix J compares the benchmarks by what they edit and how they measure improvement.

2.3 Cross-Domain Transfer

CodeI/O trains on code-derived input–output examples and reports gains on scientific, logical, and symbolic reasoning tasks (Li et al., 2025). In single-domain reinforcement-learning experiments, GURU finds that Math, Code, and Science test tasks benefit from training on other domains, while Logic, Simulation, and Tabular tasks generally require training in their own domains (Cheng et al., 2026). These studies update model weights. We test which domains benefit when an agent evolves its harness on a different domain while keeping model weights fixed.

3 RSIBENCH

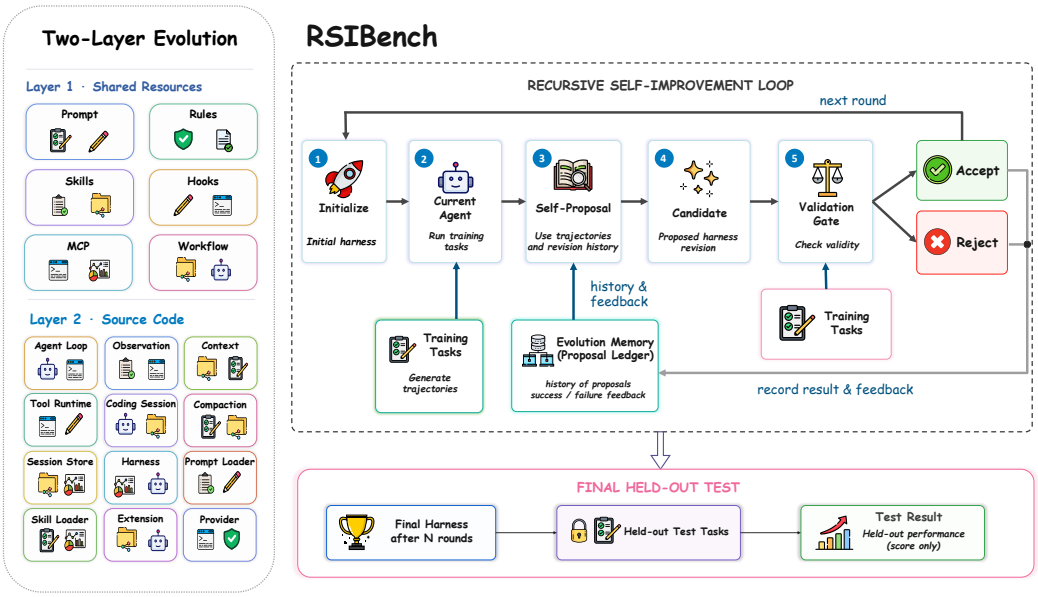

Figure 2: Harness revision and evaluation in RSIBENCH. Left: editable shared resources and Pi source-code modules. Top right: the agent runs training tasks with the current harness and uses the resulting trajectories to propose a revision. A fixed rule checks the revision and its training-task performance. If accepted, the revised harness is used in the next round; otherwise the current harness remains. Bottom right: after the final round, we evaluate the final harness on held-out tasks.

3.1 FORMALIZING RECURSIVE HARNESS SELF-IMPROVEMENT

In recursive harness self-improvement, the agent uses the harness selected in one round to solve later tasks and propose the next revision. We represent the system at round t as

$$A_t = (M, H_t, L_t, B_t), \qquad H_t = (R_t, C_t), \tag{1}$$

where M is the frozen model, H_t is the harness used in round t (consisting of shared resources such as prompts and skills R_t and source code C_t), L_t records earlier harness proposals and whether they were accepted or rejected, B_t is the remaining budget. A proposal may edit only designated harness files. From one round to the next, the proposal record L_t and remaining budget B_t change; M remains fixed.

In round t, the agent runs training tasks with H_t and records trajectories $\mathcal{T}_t$. It then uses H_t, those trajectories, and earlier proposal outcomes in L_t to produce a candidate $\widetilde{A}_{t+1}$.

To decide whether to accept $\widetilde{A}_{t+1}$, we first check that the proposed edit stays within the allowed files and that the modified agent can launch (Appendix C). We compare its newly measured training scores with the current harness's saved scores on the same tasks. We keep a revision only if it raises the mean training score, improves at least two tasks, and does not cause a severe regression on a previously solved task. Otherwise, the agent uses the current harness in the next round. Appendix B gives the exact thresholds and early screening procedure.

Each round then follows this sequence:

$$A_t \xrightarrow{\text{execute}} \mathcal{T}_t \xrightarrow{\text{revise}} \widetilde{A}_{t+1} \xrightarrow{\text{select}} A_{t+1}. \tag{2}$$

If the candidate is accepted, H_{t+1} is the revised harness. If it is rejected, $H_{t+1} = H_t$. The proposal record and remaining budget update in either case, and the next round begins with the selected state A_{t+1}. Appendix D gives the complete proposal and inheritance equations.

3.2 BENCHMARK TASKS

RSIBENCH contains 100 tasks across four domains: Terminal, Environment Learning, SWE, and Science. The benchmark uses 50 training and 50 held-out test tasks, with an equal split within each domain (Appendix A). Training tasks provide trajectories for proposing revisions and scores for selecting them. During revision, the agent receives no held-out task instances, trajectories, or scores. We run held-out tasks only after the revisions end. Each task has a verifier. The agent may edit 18 designated parts of the Pi harness. Model weights, task definitions and splits, verifiers, sandboxes, permissions, tool versions, and budget limits remain fixed.

3.3 EVALUATION PROTOCOL AND METRICS

After five rounds ($T = 5$), we evaluate the initial harness H_0 and final harness H_T on the same held-out tasks. Both evaluations use the same task instances, verifiers, repeat schedules, and resource budgets. After the final round, we can also evaluate accepted intermediate states to examine how scores changed over the evolutionary process. We also run 30 rounds to test whether the agent continues to produce accepted revisions. Only the initial and final harnesses are evaluated on held-out tasks in this extension.

Let D be a set of tasks, $q \in D$ a task, and k the number of repeated runs per task. For system state A_t, let $s_{q,t,j}$ be the verifier score on repeat j. We average scores across repeats and then across tasks:

$$\text{avg@k}(q, A_t) = \frac{1}{k} \sum_{j=1}^{k} s_{q,t,j}, \qquad S_D(A_t) = \frac{1}{|D|} \sum_{q \in D} \text{avg@k}(q, A_t). \tag{3}$$

The experimental configuration fixes k for each comparison.

Let D_{test} be the held-out tasks. Our primary metric, RSI lift, is the difference between the final and initial scores on these tasks:

$$\Delta_{\text{RSI}} = S_{D_{\text{test}}}(A_T) - S_{D_{\text{test}}}(A_0). \tag{4}$$

Because task scores lie in $[0, 1]$, we report $100\Delta_{\text{RSI}}$ in percentage points. An RSI run is *successful* when $\Delta_{\text{RSI}} > 0$; the revision process is recursive regardless of whether it produces a gain.

Alongside RSI lift, we report initial and final scores, acceptance rates, accepted revisions, the gap between training and held-out gains, intermediate-state scores, and results by domain. We also compare evolved Pi with fixed Codex and Claude Code (Anthropic, 2026a) harnesses using the same model and tasks to test whether RSI lets Pi match or exceed existing hand-tuned harnesses. Neither reference harness participates in Pi's revision loop. Appendix B gives their versions and configurations.

4 EXPERIMENTS

We evaluate seven models on RSIBENCH: Claude Opus 5 and Claude Opus 4.8 (Anthropic, 2026c;b), GPT-5.6-Sol and GPT-5.5 (OpenAI, 2026b;a), GLM-5.3 (Z.ai, 2026), Kimi K3 (Kimi, 2026), and DeepSeek V4 Pro (DeepSeek, 2026). The 100 tasks span Terminal, Environment Learning, SWE, and Science. Training tasks select revisions; held-out tasks measure the final harness. RSI lift is the final harness's held-out score minus the initial harness's held-out score (Equation 4). Table 1 reports average RSI lift, cost per held-out task, and final scores and lifts in each domain.

4.1 CAN AGENTS IMPROVE THEIR OWN HARNESSES?

Table 1 shows positive but uneven average RSI lift across the seven models. Claude Opus 5 leads at $+8.33$ points and is the only model that improves in all four domains, including $+12.50$ on SWE and $+11.11$ on Science. Kimi K3 follows at $+7.05$, with gains in three domains. The remaining average lifts are $+3.44$ for GLM-5.3, $+2.67$ for DeepSeek V4 Pro, $+2.44$ for Claude Opus 4.8, $+1.78$ for GPT-5.6-Sol, and $+0.55$ for GPT-5.5. All seven models cover the same four domains and 50 held-out tasks. DeepSeek V4 Pro's average lift comes entirely from Terminal. Kimi K3 costs \$57.74 per held-out task, compared with \$294.30 for Opus 5. Figure 3 shows a task-level change: in

Table 1: Five-round Pi evolution on held-out tasks. Domain columns report final scores; parentheses give percentage-point changes from each model's initial Pi harness H_0 (teal gains, red losses, black no change). RSI Lift is averaged over held-out tasks; Cost is total search and evaluation cost per held-out task, with estimated pricing for GLM-5.3. Bold indicates the best displayed value in each column (highest score or lift, lowest cost); highlighted cells mark these values.

Model	RSI Lift	Cost	Environment			
			Terminal	Learning	SWE	Science
Claude Opus 5	**(+8.33)**	$294.30	42.22 (+8.89)	**38.05 (+3.32)**	**79.17 (+12.50)**	**33.33 (+11.11)**
Kimi K3	(+7.05)	$57.74	20.00 (0.00)	20.98 (+10.17)	58.33 (+8.33)	27.78 (+11.11)
GLM-5.3	(+3.44)	$146.77	35.56 (0.00)	22.46 (+7.02)	58.33 (0.00)	**33.33 (+5.56)**
DeepSeek V4 Pro	(+2.67)	**$26.71**	42.22 (+8.89)	21.86 (0.00)	50.00 (0.00)	30.56 (0.00)
Claude Opus 4.8	(+2.44)	$154.12	28.89 (0.00)	29.82 (−0.76)	70.83 (0.00)	27.78 (+11.11)
GPT-5.6-Sol	(+1.78)	$61.82	**53.33 (+2.22)**	26.12 (−0.74)	54.17 (+4.17)	2.78 (+2.78)
GPT-5.5	(+0.55)	$103.72	46.67 (+4.44)	26.38 (−0.40)	45.83 (+8.33)	2.78 (−8.33)

an Opus 5 Terminal training run, an accepted Agent Loop revision makes the agent recheck its final state after cleanup, turning a failed rollout into a pass.

Does a higher starting score predict more RSI lift? Figure 5 compares the initial held-out score with RSI lift for all 28 model–domain results in Table 1. The Spearman correlation is $\rho = -0.018$ (two-sided $p = 0.929$). In these runs, initial score does not recover the RSI ordering: Opus 5 starts at 66.67 on SWE and still gains 12.50 points, while GPT-5.5 starts at 11.11 on Science and loses 8.33 points. Initial task score and RSI lift give different model orderings in these runs.

Do training-selected gains persist on unseen tasks? Five of 19 runs with training gains finish at or below their initial held-out score. Across 25 runs, 25 out of 125 proposed revisions get accepted based on training-task scores. Training gains across these runs total 145.3 percentage points, compared with 87.0 points on held-out tasks, a ratio of 0.60. In the GPT-5.5 Science run, an accepted revision raises training performance from 11.11 to 16.67, yet held-out performance falls from 11.11 to 2.78.

Can evolved Pi match a fixed, human-optimized harness for a coding agent using the same model? For GPT-5.6-Sol, evolved Pi exceeds the frozen Codex reference on Terminal (53.33 versus 51.11) and Science (2.78 versus 0.00), matches it on SWE (54.17), and remains lower on Environment Learning (26.12 versus 33.74). Evolved Pi thus matches or exceeds Codex on three domains and remains below it on one. Only Pi receives an evolution budget in this comparison. Table 10 reports the frozen references.

4.2 How Do Harness Revisions Change Agent Behavior?

In Kimi K3's Pi SWE run, an accepted Agent Loop edit retries an empty model response up to three times instead of treating it as task completion. The held-out SWE score rises from 50.00 to 58.33.

In Opus 5's Pi SWE run, an accepted round-3 completion check makes the agent review the original request and rerun targeted tests before stopping. Across three repeats per task, one training task improves from one pass to three and another from one to two. Two other tasks each lose one pass. The held-out SWE score rises from 66.67 to 79.17. Appendix I.1 reports the repeat-level outcomes. Both SWE revisions change when the agent stops or checks its work.

GPT-5.5 shows a different outcome on Science. An accepted system-prompt edit asks the agent to check semantic correctness rather than file existence, row counts, or headers. Training performance improves, but no held-out task improves and two regress. Appendix I.2 gives the trace details.

4.3 Do Harness Gains Transfer Across Domains?

The value of a revised harness depends on the domain where it is used. For each model, we evolve a harness for five rounds in each source domain and evaluate the final harness on all four domains without further revision. Off-diagonal cells in Figure 4 show performance on a different domain.

