SoL-Pi: Recursively Scaling Auto-Research Loops for Efficient Agent Harness

Haozhe Liu[1*], **Tian Ye**[1*], **Sensen Gao**[2†], **Qihang Cao**[2†], **Yitong Li**[1†], **Mingchen Zhuge**
Duomin Wang[1], **Ruihua Zhang**[1], **Ping Luo**[1], **Jiawang Bian**[2], **Lei Zhu**[1], **Ligeng Zhu**[1], **Enze Xie**[1], **Song Han**[13]

[1]NVIDIA [2]NTU [3]MIT

[*] Equal contribution. [†] Core contributors.

Code Blog Post

As coding agents move from supervised code completion to unattended, around-the-clock exploration, their work expands from isolated predictions into long trajectories of reasoning, tool use, and feedback. Token efficiency therefore becomes important for scaling recursive self-improvement. We take an RSI-inspired approach at the harness layer, scaling auto-research loops across increasingly numerous and diverse environments for harness rollouts. At this scale, the process yields reusable improvements that transfer beyond their development setting, moving automated harness discovery toward production-level outcomes. Four mechanisms survive selection and form SoL-Pi, spanning action execution, context compaction, observation handling, and delegated reading. On the 51-task EdgeBench evaluation, SoL-Pi achieves performance comparable to Pi across GPT-5.6 Sol and Opus 5 while reducing recorded token traffic by 44.7–49.0% and API cost by about one third. In other words, estimated hourly savings are $8.75–$13.50 relative to native Codex and Claude Code harnesses, and $4.36–$5.71 relative to Pi.

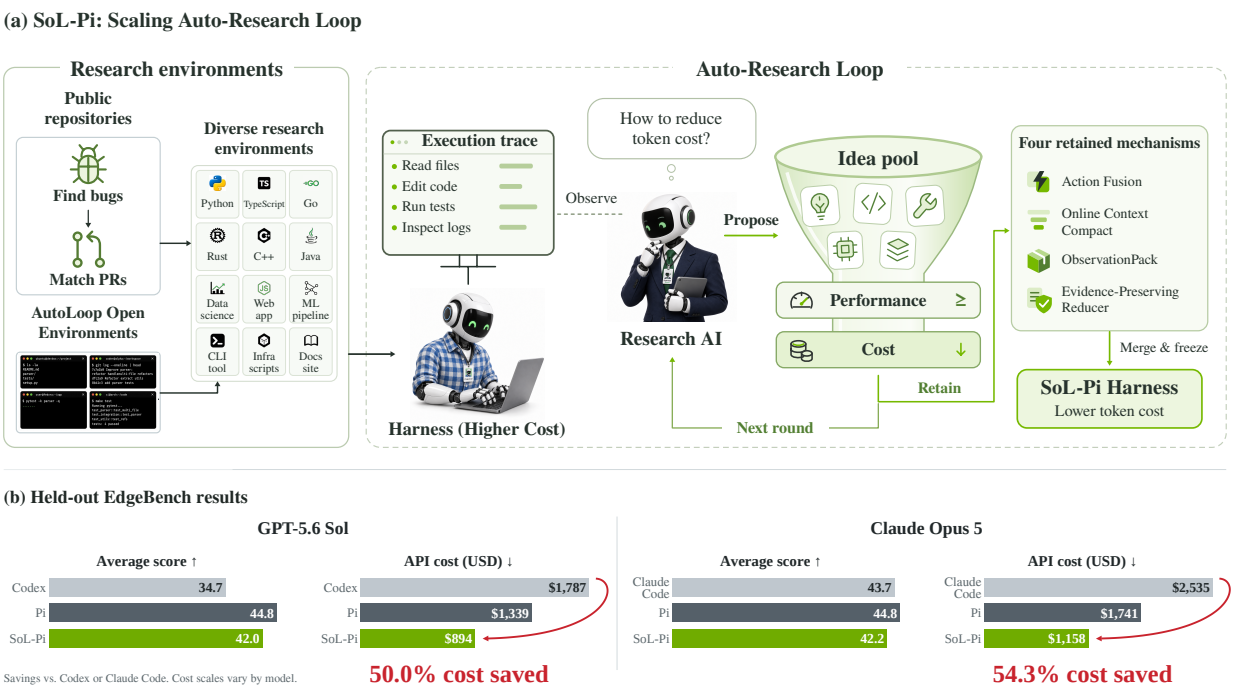


Figure 1 | **SoL-Pi discovers a more token-efficient harness through automated research.** (a) **SoL-Pi: Scaling Auto-Research Loop.** Prepared research environments supply tasks to an AI running the base harness. A research AI inspects its execution traces, proposes candidate changes, and filters the idea pool through capability and efficiency gates. Four retained mechanisms are integrated and refined into SoL-Pi before the harness is frozen for evaluation on unseen benchmarks. The trace, ideas, and gate symbols are schematic: capability is checked within fixed tolerances, and held-out results never feed back into search. (b) Example results on EdgeBench: average score and API cost for the native harnesses (Codex with GPT-5.6 Sol and Claude Code with Opus 5), Pi, and the complete four-mechanism SoL-Pi harness. SoL-Pi reduces API cost by **50.0%** relative to Codex on GPT-5.6 Sol and by **54.3%** relative to Claude Code on Opus 5.