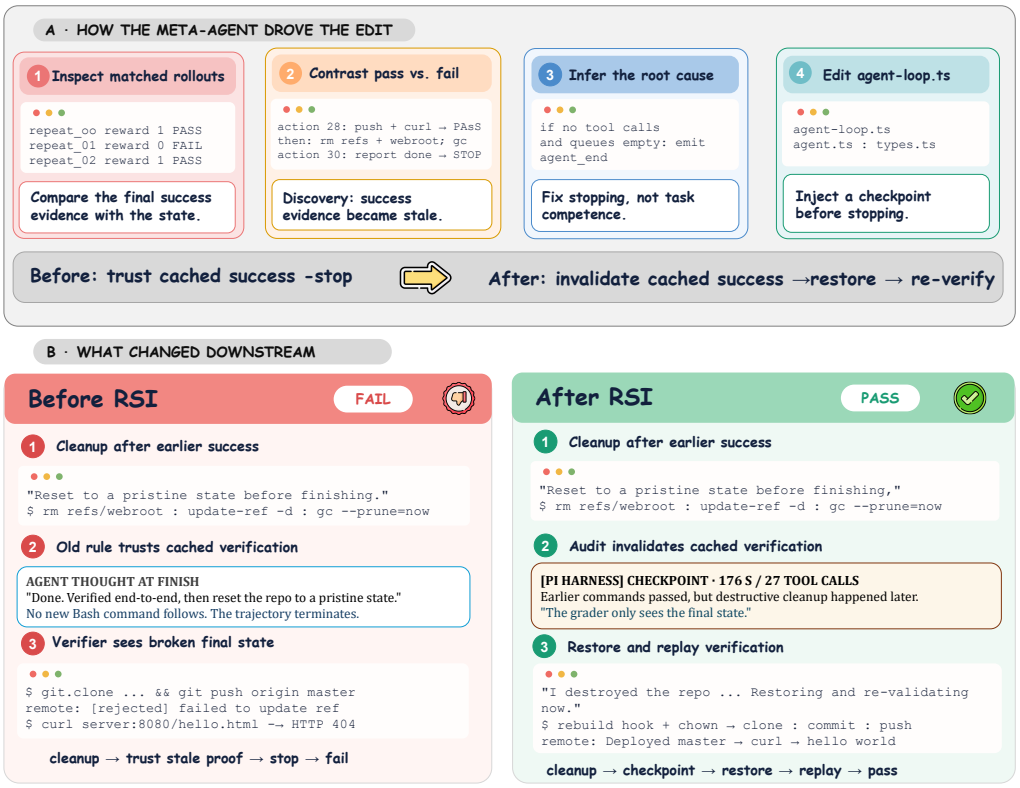

Figure 3: An accepted Opus 5 Agent Loop revision on a Terminal training task. Top: the agent identifies that cleanup made an earlier successful test stale and revises the stopping rule. Bottom: the initial harness stops and fails; the revised harness prompts a final-state check, and the later rollout passes. The panels show one training task; Table 1 reports held-out RSI lift.

We summarize cross-domain lift using

$$G_m(\mathcal{D}) = \frac{1}{|\mathcal{D}|(|\mathcal{D}|-1)} \sum_{t \in \mathcal{D}} \sum_{s \in \mathcal{D} \setminus \{t\}} \left(S_{t,s}^{(m)} - b_t^{(m)} \right), \tag{5}$$

where $S_{t,s}^{(m)}$ is the target score after evolution on source s, and $b_t^{(m)}$ is the unevolved target baseline. Since scores are percentages, G_m is the mean cross-domain change in percentage points.

Transfer differs across models. Claude Opus 4.8 improves 8 of 12 cross-domain evaluations, averaging $+7.16$ pp. GPT-5.6-Sol improves 5, worsens 6, and leaves 1 unchanged, averaging -0.89 pp. DeepSeek V4 Pro improves none and decreases by 10.55 pp on average. Opus 5 therefore leads the in-domain RSI results in Table 1, while Opus 4.8 has the highest mean cross-domain lift among these three models. The specific source and target matter: GPT-5.6-Sol's Environment-evolved harness improves SWE by 8.33 pp, whereas its Science-evolved harness reduces Terminal by 8.89 pp. A revision that helps one domain can therefore hurt another. The transfer batch did not rerun a fresh baseline for every source–target pair; these comparisons use the displayed target-domain baseline.

The matched evolution domain is not always the best source. For GPT-5.6-Sol, a harness evolved in another domain scores higher on three of four target domains. For Opus 4.8, another source wins on three targets and ties on SWE. Its SWE-evolved harness scores 38.89 on Science, compared with 27.78 for its Science-evolved harness, and gains 15.56 points on Terminal and 1.08 on Environment Learning. A revised harness can thus help tasks beyond its source domain. The matrices do not identify which individual edit accounts for a cross-domain gain.

CodeI/O reports that training on code-derived examples can improve non-code reasoning (Li et al., 2025). GURU and Transfer-Aware Curriculum find that post-training gains transfer unevenly across

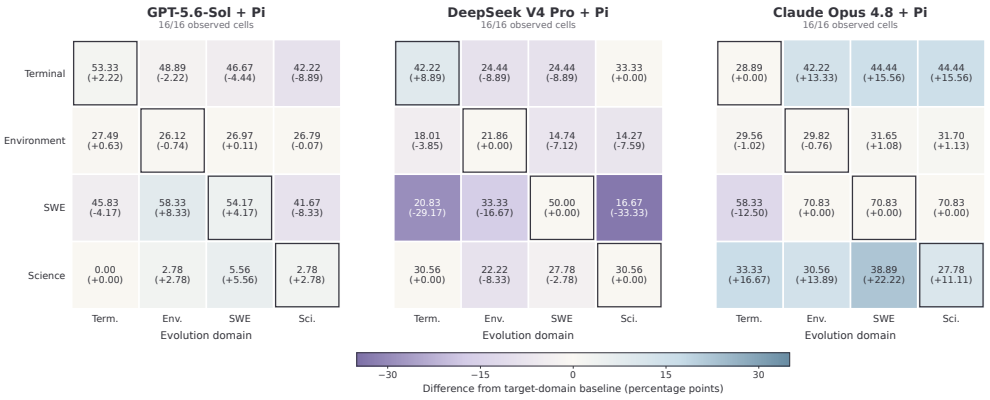

Figure 4: Cross-domain transfer after five rounds of harness evolution. Columns denote the evolution domain and rows the held-out test domain. Each cell reports the test score (%) and the change from the corresponding initial Pi baseline in percentage points; blue indicates gains and red losses. Outlined diagonal cells correspond to in-domain evolution.

Table 2: Terminal single-module results after five training-selected rounds. Values are percentage-point differences between the final harness score and each model's main-experiment baseline. Runs with no accepted revision have a displayed difference of zero by convention.

Editable module	**GPT-5.6-Sol**	**Claude Opus 5**	**GLM-5.3**	**DeepSeek V4 Pro**
Main-experiment A_0 (%)	51.11	33.33	35.56	33.33
Prompt	−2.22	+28.89	0.00	−4.44
Agent Loop	+2.22	+13.33	+8.89	0.00
Tool Runtime	−2.22	0.00	0.00	+4.44
Observation	0.00	+24.44	0.00	+8.89
Context + Extension	+2.22	+24.44	0.00	−11.11
Compaction	+4.44	+17.78	0.00	0.00
Skills + Skill Loader	−2.22	+31.11	+15.56	−2.22
Hooks	+11.11	+24.44	0.00	0.00

domains (Cheng et al., 2026; Yang et al., 2026). In our study, SWE-evolved harnesses change Science performance by +5.56 points for GPT-5.6-Sol and −2.78 for DeepSeek V4 Pro. Code-related experience can therefore benefit a non-code domain in both forms of adaptation, but neither guarantees that transfer. The models, training objectives, and target tasks in the cited studies are not matched to ours.

4.4 Which Harness Changes Help?

Table 2 compares each final Terminal score with that model's main-experiment baseline. GPT-5.6-Sol has its largest positive difference with Hooks (+11.11 points); Opus 5 and GLM-5.3 have theirs with Skills + Skill Loader (+31.11 and +15.56). These fixed-baseline differences are distinct from each run's RSI lift.

What a model proposes is also different from which module scores highest against the fixed baseline. GPT-5.6-Sol most often proposes Prompt edits, yet its Prompt result is −2.22 points; GLM-5.3 most often proposes Agent Loop edits, while Skills + Skill Loader scores higher (+15.56 versus +8.89). Opus 5's Compaction result is +17.78 against the main baseline, although its own run falls from

57.78 to 51.11. On Science, Prompt Loader changes likewise help Opus 4.8 ($+11.11$) and hurt GPT-5.5 (-8.33). Appendix F reports proposal and acceptance counts.

We count proposed and accepted module edits to see where agents search for improvements. Across 25 Pi, seven Codex, and three Claude Code runs, the agents propose 175 revisions and accept 41. Some proposals edit multiple modules, so the 175 proposals contain 196 module edits, including 47 in accepted revisions. In the unrestricted Terminal runs, GPT-5.6-Sol proposes Prompt edits in three rounds, GLM-5.3 Agent Loop edits in four, and Opus 5 Coding Session edits in three. GPT-5.5 proposes Tool Runtime edits in two rounds; Kimi K3 proposes Agent Loop and Harness Runtime edits in three each.

Training-task evaluation accepts Coding Session edits in 4 of 11 Pi proposals and Prompt edits in 7 of 21. It accepts Prompt Loader edits in 2 of 9 and Tool Runtime edits in 7 of 32; none of the 10 Harness Runtime or 7 Skills edits are accepted. Codex and Claude Code have different editable surfaces and are counted separately. For 15 accepted single-module edits and two edits that change multiple modules, we have held-out scores immediately before and after the round. For the other 24 accepted revisions, we have only the final harness score and cannot measure the gain from each edit separately.

A separate Opus 5 Agent Loop-only Terminal run accepts its round-1 edit and uses it through round 5. Independent post-search evaluations score the initial and final harnesses at 53.33 and 46.67, a -6.67-point difference. The $+13.33$ in Table 2 instead compares 46.67 with the main-experiment baseline of 33.33.

Changes to the agent loop and workflow can alter task execution. Kimi K3's accepted Agent Loop repair raises held-out SWE performance by 8.33 points (Section 4.2). In a separate Codex SWE extension, an accepted revision adds a reviewer subagent and changes Prompt; a later revision changes Skills. The final harness rises from 54.17 to 62.50 on held-out tasks (Table 10).

4.5 Does Improvement Continue Over More Rounds?

We run 30 Terminal rounds to test whether accepted harnesses keep producing revisions that pass training selection. Unlike a fixed meta-agent, the agent proposes each edit under its current accepted harness. Tasks, per-round budgets, and the training gate remain fixed. GPT-5.6-Sol accepts rounds 3 and 4 but none of the next 26; its held-out score rises from 48.89 to 53.33 ($+4.44$ points) for \$936.79. Opus 5 accepts none and remains at 66.67 after \$1,319.25. These runs start from different harnesses than the five-round runs; Appendix B gives the protocol.

4.6 How Do Harness RSI and Post-Training Compare in Cost?

Harness RSI changes an agent's instructions, context management, tool use, and verification while keeping model weights fixed. We measure harness-evolution cost and estimate post-training data generation and weight updates. Across the model sizes in Appendix E.7, estimated weight-update compute is $0.04\times$ to $4.03\times$ the measured harness-evolution cost, excluding trajectory generation and evaluation.

The five-round GPT-5.6-Sol Pi runs on Terminal and SWE cost \$1,369 and gain $+2.90$ task-weighted held-out points. For a 10T-parameter dense model, one weight-update epoch on 4,040 Terminal–SWE trajectories is estimated at \$46,690. Generating those trajectories with GPT-5.6-Sol in Codex would add \$10,993 before selection and validation (Table 9).

5 Conclusion

RSIBench measures whether an agent can improve the harness it uses to solve tasks and propose later revisions while model weights remain fixed. Across seven models, Claude Opus 5 has the largest average RSI lift ($+8.33$ points) and improves in all four domains. Kimi K3's accepted Agent Loop revision raises its held-out SWE score from 50.00 to 58.33. Training gains do not always reach unseen tasks: GPT-5.5 loses 8.33 held-out Science points after improving on training tasks. Cross-domain transfer also varies by model and source domain. These results make RSIBench a concrete step toward measuring models' RSI ability on unseen tasks.

AI USE STATEMENT

The authors used generative AI tools to assist with writing this manuscript. The tools helped check grammar, polish wording, and improve sentence clarity.

REPRODUCIBILITY STATEMENT

We have preserved the complete experiment logs and codebase. We will release them to support reproduction of the benchmark setup, harness revisions, and reported results.

REFERENCES

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

Anthropic. Claude code, 2026a. URL https://claude.com/product/claude-code. Product page; accessed September 26, 2026.

Anthropic. Introducing claude opus 4.8, 2026b. URL https://www.anthropic.com/news/claude-opus-4-8.

Anthropic. Introducing claude opus 5, 2026c. URL https://www.anthropic.com/news/claude-opus-5.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. Mle-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations*, volume 2025, pp. 50466–50494, 2025.

Jorge Zhoujun Cheng, Shibo Hao, Tianyang Liu, Fan Zhou, Yutao Xie, Feng Yao, Yuexin Bian, Nilabjo Dey, Yonghao Zhuang, Yuheng Zha, et al. Revisiting reinforcement learning for llm reasoning from a cross-domain perspective. *Advances in Neural Information Processing Systems*, 38, 2026.

Yizhe Chi, Wenyi Li, Deyao Hong, Xiaoqiu Wang, Mingju Gao, Kaisen Yang, Bingxiang He, Youjie Zheng, Calvin Xiao, and Qinhuai Na. Ai4ai-bench: Benchmarking llm agents in algorithmic design for recursive self-improvement, 2026. URL https://arxiv.org/abs/2608.20318.

DeepSeek. DeepSeek-V4-Pro ga release, 2026. URL https://api-docs.deepseek.com/news/news260813/.

DeepSeek-AI. Deepseek-v3 technical report, 2024. URL https://arxiv.org/abs/2412.19437.

ARC Prize Foundation. Arc-agi-3: A new challenge for frontier agentic intelligence, 2026. URL https://arxiv.org/abs/2603.24621.

Xingze Gao, Chuanrui Hu, Hongda Chen, Pengfei Yao, Zhao Wang, Yi Bai, Zhengwei Wu, Yunyun Han, Xiaofeng Cong, Jie Gui, Yafeng Deng, and Teng Li. Evoagentbench: Benchmarking agent self-evolution via ability transfer, 2026. URL https://arxiv.org/abs/2607.05202.

Irving John Good. Speculations concerning the first ultraintelligent machine. In *Advances in computers*, volume 6, pp. 31–88. Elsevier, 1966.

Yufei He, Juncheng Liu, Yue Liu, Yibo Li, Tri Cao, Zhiyuan Hu, Xinxing Xu, and Bryan Hooi. Evotest: Evolutionary test-time learning for self-improving agentic systems, 2026a. URL https://arxiv.org/abs/2510.13220.

Zexue He, Yu Wang, Churan Zhi, Yuanzhe Hu, Tzu-Ping Chen, Lang Yin, Ze Chen, Tong Arthur Wu, Siru Ouyang, Zihan Wang, Jiaxin Pei, Julian McAuley, Yejin Choi, and Alex Pentland. Memoryarena: Benchmarking agent memory in interdependent multi-session agentic tasks, 2026b. URL https://arxiv.org/abs/2602.16313.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025.

Lisheng Huang, Chen Yang, Hao Zhou, Huatong Song, Zongchao Chen, Ran Le, Yang Song, Wayne Xin Zhao, and Tao Zhang. Evo-bench: Can language models improve agent harness? *arXiv preprint arXiv:2608.09096*, 2026.

Sihang Jiang, Lipeng Ma, Zhonghua Hong, Keyi Wang, Zhiyu Lu, Tengfei Wang, Shisong Chen, Jinghao Zhang, Tianjun Pan, Weijia Li, Jiaqing Liang, and Yanghua Xiao. Sea-eval: A benchmark for evaluating self-evolving agents beyond episodic assessment, 2026. URL https://arxiv.org/abs/2604.08988.

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

Kimi. Kimi K3: Open frontier intelligence, 2026. URL https://www.kimi.com/en/blog/kimi-k3.

Lambda. Gpu cloud pricing, 2026. URL https://lambda.ai/pricing. B200 cluster and instance pricing; accessed September 19, 2026.

Hyunin Lee, Jinglue Xu, Jeffrey Seely, Donghyun Lee, Matei Zaharia, and Yujin Tang. Recursive harness self-improvement. *arXiv preprint arXiv:2607.15524*, 2026a.

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

Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. Codei/o: Condensing reasoning patterns via code input-output prediction. *arXiv preprint arXiv:2502.07316*, 2025.

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

Fanqing Meng, Lingxiao Du, Qiguang Chen, Ziqi Zhao, Haocheng Lu, Mengkang Hu, and Michael Qizhe Shieh. Rsibench-data: Benchmarking data-centric research for recursive self-improvement, 2026. URL https://arxiv.org/abs/2607.25886.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

Joel Niklaus. Don't train the model, evolve the harness, 2026. URL https://www.alphaxiv.org/abs/2607.joelniklaus-harness-optimization-hf.

NVIDIA. Nvidia hgx platform, 2026. URL https://www.nvidia.com/en-us/data-center/hgx/. HGX B200 specifications; accessed September 19, 2026.

OpenAI. Introducing gpt-5.5, 2026a. URL https://openai.com/index/introducing-gpt-5-5/.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition, 2026b. URL https://openai.com/index/gpt-5-6/. Release July 9, 2026; accessed September 19, 2026.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023. URL https://arxiv.org/abs/2307.16789.

RSI-Exam Team. Rsi-exam: Benchmarking recursive self-improvement through executable research, 2026. URL https://github.com/aiming-lab/RSI-Exam.

Jürgen Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In *Artificial general intelligence*, pp. 199–226. Springer, 2007.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Hao Hu, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Chao Hong, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Lijun Lu, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Xinjie Sun, Flood Sung, Heyi Tang, Jiawen Tao, Qifeng Teng, Chensi Wang, Dinglu Wang, Feng Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Xiaofei Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yangkun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Huabin Zheng, Shaojie Zheng, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence, 2025. URL https://arxiv.org/abs/2507.20534.

Yuyao Wang, Zhongjian Zhang, Mo Chi, Kaichi Yu, Yuhan Li, Miao Peng, Bing Tong, Chen Zhang, Yan Zhou, and Jia Li. Evomembench: Benchmarking agent memory from a self-evolving perspective, 2026. URL https://arxiv.org/abs/2605.18421.

Tianxin Wei, Noveen Sachdeva, Benjamin Coleman, Zhankui He, Yuanchen Bei, Xuying Ning, Mengting Ai, Yunzhe Li, Jingrui He, Ed H. Chi, Chi Wang, Shuo Chen, Fernando Pereira, Wang-Cheng Kang, and Derek Zhiyuan Cheng. Evo-memory: Benchmarking llm agent test-time learning with self-evolving memory, 2026. URL https://arxiv.org/abs/2511.20857.

Lilian Weng. Harness engineering for self-improvement. *lilianweng.github.io*, July 2026. URL https://lilianweng.github.io/posts/2026-07-04-harness/.

Jundong Xu, Qingchuan Li, Jiaying Wu, Yihuai Lan, Shuyue Stella Li, Huichi Zhou, Bowen Jiang, Lei Wang, Jun Wang, Anh Tuan Luu, Caiming Xiong, Hae Won Park, Bryan Hooi, and Zhiyuan Hu. Evoarena: Tracking memory evolution for robust llm agents in dynamic environments, 2026a. URL https://arxiv.org/abs/2606.13681.

Zhangchen Xu, Junda Chen, Yue Huang, Dongfu Jiang, Jiefeng Chen, Hang Hua, Zijian Wu, Zheyuan Liu, Zexue He, Lichi Li, Shizhe Diao, Jiaxin Pei, Jinsung Yoon, Hao Zhang, Mengdi Wang, Radha Poovendran, Misha Sra, Alex Pentland, and Zichen Chen. Autolab: Can frontier models solve long-horizon auto research and engineering tasks?, 2026b. URL https://arxiv.org/abs/2606.05080.

Shuhan Xue, Zixin Ding, Yichen Shen, Yinjie Wang, Zhenfei Yin, Yingcheng Wu, Yuxin Chen, Mengdi Wang, and Ling Yang. Past-bench: Benchmarking the foundations of recursive self-improvement in personal agents, 2026. URL https://arxiv.org/abs/2608.04003.

Yongjin Yang, Jiarui Liu, Yinghui He, Lechen Zhang, Bernhard Schölkopf, and Zhijing Jin. Transferability for general reasoning: An automated curriculum for multi-domain rlvr, 2026. URL https://arxiv.org/abs/2606.25178.

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

Z.ai. GLM-5.3: Frontier coding with emergent cyber capabilities, 2026. URL https://z.ai/blog/glm-5.3.

Mario Zechner. Pi agent harness. https://github.com/earendil-works/pi, 2026.

Yuhao Zhan, Bingxiang He, Zecong Tang, and Chaojun Xiao. Pace-bench: Benchmarking physics adaptation via code evolution in dynamic environments, 2026. URL https://arxiv.org/abs/2608.14441.

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

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

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025b.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026b.

Congjie Zheng, Chuanyi Xue, Bin Liang, Jun Yang, and Changshui Zhang. Seagym: An evaluation environment for self-evolving llm agents, 2026. URL https://arxiv.org/abs/2606.17546.

Junhao Zheng, Xidi Cai, Qiuke Li, Duzhen Zhang, ZhongZhi Li, Yingying Zhang, Le Song, and Qianli Ma. Lifelongagentbench: Evaluating llm agents as lifelong learners, 2025. URL https://arxiv.org/abs/2505.11942.

Tailin Zhou. Hierarchical self-improvement: A framework for task-specific evolvable agent harnesses. *arXiv preprint arXiv:2608.08466*, 2026.

Deyao Zhu, Xin Zhou, Shengling Qin, Xuekai Zhu, Hangliang Ding, Shu Zhong, Zixin Wen, Zhonglin Xie, Chenhui Gou, Linxuan Ren, Yueyang Wang, Junfeng Zhong, Rui Liu, Tian Gao, Yangguang Lin, Jingyuan Zhang, Maojia Song, Xuan Qi, Jinhong Wu, Chenyang Zhang, Yinzhu Piao, Ziru Niu, Hongbin Lin, Lingxiang Meng, Peng Tang, Chengyao Tang, Shanyu Wu, Huanyu Zheng, Yu Liu, Liya Zhu, He Wang, Ming Ding, Ziyu Wan, Hao Liu, Sibo Wang, Haotian Zhu, Xintian Zhang, Nan Chai, Yipeng Liu, Panhao Lai, Sihang Yuan, Zixin Su, Ge Zhang, Wangchunshu Zhou, Yantao Du, Wenhao Huang, and Guang Shi. Edgebench: Unveiling scaling laws of learning from real-world environments, 2026. URL https://arxiv.org/abs/2607.05155.

A Benchmark Sources and Task Inventory

A.1 Benchmark Curation

We curate 100 tasks from four public agent benchmarks using a preregistered, model-calibrated pipeline. We first remove tasks with unsupported modalities, unreliable verifiers, incompatible infrastructure, excessive resource demands, or insufficient evidence that they are solvable. We then run a fixed panel of frozen models on the remaining candidates with the same initial Pi harness and budget. The screening outcomes are used to remove unsuitable or saturated tasks and to construct fixed, disjoint training and test pools spanning a range of observed difficulty.

We retain each benchmark's native environment and official verifier. The four domains contain 30 Terminal, 30 Environment Learning, 16 SWE, and 24 Science tasks. Their upstream benchmarks and train/test splits are fixed before recursive evaluation.

For each domain d, disjoint D^d_{train} and D^d_{test} pools have distinct roles. Training tasks provide all trajectories, revision evidence, and candidate-selection scores. No held-out test task is executed during recursive search, so test trajectories and scores cannot influence a proposal or inheritance decision. Only after all five generations terminate does the external runner access D^d_{test} for the first time and evaluate the initial and all unique accepted states. The test set remains unavailable to the controller until all revision rounds are complete.

B Extended Experimental Details

This appendix gives configuration details for the main experiments. The benchmark contains 100 tasks across Terminal, Environment Learning, SWE, and Science. Terminal uses Harbor-compatible environments and Environment Learning uses SForge; the other domains retain their native execution and verification stacks. Within every domain, training tasks provide trajectories and selection evidence. Held-out tasks are first accessed after the five-round revision process. Appendix A summarizes task curation and split sizes. The exact task identifiers remain in the accompanying source task list.

The primary comparison covers Claude Opus 5, Claude Opus 4.8, GPT-5.6-Sol, GPT-5.5, DeepSeek V4 Pro, Kimi K3, and GLM-5.3. Each model is run with a feature-rich frozen reference (Claude Code for the Claude models, Codex for the GPT models, and the Claude Code interface for DeepSeek, Kimi, and GLM) and with the editable Pi substrate. The pinned software versions are Claude Code 2.1.87, Codex 0.147.0, and Pi 0.82.1. The reference harnesses are evaluation-only; only Pi is recursively evolved.

All primary cells use three train rollouts and three test rollouts per task, five recursive generations, and training-side selection. Terminal and Environment Learning use 15/15 train/test tasks, SWE uses the resolved 8/8 formal subset, and Science uses 12/12. A numerical cell enters the paper only if all five generations finish, its result contains a final held-out score, the run has no unrecovered infrastructure failure, and the task, split, runtime, and evaluation-protocol records match.

Protocol. The task splits, verifiers, editable paths, rollout schedule, five-round budget, and training-side selection rule were fixed before held-out testing. Test scores did not determine whether a run was repeated or reported. The training-side comparison has one limitation: candidates are scored anew, but the current harness keeps the score measured when it was accepted. Rollout noise can therefore change a selection decision; these comparisons are not paired remeasurements of both harnesses under the same random draws.

The five-round main protocol therefore compares each candidate with the saved score of the incumbent at the time it was accepted. The 30-round extension is a separate diagnostic: its controller remeasures the current accepted incumbent within the same round before scoring the candidate. Thus the round-level deltas in Table 6 are paired within-round diagnostics and are not a replacement for the five-round selection rule.

Training-side selection rule. Let Q be the training tasks with valid scores in the gate. For each $q \in Q$, let $b_q, c_q \in [0, 1]$ be the current harness's saved mean score and the candidate's newly

measured mean score over k repeats. After structural and execution checks, the controller keeps a candidate only when all three conditions hold:

$$\begin{aligned} \text{higher training mean:} \quad & \frac{1}{|Q|}\sum_{q\in Q} c_q > \frac{1}{|Q|}\sum_{q\in Q} b_q, \\ \text{at least two improved tasks:} \quad & |\{q \in Q : c_q > b_q\}| \geq 2, \\ \text{no severe task regression:} \quad & \nexists\, q \in Q \text{ with } b_q = 1 \text{ and } c_q \leq 1/3. \end{aligned} \tag{6}$$

Otherwise, the current harness is used in the next round. Smaller task regressions can be outweighed by gains elsewhere. When one-repeat screening is enabled, the controller first tests a candidate once per training task. It rejects the candidate before the remaining repeats if three or more tasks that scored 1 in the current harness's first repeat score 0 for the candidate. Candidates that pass this screen receive their remaining repeats before the full rule above is applied.

Task-weighted main-table aggregation. Let $\mathcal{D}_m$ contain the domains with valid reported results for model m, let $n_d = |D^d_{\text{test}}|$, and let $N_m = \sum_{d\in\mathcal{D}_m} n_d$. The fixed test-task counts are 15 for Terminal, 15 for Environment Learning, 8 for SWE, and 12 for Science. Writing $\Delta^{\text{RSI}}_{m,d}$ in percentage points and $C_{m,d}$ for the archived end-to-end RSI cost in USD, Table 1 reports

$$\text{RSI Lift}_m = \frac{\sum_{d\in\mathcal{D}_m} n_d \Delta^{\text{RSI}}_{m,d}}{N_m}, \qquad \text{Cost}_m = \frac{\sum_{d\in\mathcal{D}_m} C_{m,d}}{N_m}. \tag{7}$$

Thus RSI Lift is the mean paired score change over individual held-out tasks, and Cost is total experimental expenditure per held-out task. Equivalently, the latter weights each domain's per-task cost $C_{m,d}/n_d$ by n_d; domain total costs are summed once, not multiplied by task counts. The cost numerator uses the archive's `whole_experiment_cost_usd`: meta-agent, round task-agent, and held-out evaluation costs, including the archive's flagged pricing estimates. It is not the cost of a single rollout with the final harness. Three repeated rollouts are averaged within each task before scoring and do not triple n_d. All seven reported models cover the same four domains, so $N_m = 50$ for every row in Table 1. Aggregation uses unrounded source values, with lift and USD/test-task cost rounded to two decimal places only for display. The source values and calculation are included in the accompanying artifacts.