1. Introduction

Advances in foundation models enable agents to tackle increasingly open-ended tasks over longer horizons with less supervision [1, 2, 3, 4]. This shift supports applications such as autonomous research, software engineering agents, self-evolving personal assistants, and early forms of recursive self-improvement (RSI) [5, 6, 7, 8]. As agents operate over longer horizons, task-level token efficiency becomes a first-order systems concern [9, 10]. Existing efficiency work has primarily focused on lowering the cost per token through faster attention kernels and serving infrastructure [11, 12], model compression techniques such as quantization [13, 14], or the use of cheaper models [15, 16]. In this paper, we explore an orthogonal direction: improving token use through the agent harness that mediates interactions between the model and its environment.

Harness-level optimization can improve efficiency without additional model training, complementing infrastructure- and model-level approaches [17, 18]. However, optimizing a harness is difficult in practice. Since tool use, context management, verification, delegation, recovery, and termination are tightly coupled, a change that is locally beneficial may cause downstream failures or shift token costs to later stages of execution. In practice, harness development often requires substantial human effort to inspect long execution traces, identify recurring failure modes, and translate these observations into code changes. This process is costly and difficult to scale across tasks and environments.

To accelerate this process, we adopt an RSI-inspired approach in which an AI optimizer iteratively improves the agent harness for token efficiency. In the *SoL-Pi: Scaling Auto-Research Loop* workflow (Figure 1(a)), the research AI observes execution traces from a separate agent running the base harness, proposes candidate changes, and tests them in prepared research environments. Capability and efficiency checks determine which candidates are retained, while development results guide subsequent iterations.

Recent work has demonstrated the feasibility of automated harness improvement. Meta-Harness searches over executable harness programs and evaluates their transfer to held-out datasets and models [17], while Recursive Harness Self-Improvement (RHI) iteratively refines prompt-level specifications of the agent loop for individual tasks [19]. However, a recent study using held-out tasks finds that evolved harnesses can overfit the tasks used during search and provide only marginal gains on unseen tasks [20]. These findings motivate a clear separation between search feedback and final evaluation [21]. To discover transferable efficiency improvements, we introduce SoL-Pi, a system for discovering harness improvements that transfer beyond the tasks used during search. SoL-Pi organizes autonomous research as a broad-to-deep funnel that separates candidate development from held-out validation and scales through isolated search lineages. Its design is guided by three principles:

- **Breadth and depth.** Breadth expands hypothesis coverage, while depth repeatedly implements, reviews, and hardens promising candidates.
- **Independent validation.** Held-out evidence is evaluated only after a candidate is frozen and never returns to search, preventing validation failures from being patched into task-specific solutions.
- **Scalable orchestration.** Isolated, disposable lineages let the funnel expand across more ideas and environments without coupling failures across candidates.

Guided by these principles, we scale AI-led auto-research across roughly $\sim$150 proposed directions and $\sim$500 executable environments, comprising more than 3,000 runs and more than 60,000 agent–environment interactions. The search yields four mechanisms that together form SoL-Pi. On EdgeBench [22], SoL-Pi achieves performance comparable to Pi across GPT-5.6 Sol and Opus 5 while reducing recorded token traffic by 44.7–49.0% and API cost by about one third. Beyond these results, SoL-Pi suggests that the lasting value of RSI may lie in a search process that scales across public environments to discover reusable improvements.

2. Method

2.1. Harness Auto-Research for Token Efficiency

Our search pipeline frames harness improvement as an RSI-inspired search for reusable efficiency mechanisms, starting from a broad pool of agent-generated hypotheses. The research agent analyzes execution trajectories from the base harness to identify recurring sources of overhead, builds an idea pool of candidate harness changes, and tests their effects in development environments. To qualify for acceptance, a mechanism must improve efficiency beyond a single development environment while preserving the agent's ability to complete the required work.