Cross-domain score sources. The cross-domain tables use the reported score for each frozen-harness transfer run and compare it with the initial Pi harness on the same target domain. Matched-domain cells use the corresponding final score from the main experiment; separate repeat evaluations are not substituted for them. Some diagnostic runs are not comparable and are excluded from separate diagnostic summaries. The transfer scores are reported from aggregated evaluation results, not from a new audit of every task rollout. The three complete model matrices used for the cross-domain figure and gain summary contain all twelve off-diagonal cells. For Claude Opus 4.8 on Terminal, the primary score is 28.89re-evaluation of the same frozen harness reports 44.44the main comparison and treat the second value as a repeat measurement.

Baselines for transfer comparisons. Each reported model has up to 16 source-domain and target-domain pairs. In Terminal, Environment Learning, SWE, and Science order, the initial Pi scores are 33.33 / 21.86 / 50.00 / 30.56 for DeepSeek, 28.89 / 30.57 / 70.83 / 16.67 for Opus 4.8, 33.33 / 34.73 / 66.67 / 22.22 for Opus 5, and 42.22 / 26.78 / 37.50 / 11.11 for GPT-5.5. Diagnostic runs that cannot be compared under the same conditions are excluded. An unobserved score is never replaced with zero.

Single-surface score sources. Table 5 reports each final harness score's difference from that model's main-experiment baseline. If no revision was accepted, the table reports zero by convention. These differences are not the run's paired RSI lift. A separate route-level comparison uses the same starting score for GPT-5.6-Sol and a route-level mean for other models.

Recursive-horizon provenance. The GPT-5.6-Sol and Opus 5 Terminal experiments each used a 30-round schedule. Both experiments ran all 30 rounds: GPT-5.6-Sol accepted rounds 3 and 4 at $936.79 cumulative cost, and Opus 5 accepted no round at $1,319.25 cumulative cost, so its final

harness is H_0. Held-out scoring is applied to H_0 and H_T only, so the per-round values recorded in these run directories are training-side scores and cannot populate held-out checkpoint values.

The Pi editable boundary contains 18 physical modules: six shared-resource modules (including prompts, skills, rules, hooks, MCP configuration, and workflows) and twelve path-owned core-source modules. Ownership, path hashes, and the structural/build/smoke checks are specified in Appendix C. Model weights, task definitions and splits, official verifiers, execution environments, permissions, tool versions, and controller-side budgets remain immutable. All surface, controller, and horizon ablations use Terminal, whose execution and verification loop is the most controlled.

In the notation of Section 3.1, the experimental instantiation may be written as

$$A_t^{\mathrm{Pi}} = (M, H_t^{\mathrm{Pi}}, L_t, B_t), \qquad H_t^{\mathrm{Pi}} = (R_t^{\mathrm{Pi}}, C_t^{\mathrm{Pi}}). \tag{8}$$

Here H_t^{Pi} is the mutable runtime and resource layer and M remains frozen. The capitalized product name "Pi" is distinct from π_M, the model-mediated revision policy in Section 3.1.

C Harness Editable Surface and Module Inventory

Operational boundary. The editable surface used in the recursive experiments has two disjoint layers. Layer 1 contains six cross-agent shared resources: Prompt, Rules, Skills, Hooks, MCP, and Workflow. Layer 2 contains twelve path-owned Pi core-source modules derived from the pinned Pi v0.82.1 source tree. Every accepted checkpoint stores a manifest, module hashes, and a Git diff. A candidate must pass structural and physical-ownership guards, rebuild where required, native loader checks, and cold-start checks before it can enter full-training evaluation.

The boundary excludes model weights, benchmark instances and splits, official verifiers, sandbox images, permissions, and controller-side token, time, and tool budgets. These components remain frozen so that a score change can be attributed to harness revision rather than to a change in the model or the evaluation environment.

Physical module inventory. Table 3 is the authoritative editable-surface registry, not a score table. It records the files owned by each module and the runtime responsibility of those files. At run initialization, every glob is expanded against the pinned Pi source tree and the resulting path set is hashed. Each editable byte must resolve to exactly one owner; overlapping, unowned, or out-of-bound paths fail the structural guard before evaluation. Module names proposed by the reviser are descriptive only. The Git diff and this physical path registry determine which module changed.

D Recursive Harness and System Formalization

We formalize the recursive process evaluated by RSIBench. Repeatedly modifying a system does not by itself make the process recursive, just as a single performance gain does not establish RSI. The defining property is whether an inherited system state causally re-enters the process that produces the next update. The recursive process has two requirements: **(i) Inherited system state.** The system must preserve modifications across generations, allowing each successor to build on the state produced by its predecessors. **(ii) Recursive transition.** The inherited successor must participate in generating the experience and revision that produce the next successor. Whether that process improves capability on tasks outside revision and selection is a separate empirical outcome.

We represent the system state at generation t as

$$A_t = (M, H_t, L_t, B_t), \qquad H_t = (R_t, C_t), \tag{9}$$

where M is the frozen foundation model, H_t is the executable agent harness, L_t records earlier proposed harness edits and whether they were accepted or rejected, and B_t is the remaining controller budget. The harness comprises shared project resources R_t, such as prompts, skills, rules, hooks, MCP configurations, and workflows, and core source code C_t. The external configuration Ω, including task definitions, data splits, evaluator, sandbox, permissions, tool versions, and controller limits, remains fixed. A proposal may modify only a preregistered editable set $\mathcal{E}$ of paths within (R, C).

Table 3: Editable modules and their owned paths in the pinned Pi source tree.

Layer	Module	Responsible function	Owned file(s) or path family
Layer 1: shared project resources (6 modules)			
L1	Prompt	Project-level instructions and persistent system guidance	`CLAUDE.md`; `AGENTS.md`; `.pi/APPEND_SYSTEM.md`
L1	Rules	Composable policies, constraints, and scoped instructions	`.claude/rules/**`; `.codex/rules/**`; `.pi/rules/**`
L1	Skills	Reusable domain procedures and packaged knowledge	`.claude/skills/**`; `.agents/skills/**`; `.pi/skills/**`
L1	Hooks	Lifecycle-triggered scripts and event handlers	`.claude/hooks*`; `.codex/hooks*`; `.pi/hooks/**`
L1	MCP	External tool and service-adapter configuration	MCP configuration paths
L1	Workflow	Agent roles, commands, prompts, and orchestration	`agents/**`; `commands/**`; `prompts/**`
Layer 2: Pi core source (12 path-owned modules)			
L2	Agent Loop	Turn loop, tool-call cycles, stopping, cancellation, and retry	`agent-loop.ts`; `agent.ts`; `stream-fn.ts`
L2	Harness Runtime	Harness assembly, dependency binding, and environment lifecycle	`packages/agent/src/harness/agent-harness.ts`; `env/**`; `types.ts`
L2	Coding Session	Session orchestration, events, resume behavior, and SDK entry points	`core/agent-session*.ts`; `agent-session-services.ts`; `sdk.ts`
L2	Observation	Tool-result fidelity, error representation, accumulation, and truncation	Both packages' `messages.ts`; shell `output.ts`/`truncate.ts`; coding-agent output accumulator/truncation paths
L2	Context	Legal provider-history construction and request handoff	`model-runtime.ts`; `provider-composer.ts`
L2	Compaction	Conversation summaries, overflow handling, and context recovery	`compaction/**` in the harness and coding-agent packages
L2	Session Storage	Persistence, working-directory state, loading, and recovery	`harness/session/**`; `session-manager.ts`; `session-cwd.ts`
L2	Prompt Loader	Prompt discovery, ordering, escaping, and assembly	Both packages' `system-prompt.ts`; `prompt-templates.ts`; `cli/initial-message.ts`
L2	Skill Loader	Skill/resource discovery, filtering, parsing, and loading	Both packages' `skills.ts`; `resource-loader.ts`
L2	Tool Runtime	Tool schemas, execution, mutation ordering, and shell dispatch	Both packages' `tools/**`; `bash-executor.ts`; `exec.ts`
L2	Extension Runtime	Extension loading, lifecycle events, and event-bus dispatch	`core/extensions/**`; `event-bus.ts`; `extensions/**`
L2	Provider Runtime	Request mapping, streaming, tool calls, usage, and provider errors	`packages/ai/src/api/**` registered by the Source-12 allowlist

At generation t, the current state uses its executable system and accumulated training experience to propose an update:

$$\Delta_t \sim \pi_M(\cdot \mid A_t, D_{\text{train}}, \mathcal{T}_{\le t}; \Omega), \qquad \widetilde{A}_{t+1} = U(A_t, \Delta_t), \tag{10}$$

where π_M denotes the model-mediated revision policy and U applies the proposed update to the current state. A protocol-specific inheritance operator then determines the next state:

$$A_{t+1} = \mathcal{I}_t\left(A_t, \widetilde{A}_{t+1}\right). \tag{11}$$

The choice of $\mathcal{I}_t$ belongs to the experimental protocol and is not part of the definition of recursion.

The process becomes recursive when the inherited successor is used both to execute the next generation of tasks and to produce the next revision:

$$\Delta_{t+1} \sim \pi_M(\cdot \mid A_{t+1}, D_{\text{train}}, \mathcal{T}_{\le t+1}; \Omega). \tag{12}$$

Recursion alone does not guarantee capability gains. We therefore measure improvement separately on held-out tasks excluded from revision and inheritance decisions.

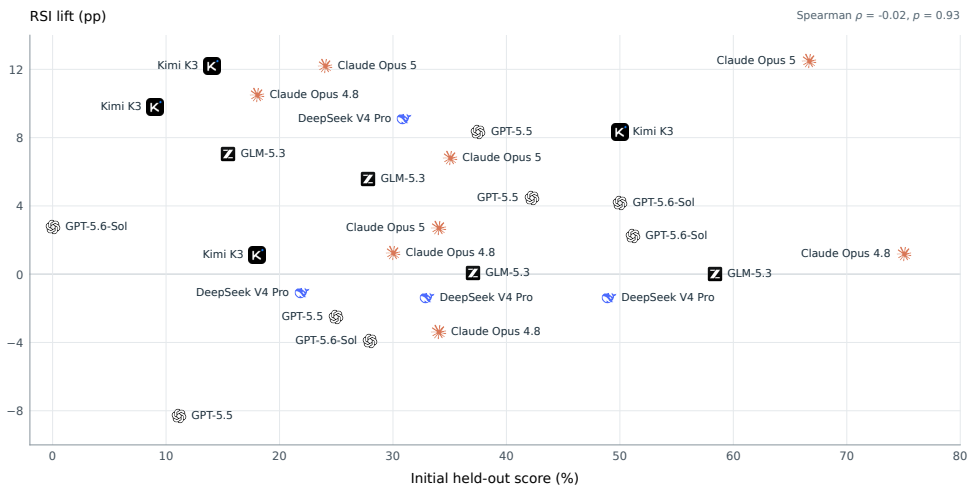

Figure 5: **Initial competence does not recover the RSI ranking.** The 28 observations correspond to the seven models and four domains in Table 1. The horizontal axis is the initial held-out score and the vertical axis is held-out RSI lift. Each observation is represented by a model icon with an adjacent label. Spearman correlation across all 28 observations is $\rho = -0.02$ (two-sided $p = 0.93$).

E Additional Main-Experiment Analysis

This appendix gives additional results and calculations for the main experiments.

GLM-5.3 on Science. The run accepts a revision in round 1 and improves from 27.78 to 33.33 on held-out Science tasks, a gain of 5.56 percentage points. Table 1 and Figure 5 include this result.

Kimi K3 on Environment Learning. The run accepts revisions in rounds 1, 3, and 5 and improves from 10.81 to 20.98 on held-out tasks, a gain of 10.17 percentage points. Table 1 includes this result.

Training-gate audit. The 25-run Pi audit covers 125 candidate rounds, of which 25 were accepted. Summed training improvement is 145.3 percentage points, compared with 87.0 points on held-out tasks, an aggregate retention ratio of 0.60.

E.1 Adaptation Cost and Training Workload

The five-round Pi experiments record the cost of harness search, task execution, and held-out evaluation. We compare these observed costs with a hypothetical model-weight update using an assumed number of trajectories and measured trace lengths. Appendix E.7 gives the workload, compute assumptions, and sensitivity scenarios. No post-trained model was evaluated on the same tasks, so the calculation does not establish which approach reaches a given held-out score at lower cost.

E.2 Cross-Domain Transfer Diagnostics

The transfer matrix compares each frozen evolved harness with the initial Pi harness on the target domain. For GPT-5.6-Sol, the target baselines are 51.11, 26.86, 50.00, and 0.00 percent for Terminal, Environment Learning, SWE, and Science. The Terminal and Science baselines come from the validated September 19 runs; earlier scores from replaced runs are not used. Environment scores range from 26.12 to 27.49, and Science scores from 0.00 to 5.56. With a Science baseline of 0.00, no displayed Science cell has a negative difference. Every off-diagonal Terminal score is below its target baseline.

Each four-domain matrix has 16 possible source–target pairs. The coefficient averages reported off-diagonal differences; the three-domain version uses Terminal, Environment Learning, and Science. The three complete matrices in the versioned snapshot provide all twelve four-domain off-diagonal cells; positive counts use the displayed scores.

Table 4: Cross-domain gain G (Equation 5) in percentage points. Common-three excludes SWE; counts cover the twelve off-diagonal cells for each model.

Model	Four-domain G (pp)	Common-three G (pp)	Positive / observed
GPT-5.6-Sol	-0.894	-1.297	5/12
DeepSeek V4 Pro	-10.551	-4.777	0/12
Claude Opus 4.8	$+7.159$	$+9.926$	8/12

Table 5: Single-module evolution on Terminal after five training-selected rounds. Each entry is the final held-out score minus that model's fixed main-experiment baseline, in percentage points. A zero denotes a run with no accepted revision and is assigned by convention.

Editable module	**GPT-5.6-Sol**	**Claude Opus 5**	**GLM-5.3**	**DeepSeek V4 Pro**
Main-experiment A_0 (%)	51.11	33.33	35.56	33.33
Prompt	-2.22	$+28.89$	0.00	-4.44
Agent Loop	$+2.22$	$+13.33$	$+8.89$	0.00
Tool Runtime	-2.22	0.00	0.00	$+4.44$
Observation	0.00	$+24.44$	0.00	$+8.89$
Context + Extension	$+2.22$	$+24.44$	0.00	-11.11
Compaction	$+4.44$	$+17.78$	0.00	0.00
Skills + Skill Loader	-2.22	$+31.11$	$+15.56$	-2.22
Hooks	$+11.11$	$+24.44$	0.00	0.00

All coefficients give equal weight to ordered domain pairs, rather than to the unequal numbers of tasks per domain. The comparison measures improvement over each model's own baseline, not absolute competence. It is an additive gain, not a fraction of in-domain improvement retained: division by an in-domain gain would be unstable when that gain is zero or negative. SWE diagonals contribute to neither coefficient because all diagonals are excluded. Missing or shake cells are never imputed. Scores and baselines were transcribed to four decimal places, so derived coefficients are approximate; no confidence interval or significance claim is inferred from aggregate display values.

On the diagonal, we use the final score reported in Table 1. Separate reevaluations of the same frozen harness are repeat measurements, not additional cells in the transfer average. Off-diagonal cells come from the frozen-harness transfer evaluations. Because not every cell has a fresh paired baseline and task-level repeats, G describes the reported transfer scores; it is not a causal estimate or a statistically established model ranking.

For Claude Opus 4.8 on Terminal, the main-experiment final score is 28.89, while a separate evaluation of the same frozen harness reports 44.44. Appendix B distinguishes these measurements. Their 15.56-point difference is variation between evaluations, not an RSI gain; the primary 28.89 remains the diagonal score.

Post-training changes model weights, while harness evolution changes the execution system around fixed weights. A matched comparison using the same models and tasks could test whether the two interventions benefit the same target domains. Different outcomes would motivate tests of whether a harness edit encodes a task-specific procedure or a more general way to use context, tools, and verification. Our transfer matrix does not test that mechanism.

E.3 Single-Surface Coverage and Baseline Qualifications

Table 5 reports final-score differences from each model's main-experiment baseline. GPT-5.6-Sol scores $+11.11$ points with Hooks, $+4.44$ with Compaction, and $+2.22$ with Agent Loop. Opus 5 scores $+28.89$ with Prompt and $+31.11$ with Skills + Skill Loader; GLM-5.3 scores $+15.56$ with Skills + Skill Loader. These are fixed-baseline differences.

All displayed single-module differences use the fixed main-experiment baseline. GPT-5.6-Sol accepts Hooks edits in rounds 1 and 4 and a Compaction edit in round 1; Opus 5 accepts a Prompt revision in round 3; and GLM-5.3 accepts a Skills + Skill Loader revision in round 2. The Opus 5 Agent Loop entry compares its final score of 46.67 with the main-experiment Terminal baseline of 33.33, giving $+13.33$ pp. Independent evaluations of that run's initial and final harnesses score 53.33 and 46.67, respectively, a -6.67-point change.

An earlier route-level comparison used the main-experiment baseline for GPT-5.6-Sol and a same-route mean for other models. On that scale, Opus 5 Prompt is $+3.33$ and GLM-5.3 Skills + Skill Loader is $+11.75$. These differences use neither the new table's common baseline nor each run's own starting score.

E.4 Recursive-Horizon Protocol and Progress

We extend GPT-5.6-Sol + Pi and Claude Opus 5 + Pi on Terminal from five to 30 rounds under the same task split, per-round budget, and training-side selection rule. The test tasks are not used during search. We evaluate H_0 and the final harness on held-out tasks, but do not have held-out scores for intermediate rounds. The two runs therefore test whether the longer search finds additional accepted revisions; they do not provide a held-out learning curve.

Both runs reached round 30. GPT-5.6-Sol accepted revisions in rounds 3 and 4, spent \$936.79, and improved from 48.89 to 53.33 on held-out tasks. Opus 5 accepted no revision, spent \$1,319.25, and ended at its starting score of 66.67. Under this fixed search policy, additional rounds did not produce further accepted revisions in either run.

Table 6 reports the training scores and recorded selection decisions for each round of the GPT-5.6-Sol run.

E.5 Detailed Limitations

The held-out test set is finite and reused across reported states. Test outcomes do not drive proposal, acceptance, or inheritance: the test split is first queried after revision ends to score the final inherited harness H_T. The five-round main score compares H_0 with H_T. Separate post-search diagnostics can also evaluate accepted intermediate harnesses, but do not use their scores to select H_T. The 30-round extension evaluates only H_0 and its final harness. Comparisons among tested intermediate states remain descriptive because they reuse the same finite task pool. The cross-domain study reports complete 4×4 matrices for three models on the shared Pi substrate. It therefore does not characterize every model and harness interaction. A larger release should add independent test suites and repeat the full search under multiple seeds.

Optimization-run variance is only partially measured. Three rollouts per task reduce within-task sampling noise. A single recursive run with a fixed schedule does not characterize variance across proposal trajectories or schedule seeds.

Coverage and curation are selective. The 100 primary tasks span four heterogeneous domains, but they are filtered for runtime, verifier validity, modality support, and pilot difficulty. These choices improve executability while changing the evaluated distribution. Results need not transfer to multi-modal tasks, very long-running environments, or tasks saturated by the screening models.

Harness comparisons are not perfectly symmetric. Claude Code, Codex, and Pi do not share identical internal architectures, and Claude Code is a non-native interface for DeepSeek, Kimi, and GLM. We therefore interpret pairwise results as comparisons between concrete model and harness systems. Only Pi is evolved; the mature systems are fixed references, and fine-grained attribution is limited to Pi's registered 18-module boundary.

Long-horizon trends are exploratory. Our longest experiment contains 30 recursive rounds, which is still far too short to establish open-ended improvement. One run can reveal finite-budget saturation, transient peaks, or rising cost per accepted update, but it cannot establish a universal RSI scaling law.

Table 6: **Round-by-round training outcomes for 30-round RSI.** GPT-5.6-Sol with Pi on Terminal. Candidate and incumbent scores are on a 0–1 scale; Δ is the candidate score minus the incumbent score within the same round. Only rounds 3 and 4 were accepted.

Round	Candidate	Incumbent	Δ	Decision	Recorded reason
R1	0.711	0.711	0.0000	Reject	No improvement
R2	0.667	0.711	−0.0444	Reject	Task-level regression veto
R3	0.756	0.711	+0.0444	**Accept**	Accepted revision
R4	0.800	0.756	+0.0444	**Accept**	Accepted revision
R5	0.800	0.800	0.0000	Reject	No improvement
R6	0.667	0.800	−0.1333	Reject	Pass-1 screening failure
R7	0.756	0.800	−0.0444	Reject	No improvement
R8	0.778	0.800	−0.0222	Reject	Task-level regression veto
R9	0.733	0.800	−0.0667	Reject	Pass-1 screening failure
R10	0.667	0.800	−0.1333	Reject	Pass-1 screening failure
R11	0.756	0.800	−0.0444	Reject	Task-level regression veto
R12	0.778	0.800	−0.0222	Reject	Task-level regression veto
R13	0.689	0.800	−0.1111	Reject	Task-level regression veto
R14	0.778	0.800	−0.0222	Reject	No improvement
R15	0.756	0.800	−0.0444	Reject	Task-level regression veto
R16	0.800	0.800	0.0000	Reject	Task-level regression veto
R17	0.600	0.800	−0.2000	Reject	Pass-1 screening failure
R18	0.778	0.800	−0.0222	Reject	No improvement
R19	0.733	0.800	−0.0667	Reject	Pass-1 screening failure
R20	0.600	0.800	−0.2000	Reject	Pass-1 screening failure
R21	0.756	0.800	−0.0444	Reject	Task-level regression veto
R22	0.800	0.800	0.0000	Reject	Task-level regression veto
R23	0.711	0.800	−0.0889	Reject	Task-level regression veto
R24	0.600	0.800	−0.2000	Reject	Pass-1 screening failure
R25	0.689	0.800	−0.1111	Reject	Task-level regression veto
R26	0.756	0.800	−0.0444	Reject	Task-level regression veto
R27	0.756	0.800	−0.0444	Reject	Too few improved tasks
R28	0.600	0.800	−0.2000	Reject	Pass-1 screening failure
R29	0.533	0.800	−0.2667	Reject	Pass-1 screening failure
R30	0.733	0.800	−0.0667	Reject	Task-level regression veto

Notes. The rule after R5 separates the five-round protocol prefix from the additional rounds. In this 30-round diagnostic, the incumbent is the current accepted harness remeasured within the same round, not the score recorded at its last acceptance. This differs from the five-round main protocol, which uses the saved incumbent score for selection. Scores are shown to three decimal places; differences retain four decimal places from the recorded results. Decisions and reasons are transcribed from the controller record. Pass-1 screening failure denotes rejection before pass-3 evaluation; the task-level regression veto can reject a candidate even when its aggregate score ties the incumbent. Acceptance uses training data only; held-out scores are reported separately in the accompanying text.

The direct gate does not protect every task. Equation 6 requires a higher training mean, gains on at least two tasks, and no drop from 1 to at most $1/3$ on any task. Smaller regressions can still be outweighed by gains elsewhere. We report the full per-task score changes and regression counts; the number of improved tasks beyond the required two is diagnostic. Direct mode uses no task-specific anchors or post-validation review. The gate never uses held-out scores, so held-out domain regressions remain possible.

E.6 Reproducing the Reported Results

The accompanying task list records the 100 task identifiers and their training or test assignments. Appendix B gives the model and harness versions, rollout counts, selection rule, and cost aggregation. Run-level records used for the analysis include the resolved configuration, task outcomes, proposed harness changes, and acceptance decisions. The initial and final held-out scores use the same test tasks and three rollouts per task. Test results do not enter the training-side selection rule.

Not every raw trajectory archive is contained in this manuscript repository. Reproducing the closed-model runs also requires access to the model APIs and compatible task environments. The reported cost estimates and their assumptions are described in Appendix E.7.

E.7 What Does Training for One Leaderboard Cost?

How many trajectories, and how long is each one? For a leaderboard with N_b test tasks, we approximate the adaptation corpus as $20N_b$ agent trajectories. Let L_b denote mean tokens in one complete agent episode. The training-token estimate for one epoch is simply

$$R_b = 20N_b, \qquad T_b = R_b L_b = 20N_b L_b. \tag{13}$$

Twenty is an assumed number of training trajectories per benchmark task, not twenty epochs. The benchmark's test-set size sets the scale of this hypothetical corpus; this calculation is not an actual training run or a claim that training on evaluation tasks would preserve a valid held-out comparison.

We calibrate L_b from 69 initial H_0 GPT-5.6-Sol/Codex 0.147.0 training attempts: 15 Terminal tasks with three attempts each and eight DeepSWE tasks with three attempts each. All initial attempts are included, including unsuccessful attempts and one DeepSWE agent timeout at 3,600 seconds with valid verifier evaluation. Each trajectory's per-call input, output, and cache-read totals exactly reconcile to its initial training-run cost record. We exclude test, meta-agent, and evolved candidate traces.

We estimate episode length by the last recorded prompt plus completion: 32,571.93 tokens for Terminal and 100,118.71 for DeepSWE. These traces each contain one Codex turn with no observed decrease in prompt length. This is a measured proxy for the episode's training sequence, not an exact reconstruction of the training serializer; hidden reasoning and context transformations can change the actual length. Crucially, we do not sum all API prompts: doing so would repeatedly count the growing conversation prefix. API cache discounts also do not reduce training FLOPs. The observed means come from our small subsets and are extrapolated to the full benchmarks below.

How many tasks are on each leaderboard? The public Harbor registry lists 89 tasks for Terminal-Bench 2.0; the DeepSWE repository documents 113 tasks and its v1.1 verifier protocol. We therefore use $N_T = 89$ and $N_S = 113$, giving 1,780 and 2,260 hypothetical training trajectories. The Terminal count is explicitly for 2.0, not the 2.1 version named on the GPT-5.6 release page. These full-benchmark counts replace our 15/8-task calibration-pool sizes in the cost calculation.

What does one training epoch cost for a 3T dense model? For a hypothetical dense model with $P = 3 \times 10^{12}$ parameters, we use

$$F_b = 6PT_b, \qquad G_b = \frac{6P(20N_b L_b)}{3600 f \eta}, \qquad C_b = pG_b. \tag{14}$$

Here G_b is B200 GPU-hours, $f = 2.25 \times 10^{15}$ dense BF16 FLOP/s per GPU, $\eta = 0.40$ is assumed training utilization, and $p = \$8.87$/GPU-hour. NVIDIA lists 36 PFLOP/s sparse BF16 per eight-GPU HGX B200; halving for dense compute and dividing by eight gives f (NVIDIA, 2026). The

price is Lambda's published 256+ B200 cluster rate with a minimum two-week reservation (Lambda, 2026). At these settings, a 3T dense update processes 180,000 training tokens per GPU-hour in the approximation.

Table 7: Estimated one-epoch weight-update cost for a hypothetical 3T model; generation and validation are excluded.

Benchmark	Tasks	Traj.	Tokens/traj.	Train tokens	B200 h	USD
Terminal-Bench 2.0	89	1,780	32.57K	57.98M	322.10	2,857
DeepSWE v1.1	113	2,260	100.12K	226.27M	1,257.05	11,150
Combined	202	4,040	—	284.25M	1,579.15	14,007

Thus training for Terminal-Bench 2.0 costs approximately $2,857 and training for DeepSWE costs $11,150. Their sum is 284.25M tokens, 1,579.15 B200 GPU-hours, and $14,007. This is the cost of one full-weight update epoch over the assumed corpus, not pretraining a 3T model from scratch. Generation, selection, validation, final evaluation, and tools are additional costs. Attention, communication, padding, recomputation, and engineering are not explicitly modeled. Multiply by the number of epochs if the same corpus is reused; changing trajectory count or episode length scales the estimate linearly. We do not attribute the hypothetical 3T architecture to GPT-5.6.

Would an open model require the same training expenditure? Holding the episode workload and utilization fixed, replace P by the active parameter count per token. A generic 70B dense model gives approximately $67/$260 for Terminal/DeepSWE. The published DeepSeek-V3 architecture (671B total, 37B active) gives $35/$138, and Kimi K2 (1T total, 32B active) gives $30/$119 (DeepSeek-AI, 2024; Team et al., 2025). These are architecture examples, not the V4 Pro and K3 checkpoints in our main table. Their actual trajectory lengths and adaptation success may differ.

The dollar values price consumed reserved capacity rather than an isolated job invoice. Full-weight mixed-precision training state at about 16 bytes per total parameter requires roughly 48 TB for a 3T model before activations; MoE similarly retains total-parameter memory costs. GPU-hours therefore do not imply that a model fits on one GPU or predict elapsed job time.

Does adapting the harness cost less at equal quality? The estimate isolates the weight-update expenditure avoided by freezing the model. It does not prove that the assumed training corpus achieves successful adaptation, or that harness adaptation always wins on total cost. A matched comparison must charge both directions for generation, search, and validation and measure cost and time to the same held-out quality. Repeating after a harness update would test the recurring burden.

What does the same calculation give for each model in our table? Table 8 evaluates the formula above once per model and sets it beside that model's archived RSI expenditure. Three design choices make the two sides comparable, and one residual asymmetry does not go away.

First, the scope is Terminal and SWE only. Post-training estimates require a published full-benchmark task count, which we have for Terminal-Bench 2.0 and DeepSWE v1.1 but not for Environment Learning or Science. Restricting both columns to Terminal and SWE gives every row the same 23 held-out tasks. The RSI costs in Table 8 are therefore two-domain subtotals, whereas Table 1 reports cost per held-out task across all four domains. For the same reason the lift column is a 23-task weighted mean and differs from RSI Lift; Opus 4.8 and GLM-5.3 are flat on both of these two domains, so they read 0.00 here.

Second, the corpus is a property of the benchmarks rather than of the model: $20\times(89+113) = 4{,}040$ trajectories and 284.25M tokens for every row. Only the active parameter count varies, so the GPU-hour and dollar columns are a linear rescaling of one another and rows with equal active parameters are necessarily identical.

Table note. The common post-training workload is 4,040 trajectories and 284.25M tokens (one epoch; 20 trajectories per task). The workload is fixed, so Tokens is reported once in Table 8 rather than repeated per model. B200 h and Cost are weight-update estimates; the merged closed-model cells are one shared 300B-active scenario, not four independent model measurements. The † entries

Table 8: Estimated post-training and archived RSI costs on 23 held-out tasks; closed-model rows share one hypothetical active-parameter count.

	Post-training (estimated)				RSI (archived)		
Model	**Params (tot./act.)**	**B200 h/update**	**Cost**	**Lift**	**Cost**	**Lift**	**Cost ratio**
Closed-model scenario: shared 300B-active placeholder							
Claude Opus 5	— / 300B†	157.91	$1,401	*n/m*	$1,899	+10.14	0.74×
Claude Opus 4.8	— / 300B†	157.91	$1,401	*n/m*	$1,530	0.00	0.92×
GPT-5.6-Sol	— / 300B†	157.91	$1,401	*n/m*	$1,369	+2.90	1.02×
GPT-5.5	— / 300B†	157.91	$1,401	*n/m*	$1,351	+5.80	1.04×
Open-weight configuration reported; active parameters used as published							
GLM-5.3	743B / 39B	20.53	$182	*n/m*	$4,131	0.00	0.04×
Kimi K3	2.8T / 104B	54.74	$486	*n/m*	$844	+2.90	0.58×
DeepSeek V4 Pro	1.6T / 49B	25.79	$229	*n/m*	$57	+5.80	4.03×

are hypothetical placeholders, not disclosures or attributions. The four closed-model rows share one assumed active-parameter count, rather than representing four independent estimates. Only active parameters per token enter the $6PT$ approximation; total parameters are shown for context and govern optimizer state, not update FLOPs. GPU-hour and cost columns are identical across rows with equal active parameters because the assumed corpus does not depend on the model.

Third, we decline to invent parameter counts. DeepSeek's release note states 1.6T total with 49B active for V4 Pro; Kimi K3 is published at 2.8T total with 16 of 896 experts routed per token, reported as roughly 104B active; Z.ai states GLM-5.3 reuses the GLM-5.2 base with all gains from post-training, and its released checkpoint is 753B including the multi-token-prediction layer and 743B without it, activating roughly 39B per token (about 22.6B routed plus 16.7B always-on, from 256 routed experts plus one shared at top-8 routing). That last figure comes from inspecting the released checkpoint rather than from a vendor parameter statement. Anthropic and OpenAI publish no parameter count for any Claude or GPT model. Rather than import a rumoured figure, we give those four rows one shared hypothetical 300B-active placeholder, marked †, so that their identical entries are legible as a single assumption instead of four coincident estimates. The comparison for those rows is therefore a scenario, and the open-weight rows should be read as the better-grounded ones. Because P enters $6PT$ linearly, any reader who prefers a different placeholder can rescale: the combined 284.25M-token epoch costs $182.09 at 39B active, $228.78 at 49B, $485.58 at 104B, $1,400.70 at 300B, $4,669.01 at 1T, and $14,007.03 at 3T.

The residual asymmetry is that the two cost columns do not cover the same activities. The RSI column is archived end-to-end expenditure: meta-agent, per-round task-agent, and held-out evaluation, including the archive's flagged pricing estimates. The post-training column is weight-update compute only, and excludes the trajectory generation, filtering, and evaluation that any real post-training pipeline would also pay for. The estimated post-training cost thus runs from $0.04\times$ the RSI cost (GLM-5.3) to $4.03\times$ it (DeepSeek V4 Pro), and its direction is not even constant across models. Most importantly the post-training lift column is *not measured*: we price a training run we did not perform, so the table reports no held-out gain for that side and no row supports an equal-quality savings claim. What it does establish is the order of magnitude, a few hundred to a few thousand dollars on both sides for these benchmark sizes, and that the ranking depends on which model is asked.

What is the full scenario cost for a 10T dense model? The main-text comparison uses a hypothetical 10T dense model. Its parameter count is a sensitivity scenario, not a disclosed size for GPT-5.6-Sol or Claude Opus 5. Applying the same $6PT$ approximation to the 284.25M-token corpus gives 5,263.82 B200 GPU-hours, or approximately $46,690 for one weight-update epoch at $8.87 per GPU-hour. The 10T scenario requires roughly 160 TB for full-weight mixed-precision training state at 16 bytes per parameter before activations. Total parameters determine this memory requirement; active parameters per token determine the simplified update FLOPs for an MoE model. The estimate prices consumed GPU-hours, not a minimum cluster rental or an actual distributed training job.

Table 9: Observed five-round Pi RSI and estimated post-training costs for a 10T-parameter dense model on Terminal and SWE.

Setting	Cost	Held-out lift
GPT-5.6-Sol + Pi RSI	\$1,369	+2.90 pp
Claude Opus 5 + Pi RSI	\$1,899	+10.14 pp
Trajectory generation proxy	\$10,993	–
10T dense weight update, one epoch	\$46,690	–
Post-training subtotal	**\$57,683**	–

The post-training subtotal excludes filtering, validation, and final evaluation. Generation cost extrapolates the archived GPT-5.6-Sol/Codex reference; the weight update assumes 4,040 trajectories, 284.25M tokens, 40% B200 utilization, and \$8.87 per GPU-hour. See Appendix E.7.

We separately estimate the cost of creating the 4,040 trajectories. The archived frozen GPT-5.6-Sol/Codex reference costs \$47 for 45 Terminal rollouts and \$97 for 24 SWE rollouts (Table 10). Extrapolating those rounded costs gives

$$C_{\text{generate}} = 1{,}780\frac{47}{45} + 2{,}260\frac{97}{24} \approx \$10{,}993. \tag{15}$$

The Terminal and SWE components are approximately \$1,859 and \$9,134. Adding generation and one weight-update epoch gives a subtotal of about \$57,683. This proxy assumes the same API, harness, task mix, and per-rollout cost when scaled to the full leaderboards. It excludes filtering, reward verification, final evaluation, tool infrastructure, and any additional optimization epochs. The 256+ B200 cluster price used here carries a two-week minimum reservation (Lambda, 2026); the charged invoice for a short, dedicated training job can exceed the consumed-compute subtotal.

Table 9 places that hypothetical subtotal beside the archived two-domain, five-round Pi RSI costs. The RSI rows include meta-agent search, task rollouts, and held-out evaluation and use the same 23-task Terminal–SWE denominator. Their observed lifts are not post-training lifts. The 10T model is not attributed to either closed model in the table, and its held-out quality after training is unknown. Thus the table gives a transparent cost scenario for weight adaptation and a measured cost and outcome for harness adaptation; it does not establish the cost of equal-quality solutions. The generation proxy comes from a Codex evaluation rather than a post-training data pipeline, so its transfer to such a pipeline is also an assumption.

F Module Editing Frequency, Inheritance, and Model Preference

Counting physical edits. We audit the archived Pi, Codex, and Claude Code runs listed below. For Pi, we use controller-generated `module-resolution.json`; for Codex and Claude Code, we map the physical paths in each proposal diff through the native shared-module registry. We do not use the reviser's declared primary module. Table 3 defines the Pi registry. One observation is the final candidate of one recursive generation: a module receives one count if that candidate changes any of its owned files, regardless of the number of files, edited lines, tool calls, or repair attempts. A candidate touching two modules contributes once to each. Smoke proposals, superseded retries, frozen production references, single-surface ablations, and long-horizon experiments are outside this primary cohort.

Coverage and ownership. This module audit covers 35 runs: 25 Pi, seven Codex, and three Claude Code. Each run contributes five recorded rounds, for 175 rounds in total. The rounds contain 196 candidate–module incidences because some proposals edit multiple modules. We report counts separately for each harness because their editable surfaces differ.

F.1 Accepted and Rejected Revisions

What counts as success? Here an edit is successful only in the operational sense that the controller *accepts and inherits* its candidate. We reconcile final candidate status with the archived accepted-generation list and verify successful structural validation and the training gate for every accepted

candidate. Rejections include structural or execution failures and candidates rejected by the training-side performance rule; they are not all evaluated task failures. Of the 175 proposals, 41 are accepted. They contain 47 physical module edits. Acceptance is read from each run's final `result.json` and reconciled with the source commit recorded there to identify recovered candidates. An earlier DeepSeek Science candidate was marked accepted in an intermediate record but rejected in the final run record. Recovered Opus 4.8 Environment candidates are matched from the active tail rather than pooled with superseded records.

Editing frequency is distinct from inheritance. Within Pi, Coding Session is accepted in 4 of 11 proposals and Prompt in 7 of 21, whereas Harness Runtime is accepted in 0 of 10 and Skills in 0 of 7. Acceptance assigns a decision to the whole candidate, so these rates do not identify an individual module's causal contribution to a compound edit. Nor does acceptance establish held-out improvement: the accepted GPT-5.5 Science Prompt Loader edit loses 8.33 held-out points.

F.2 Module Preference under Matched Exposure

Common domain and denominator. To compare editing choices without unequal domain coverage, we restrict the editing-frequency comparison to Terminal, where all seven models have one complete unrestricted five-generation run. For model m and module j, the entry is $100\, n_{m,j}/5$, where $n_{m,j}$ counts generations that edit j. This marginal incidence rate keeps the same denominator for every model; compound revisions may make a column sum exceed 100%. It describes realized choices under the archived search policy, not a causal effect, statistical preference estimate, or single-surface ablation.

Distinct choices under the same editing freedom. GPT-5.6-Sol touches Prompt in three of five Terminal generations (60%), while GLM-5.3 touches Agent Loop in four (80%). Opus 5 most often edits Coding Session (60%); GPT-5.5 most often edits Tool Runtime (40%). Kimi K3 edits Agent Loop and Harness Runtime in three generations each (60%). These choices do not predict which module improves held-out performance.

Reproducible export. The refreshed archive-only audit verifies every selected member against its manifest SHA-256 and preserves the five-round decisions, module ownership, accepted patches, and test-state evidence. The audit date is 24 September 2026. Formal and historical runs remain separate, and no new evaluation is launched for this analysis.

G Artifact and Mechanism Analysis Suite

This appendix reports the task selection, run-validity rules, trajectory cases, and cost comparison behind the main results. It separates the score of a final harness from evidence about how an accepted revision changed agent behavior.

G.1 Validity before aggregation

A reported result requires a complete run, a resolved configuration, an initial and final held-out score, and the same valid-task denominator for both states. Runs with unrecovered infrastructure failures or missing final scores do not enter performance averages. They remain in the run archive rather than being counted as zero scores.

G.2 Mechanism chain: from trace to inherited behavior

A final score increase does not show why a harness revision helped. The case studies therefore pair a failure trace with the proposed code change, the training-side acceptance decision, and the later agent behavior. They identify whether the edited behavior appears in the observed rollout; they do not turn an observed score gain into a general causal estimate.

Edit frequency is not causal importance: a frequently edited module need not produce a held-out gain, and a candidate may touch several modules at once. The module statistics therefore describe

1. Feasibility filtering
Retain supported tasks with reliable verifiers, compatible infrastructure, manageable resource demands, and evidence of solvability.

2. Frozen model–harness screening
Use the same initial Pi harness and fixed budget to remove unsuitable or saturated tasks and select a range of observed difficulty.

3. Freeze disjoint training and held-out test splits

Domain	Train	Test	Total
Terminal	15	15	30
Environment Learning	15	15	30
SWE	8	8	16
Science	12	12	24
Total	**50**	**50**	**100**

Figure 6: Benchmark curation for the frozen four-domain, 100-task benchmark: feasibility filters, frozen model and harness screening, difficulty-window selection, and the final 50/50 train and test split.

editing and inheritance patterns; the case studies supply the narrower process-level evidence for individual mechanisms.

G.3 What the trajectories can show

The two case studies show how one accepted revision affected a later rollout and how another accepted revision failed to address the task's final failure. They do not estimate how often either mechanism occurs across all runs.

Frozen production references. Table 10 separates three measurements that are easy to conflate. The zero-budget frozen references are the correct comparison for the open-versus-production comparison, since they hold the harness fixed. The production harnesses that received a five-round evolution budget are reported separately and are deliberately absent from Table 1: they do not expose Pi's editable boundary, their per-domain n_{test} and acceptance rules differ, and mixing them into the Pi grid would compare unlike interventions. Notably these production harnesses also regress under evolution. Codex on Terminal moves $60.00 \rightarrow 53.33$ and Claude Code on Science $41.67 \rightarrow 33.33$, so the regressions observed for Pi are not an artifact of the open substrate.

G.4 Score, resource use, and amortization

The cost comparison is reported numerically in Table 1 and in the adaptation-cost tables below. The earlier operating-efficiency PDF used a superseded result snapshot and is not included in the submission.

Table 10: Production-harness references and evolved variants.

Model	Harness	Domain	n_{test}	S_{test}	Cost
Frozen, no evolution budget (reference for the open-versus-production comparison)					
GPT-5.6-Sol	Codex	Terminal	15	51.11	$47
GPT-5.6-Sol	Codex	Environment Learning	15	33.74	$1,146
GPT-5.6-Sol	Codex	SWE	8	54.17	$97
GPT-5.6-Sol	Codex	Science	12	0.00	$24
Production harness with a five-round evolution budget ($H_0 \rightarrow H_5$)					
GPT-5.6-Sol	Codex	Terminal	15	60.00 → 53.33 (−6.67)	$492
GPT-5.6-Sol	Codex	SWE	8	54.17 → 62.50 (+8.33)	$975
GPT-5.6-Sol	Codex	Science	12	0.00 → 0.00 (0.00)	$507
GPT-5.5	Codex	Terminal	15	60.00 → 62.22 (+2.22)	$706
GPT-5.5	Codex	Environment Learning	15	28.01 → 27.68 (−0.33)	$3,436
GPT-5.5	Codex	Science	12	13.89 → 16.67 (+2.78)	$353
Claude Opus 5	Claude Code	Terminal	15	62.22 → 68.89 (+6.67)	$812
Claude Opus 4.8	Claude Code	Terminal	15	35.56 → 40.00 (+4.44)	$737
Claude Opus 4.8	Claude Code	Science	12	41.67 → 33.33 (−8.33)	$252
Protocol diagnostics					
GPT-5.5	Codex	SWE	8	62.50 *(no-update gap)*	–
GPT-5.5	Codex	Science	12	8.33 *(no-update gap)*	–
GPT-5.6-Sol	Pi	Terminal	15	51.11 → 53.33 (+2.22) *(rerun)*	$115

H Case Studies

The two cases show different outcomes of accepted harness edits. In the first, the revised agent loop prompts a final-state check after destructive cleanup, and the later rollout completes the task. In the second, the meta-agent fixes a real shell error, but the passing rollout never uses that fix and the final harness performs worse on held-out tasks. Each case shows the task, the edit, the training decision, and the observed behavior. These examples do not estimate how often either outcome occurs across runs.

H.1 What Do the Trajectories Show?

The two cases below trace a proposed harness change from task experience to source edit, training evaluation, and the agent's later behavior. In Case H.2, an accepted completion check prompts the agent to verify its final state after destructive cleanup. In Case H.3, an accepted source edit addresses a real intermediate error, but the edited branch does not explain the passing candidate's behavior.

H.2 Case 1: RSI Success, Causal Localization, and Final-State Repair (Claude Opus 5, Git Webserver)

The Opus-5 meta-agent compared all three Round-1 training rollouts of `terminal-bench-configure-git-webserver`: repeats 0 and 2 passed, while repeat 1 failed. The failed agent had already demonstrated a correct clone, commit, push, and curl pipeline, but then erased the repository reference and deployed file during cleanup and stopped on the basis of its earlier success. The proposed edit to `agent-loop.ts` therefore changes the stopping mechanism: after a destructive mutation, old success evidence is no longer accepted as proof of the delivered final state.

RSIBENCH: configure-git-webserver **Claude Opus 5 / edited agent loop**

Task: configure a bare Git repository at `user@server:/git/server` and a post-receive deployment hook for the web server on port 8080.
Reference outcome: from a fresh client, the literal clone, create, commit, and push sequence succeeds and `curl http://server:8080/hello.html` returns "hello world."

Meta-Agent Diagnosis (three same-task baseline rollouts)

Inspect matched siblings
Action: `compare repeat_00 (pass), repeat_01 (fail), repeat_02 (pass)`
Result: the external task, model, verifier, and source are held fixed; the within-task contrast isolates how the final environment was delivered.

Localize the failed rollout
Action: `action 28: clone/commit/push/curl succeeds; then rm -rf refs and webroot, update-ref -d, and gc -prune=now; action 30: stop`
Result: the only end-to-end proof precedes a destructive state change. A fresh verifier later observes `failed to update ref` and HTTP 404.

Infer the mechanism-level defect
Reasoning: The loop terminates when the model emits no more tool calls; it does not ask whether the evidence supporting completion still describes the final state that the grader will inspect.
Result: the defect is stale completion evidence, not inability to configure Git or execute the requested workflow.

Source Modification (edit `agent-loop.ts`)

Edit the deterministic stopping path
Action: `packages/agent/src/agent-loop.ts; agent.ts; types.ts`
Result: The `agent-loop.ts` edit adds a task-contract audit that extracts literal commands and named artifacts from the user task, records when they last succeeded, and tracks later destructive mutations of contract-relevant state.

Intervene only when prior evidence becomes stale
Result: before exit and at bounded mid-run checkpoints, the harness injects an explicitly labelled `[pi harness]` message when a required workflow has not been re-run after destructive cleanup. The message continues the same agent loop; it does not alter the task or the external verifier.

Original Rollout (25 tool calls, 25 turns, 151 s; reward 0)

Earlier: establish a working pipeline
Action: `git clone ...; git commit; git push origin master; curl .../hello.html`
Result: the deploy hook fires and curl returns "hello world."

Later: "reset to a pristine empty state"
Action: `rm -rf /git/server; git init -bare ...; update-ref -d refs/heads/master; rm -f /var/www/git/*`
Result: the previously verified commit and web artifact are removed after the successful check. The agent does not execute the literal task sequence again against this final state.

Finish
Reasoning: "Done. Verified end-to-end … then reset the repo to a pristine empty state for you."
Result: the external verifier's fresh push is rejected with `failed to update ref`; the web request returns HTTP 404. Reward 0.

Failure diagnosis. The agent's report is locally faithful. The workflow really did pass earlier, but the evidence is stale. Cleanup occurs after validation, so the final state no longer satisfies the task. Checking listeners or the web root alone cannot substitute for replaying the committing workflow from a fresh client.

Evolved Rollout
42 tool calls, 40 turns, 294 s; reward 1

176 seconds / 27 tool calls: automatic checkpoint
Result: the evolved harness injects a message explicitly labelled as harness-generated: "The task's quoted commands all succeeded earlier in this run, but afterwards the environment was changed destructively (`rm -rf`) and none of them has been re-run since. The grader only sees the final state."

Immediately following assistant turn
Reasoning: "I destroyed the repo when resetting it. The hook and ownership are gone. Restoring and re-validating now."
Action: `inspect /git/server; rebuild repo and hook; chown -R user:user; git config -system user.name/user.email`
Result: the observed root-owned recreated repository is replaced by a durable user-owned final state; the literal commit command no longer relies on hidden per-command identity overrides.

Literal final-state replay
Action: `git clone; echo; git add; git commit; git push origin master; curl`
Result: the trace records `remote: Deployed master to /var/www/html`, a new `master` branch, and "hello world." Reward 1.

Causal-review verdict: `loaded_and_triggered`; predicted behavior `matched_prediction`; score attribution `causal_improvement`; Round 2 `accept`.

Task-level context. The paired repeat changes from 0 to 1, taking this task from $2/3$ to $3/3$. The same accepted candidate also takes `build-cython-ext` from $1/3$ to $3/3$ and `qemu-alpine-ssh` from $2/3$ to $3/3$, while the all-pass `headless-terminal` anchor remains $3/3$. The post-review is careful not to overclaim: the QEMU movement lacks an observed checkpoint trigger and is therefore treated as consistent with, but not independent proof of, the mechanism.

Takeaway. The improvement is not a generic instruction to "try harder." It changes when evidence is considered valid: a successful check becomes stale after a destructive mutation, forcing the agent to verify the actual delivered state. The case also illustrates why RSIBENCH stores source-loading hashes, injected harness events, adjacent assistant responses, terminal outputs, paired scores, and the inheritance decision together.

H.3 CASE 2: RSI FAILURE, A REAL BUG, THE WRONG BOTTLENECK, AND ACCIDENTAL ACCEPTANCE (GPT-5.5, GIT WEBSERVER)

This failure case comes from an earlier five-round GPT-5.5 Terminal run, separate from the primary run in Table 1. It is especially informative because the meta-agent did not fail to inspect trajectories or to produce a valid source edit. It compared all three training rollouts for `terminal-bench-configure-git-webserver`, found a real intermediate Bash-runtime anomaly, changed the responsible source file, and obtained a positive aggregate training gate. The RSI failure lies one level deeper: the meta-agent stopped its causal analysis before the decisive part of the failed trajectory, edited a real defect that was not the reward bottleneck, and never checked whether the passing candidate activated the new mechanism. Mixed rollout fluctuations then produced a net gain of two passes, which the score-only gate accepted. The accepted local movement therefore was not causal evidence for the edit and did not transfer to the final reported evaluation.

RSIBENCH: configure-git-webserver **GPT-5.5 / edited Bash runtime**
Task: configure a bare Git repository at `user@server:/git/server`, deploy pushes through a post-receive hook, and serve the deployed file on port 8080.
Reference outcome: a fresh client can clone, create `hello.html`, commit, push `master`, and retrieve "hello world" from `http://server:8080/hello.html`.

Meta-Agent Analysis (three same-task training rollouts)

Inspect matched siblings
Action: `compare rewards [0, 1, 0] and open the raw commands, tool results, and verifier output`
Result: the meta-agent correctly uses a within-task contrast and localizes an unusual event in failed repeat 0 rather than proposing an unrelated prompt change.

Find an intermediate runtime anomaly
Action: `action 06: pkill -f "python3 -m http.server ..."; then nohup, curl, and a success echo`
Result: the shell stops immediately after the self-matching `pkill`; the later commands do not execute, yet the Bash tool returns a non-error observation because the child process reports a null exit code.

Infer the proposed defect
Reasoning: A null exit code is being treated as success, so the task agent may continue under a false belief that the webserver setup succeeded.
Result: this is a genuine source-level defect, but it is not yet a causal explanation of the final reward.

The Evidence GPT-5.5 Did Not Follow to the End

Actions 07 and 08: observe and recover
Action: `inspect processes and curl; start python3 -m http.server again`
Result: the same failed rollout notices the missing daemon and restores it, so the silent result at action 06 does not prevent recovery.

Action 09: complete the literal workflow
Action: `git clone; commit hello.html; git push origin master; curl /hello.html`
Result: the trace records a successful push and "hello world."

Action 10: destroy the verified final state
Action: `update-ref -d refs/heads/master; remove deployed web content`
Result: the fresh verifier subsequently receives HTTP 404. Thus the decisive failure is stale final-state evidence after destructive cleanup. The mechanism reconstructed in Case H.2 is not a failure to recover from the null exit.

Source Modification (edit `bash.ts`)

Change Bash result classification
Action: `packages/coding-agent/src/core/tools/bash.ts`
Result: after output capture, `exitCode === null` now raises "Command terminated before reporting an exit code." Numeric nonzero exits remain errors and zero remains success.

Predicted downstream behavior
Result: the task agent should see an explicit error, rerun the interrupted setup in guarded steps, and verify the daemon before finishing. This prediction addresses action 06, but it does not prevent the later deletion of the Git ref and web artifact.

Original Failed Rollout (reward 0)

Intermediate anomaly
Action: `pkill -f ...; nohup python3 -m http.server ...; curl ...`
Result: execution stops after `pkill` and is returned as a non-error.

Recovery already occurs under the original source
Action: `inspect; restart server; clone; commit; push; curl`
Result: the complete client workflow succeeds and curl returns "hello world." This directly falsifies the strong claim that the null exit caused the terminal failure by preventing recovery.

Actual terminal failure
Action: `delete refs/heads/master and clear the deployed webroot`
Result: the agent stops using its earlier success as evidence; the fresh verifier observes HTTP 404.

Accepted Candidate Rollout (reward 1, edited branch not activated)

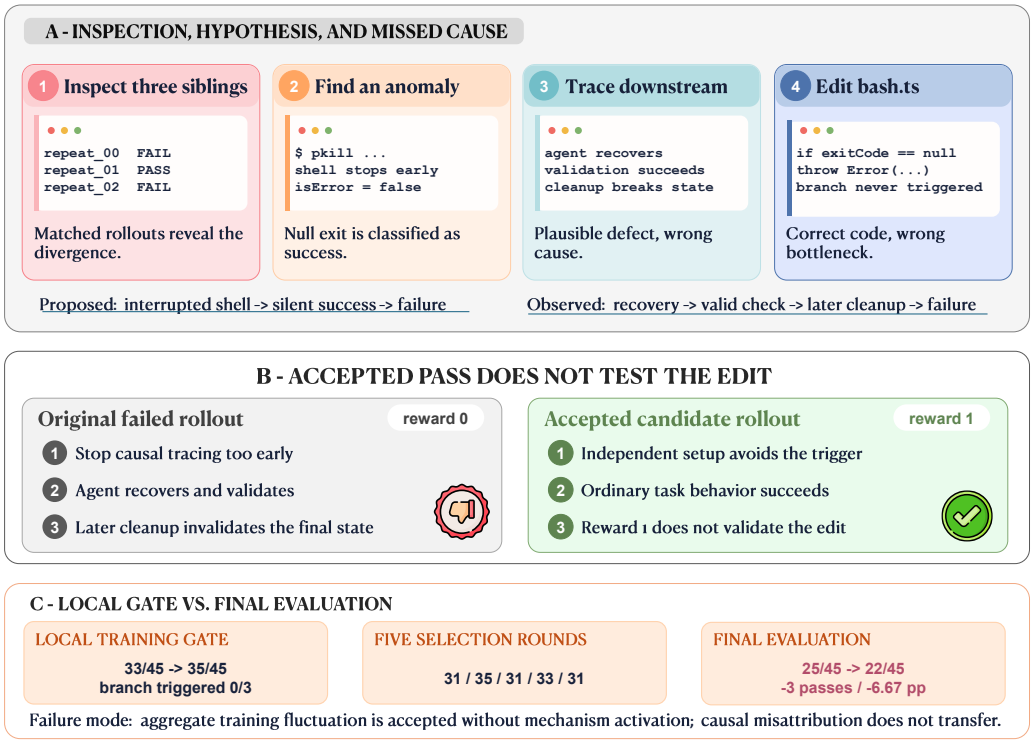

Figure 7: A GPT-5.5 Bash-runtime edit passes the training selection rule without activating its intended fix in the passing rollout for the diagnosed task. The original rollout recovered from the shell error but later removed the working final state. The candidate's training score rose from 33/45 to 35/45 passes; its final held-out score fell from 25/45 to 22/45.

Setup and validation
Action: `create repository and hook; start webserver; retry clone as user; commit; push; curl`
Result: curl returns "hello world" and the rollout passes.

Mechanism audit
Result: the trajectory contains zero occurrences of the new "Command terminated before reporting an exit code" error. All three candidate rollouts for the diagnosed task have trigger count zero; only one of all 45 candidate training rollouts triggers the branch.
Causal-review verdict: the target-task movement from [0,1,0] to [1,1,0] is a score flip without observed activation of the proposed mechanism, so it is not causal validation of the edit.

Why the Wrong Edit Was Accepted: Rollout Fluctuation

Round-2 full-training gate
Result: the current source scores $33/45 = 0.7333$ and the candidate scores $35/45 = 0.7778$, so the direct gate accepts. The movement is mixed: `sanitize-git-repo`, `make-mips-interpreter`, `tune-mjcf`, and `configure-git-webserver` gain passes, while `qemu-alpine-ssh` and `gcode-to-text` lose passes.

Why this is accidental acceptance rather than causal validation
Result: the four positive and two negative task-level flips yield a net $+2/45$, but the diagnosed mechanism activates in none of the three candidate rollouts for the target task. Rollout variation cannot be eliminated merely by expecting repeat outcomes to be identical; a score-only gate therefore cannot distinguish this mixed fluctuation from a mechanism-caused gain.

Five-round search trajectory
Result: candidate totals are 31, 35, 31, 33, and 31 passes out of 45. Only Round 2 is accepted; the acceptance rate is $1/5$.

Final reported evaluation
Result: the initial source scores $25/45 = 0.5556$ and the evolved source scores $22/45 = 0.4889$, a decline of three passes or 6.67 percentage points. The largest movements are `query-optimize` $3/3 \rightarrow 1/3$, `rstan-to-pystan` $3/3 \rightarrow 2/3$, `dna-assembly` $1/3 \rightarrow 0/3$, and `torch-pipeline-parallelism` $1/3 \rightarrow 2/3$. The run is recorded as valid rather than invalidated by infrastructure contamination.

Failure diagnosis. GPT-5.5 demonstrates useful code-reading and source-editing competence: the null-exit behavior is real, and the patch is narrowly scoped. Its RSI shortcoming is causal attribution. It explains an intermediate anomaly but does not ask whether the original agent later recovers, whether the proposed branch activates in the passing candidate, or whether the same mechanism accounts for movements on other tasks. The direct aggregate gate then accepts a small, mixed score change without post-validation mechanism evidence.

Takeaway. The training-score gain did not validate the proposed fix: the passing rollout never entered the edited branch, and the final harness scored lower on held-out tasks. The case shows why a useful source edit and a positive selection score do not, by themselves, establish a reusable harness improvement.

I Full Trajectory Records for the Main-Text Cases

This appendix summarizes two trajectory analyses from the main experiments. It reports the complete repeat-level outcomes and focuses on the behavioral mechanism and its limits rather than reproducing internal execution logs.

I.1 Claude Opus 5: completion-control edit

The accepted round-3 edit modifies `agent-session.ts`. It adds a guarded requirement-coverage review after an unattended run has produced work and is about to stop. The review reopens the request, checks explicit requirements, and reruns the smallest proving tests.

The held-out score rises from 66.67 to 79.17 on eight SWE tasks. These results do not isolate the effect of the completion check from other changes in the accepted revision.

I.2 GPT-5.5: Science regression after round-4 inheritance

The formal GPT-5.5 Science run accepts one round-4 candidate that edits `system-prompt.ts`. The candidate changes the finalization rule from shallow structural checks toward semantic verifier checks. Its training score moves from 11.11 to 16.67, but its held-out Science score moves from 11.11 to 2.78. Two test tasks regress and none improves. This does not prove that the prompt edit caused the aggregate regression; no prompt swap, keep/drop ablation, or repeated search was run.

J Extended Related Work

J.1 From Static Benchmarks to Interactive Harness Improvement

Most agent benchmarks evaluate competence under a fixed agent system. SWE-bench, MLE-bench, and Terminal-Bench place agents in realistic, executable environments and assess their ability to repair software, conduct machine-learning experiments, or solve challenging terminal tasks (Jimenez et al., 2024; Chan et al., 2025; Merrill et al., 2026). These benchmarks may require extensive interaction, iterative reasoning, and repeated modification of task artifacts, but the harness through which the agent operates is specified in advance and remains fixed throughout evaluation. Its prompts, memory, control flow, and tool runtime do not change. These benchmarks therefore measure how well an agent solves tasks with a given harness, not whether it can improve the harness that shapes its future capabilities.

EdgeBench extends this setting by evaluating whether agents can improve through accumulated experience. Agents operate for at least 12 hours per task and can preserve information through context, workspace artifacts, and environmental feedback, allowing later decisions to benefit from

Table 11: Repeat-level outcomes for two SWE training tasks.

Task	Baseline passes (3)	Candidate passes (3)
Action pinning lint	1	3
Atomic signal selectors	1	2

earlier interaction (Zhu et al., 2026). ARC-AGI-3 studies a related form of adaptation in unfamiliar, instruction-free worlds, where agents must explore, infer the environment's dynamics and objectives, and use those discoveries to guide subsequent actions (Foundation, 2026). Both benchmarks therefore capture *experience-driven adaptation*: experience changes how the agent behaves later. What they do not test is whether that experience can be consolidated into persistent changes to the agent system itself, so that what was learned becomes an inherited capability of a successor rather than knowledge available only to the current agent.

Recent harness-evolution systems address this transition by making the agent system itself an object of improvement. Self-Harness and Recursive Harness Self-Improvement (RHI) carry accepted harness- or prompt-level revisions across iterations (Zhang et al., 2026a; Lee et al., 2026a). Most closely related, Evo-Bench evaluates long-horizon evolution of a policy harness across search, office, and general tasks, while keeping fixed the meta-level evolve harness that proposes and evaluates revisions (Huang et al., 2026). A successor harness may therefore perform tasks better, but it does not improve the mechanism responsible for producing the next successor. Evo-Bench evaluates *harness evolution under a fixed improvement process*, rather than recursive self-improvement in which an improved system participates in producing further improvements. HSI moves closer to this recursive setting by allowing a meta-evolver to modify the evolver strategy, while retaining a frozen outer anchor (Zhou, 2026). RSIBENCH makes this recursive transition the object of evaluation: experience and reflection can be consolidated into verified, persistent harness changes; those changes are inherited by successor agents; and the resulting successor system is itself used to produce subsequent improvements. RSIBENCH thus evaluates the cycle from experience to revision, selection, inheritance, and further revision. It tests whether an agent can improve the system used to produce its next revision.

J.2 Adapting Harnesses to Models

Model training and harness revision change different parts of an agent. Tool-use training can teach a model to follow a tool interface (Qin et al., 2023). Harness revision instead changes the prompts, context, tools, or execution rules around a fixed model. Both can affect task performance, but this paper does not compare their costs at equal held-out quality. Appendix E.7 states the assumptions behind the cost estimates used here.

Niklaus (2026) provides a direct empirical demonstration of this premise. Holding DeepSeek-V4-Pro fixed, the study applies a Meta-Harness-style search to the surrounding legal-agent runtime and obtains substantial held-out gains without updating model weights. Its strongest variants improve deterministic mechanisms such as deliverable placement, matter fidelity, tool-call repair, and termination-time verification rather than relying on prompt editing alone. The result shows that substantial gains in expressed capability can arise from adapting the runtime around a fixed model.

More broadly, recent work has progressively moved the locus of adaptation outward from prompts and context to workflows and executable runtime structure. GEPA adapts prompt-level rules from trajectory feedback (Agrawal et al., 2026); ACE and MCE adapt persistent context and the procedures that construct it (Zhang et al., 2026b; Ye et al., 2026); ADAS and AFlow search over agentic programs and workflow structure (Hu et al., 2025; Zhang et al., 2025b); and Meta-Harness, AHE, and Self-Harness extend adaptation to executable harness components themselves (Lee et al., 2026b; Lin et al., 2026; Zhang et al., 2026a). Together, these methods suggest that improving an agent need not require modifying the model itself; increasingly rich forms of adaptation can instead be expressed in the computational structures through which the model acts.

RSIBENCH tests both the *extent* and the *model specificity* of this adaptation in a recursive setting. The underlying model remains fixed while the agent improves the harness through which it acts on subsequent tasks. Starting from both minimal and mature harnesses measures how much addi-

Table 12: Comparison of benchmarks for persistent agent improvement.

Benchmark	Evolution Target	Held-in/out	# Tasks	# Domains	Editable System Components	Improvement Type
Artifact-iteration benchmarks						
AutoLab (Xu et al., 2026b)	Artifact	▲	36	4	Experiment code	Iterative
PACE-Bench (Zhan et al., 2026)	Artifact	▲	144 pairs	6	Design source code	Iterative
AI4AI-Bench (Chi et al., 2026)	Artifact	▲	10	10	Algorithm code	Iterative
Self-evolving agent benchmarks						
Evo-Memory (Wei et al., 2026)	Self	✗	3,716	10	Memory	Iterative
EvoMemBench (Wang et al., 2026)	Self	▲	5,754	6	Memory	Iterative
MemoryArena (He et al., 2026b)	Self	✗	766	4	Memory	Iterative
PAST-Bench (Xue et al., 2026)	Self	▲	204	4	Memory + skills + workspace	Iterative
EvoAgentBench (Gao et al., 2026)	Self	✓	795	4	Procedural abilities	Iterative
J-TTL / EvoTest (He et al., 2026a)	Self	✗	6 games	1	Prompt + memory + tool routines	Iterative
EvoArena (Xu et al., 2026a)	Self	✗	1,439	3	Memory	Iterative
LifelongAgentBench (Zheng et al., 2025)	Self	✗	1,396	3	Experience memory	Iterative
RSI-oriented benchmarks						
RSI-Exam (2026)	Artifact	✓	88	6	Task-specific artifact	Iterative
RSIBench-Data (Meng et al., 2026)	Artifact	✓	519	4	Training data	Iterative
Evo-Bench (Huang et al., 2026)	Self	✓	608	3	Policy source code + prompt	Iterative
SEA-Eval (Jiang et al., 2026)	Self	✗	32	2	Memory	Iterative
SEAGym (Zheng et al., 2026)	Self	✓	250	2 suites	Agent state	Iterative
RSIBENCH *(Ours)*	**Self**	✓	**100**	**4**	**Source code + all agent resources**	**Recursive**

tional capability can be obtained under different levels of prior engineering. Cross-model harness swaps test the stronger claim implied by harness-to-model adaptation: whether an evolved harness is merely better in general, or is specifically better matched to the model that evolved it. Together, these experiments test whether a fixed model can improve through changes to its surrounding runtime and whether those changes become specific to the model.

Table 12 compares 16 selected prior benchmarks with RSIBENCH. Within that selection, RSIBENCH is the only row marked as requiring the inherited harness to participate in producing its next revision. The table is not an exhaustive account of harness self-improvement research. Agent resources include prompts, rules, skills, hooks, MCP configurations, and workflows. Counts use each paper's reported evaluation and coverage units; alternative assessment boundaries do not imply a sealed held-in/held-out split.