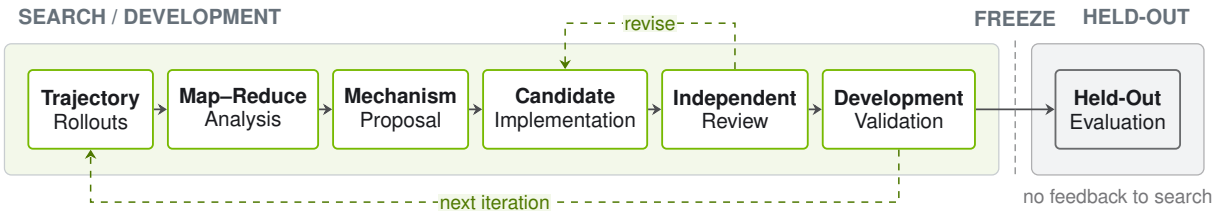


Figure 2 | **Development feedback and held-out evaluation remain separate throughout the search.** Execution trajectories guide mechanism proposals and implementation, while independent review and development validation inform revisions. Held-out evaluation occurs only after the candidate is frozen, and its results never feed back into the search loop.

Before experimentation begins, capability metrics, acceptable tolerances, and efficiency metrics are fixed and remain unchanged throughout the search. These metrics and tolerances are strictly isolated from the optimizing agent's control to prevent it from gaming the acceptance criteria. Candidate selection applies two sequential gates: every capability metric must remain within its predeclared tolerance, and the candidate must improve at least one declared efficiency metric. Among candidates that pass both gates, the pipeline retains the nondominated results under the declared metrics. Since each mechanism is explored independently, integration remains part of the *SoL-Pi: Scaling Auto-Research Loop* workflow in Figure 1(a): we combine retained mechanisms into SoL-Pi and refine its hyperparameters and implementation while preserving capability.

We reserve EdgeBench for final validation, keeping it isolated from the search process. The harness and acceptance rule are frozen before evaluation. Held-out results never feed back into the Auto-Research Loops: a failed validation rejects the candidate without triggering further optimization.

2.2. Broad-to-Deep Harness Search

Our search pipeline allocates research effort in two stages: an outer stage explores a broad pool of mechanism hypotheses, and an inner stage develops selected hypotheses independently. The outer search begins with 152 proposed directions across six proposal families: context, progress, tools, delegation, prompt and policy, and improvement and evaluation. Before rollout budgets are assigned, Oracle Analysis examines existing development trajectories to identify avoidable work in the base harness. Each selected direction must identify a concrete source of overhead and propose a harness change to address it. Independent development allows unpromising directions to terminate without affecting other experiments. The proposal families classify hypotheses by their origin rather than constrain where changes are implemented. For example, ObservationPack originates from two context hypotheses but ultimately modifies the observation boundary. This distinction helps organize the search while leaving the architecture of each mechanism open.

Each independent search follows the conventional autoresearch cycle: propose a change, implement it, run a fixed experiment, inspect the result, and retain, revise, or discard the candidate [23]. We extend this cycle with an iterative implementation loop based on the Ralph Loop [24]. The implementer refines the candidate to meet an explicit completion criterion, then an independent reviewer checks it before evaluation. Failed reviews trigger revision. Each iteration may yield multiple exploration trajectories. Independent analyzers examine one trajectory each for repeated actions, context growth, large observations, or sparse diagnostic signals. A reducer combines their findings into a candidate-level summary that guides the next proposal. Figure 2 summarizes the workflow and its isolation boundary.

We run each independent search as a disposable instance of a shared skill template containing a minimal research loop and operating instructions. Each experiment copies the template, sets its parameters, and runs to completion, retaining the candidate and evidence while discarding modified orchestration code. Search breadth comes from running more isolated loops; depth comes from repeated refinement within each loop. These counts describe the scope of our search; they do not establish a scaling law.

2.3. Search Environments

Our search pipeline uses separate environments for mechanism discovery and transfer evaluation. The search set comprises 535 executable environments: 495 repository tasks derived from GitHub issue–pull request pairs and 40 synthetic tasks with executable success verifiers. Together, they support automatically evaluated search grounded in real

software changes and open-ended problem solving.

Repository-derived environments. Each environment pairs a GitHub issue with its pre-fix repository state and offline dependencies, using the accepted patch and change history as a reference trajectory. The pull request and regression test are hidden from the agent. We retain only environments whose test fails before the patch and passes afterward.

Verifier-driven environments. We first generate an executable verifier that defines success, then construct a task environment around it. This allows multiple valid solution paths without requiring a reference trajectory. The 40 synthetic environments primarily use a Terminal-Bench-2-style verifier interface [25], extending the search beyond repository histories while preserving automatic evaluation. Figure 3 summarizes both construction paths and the isolation boundary between mechanism search and held-out evaluation.

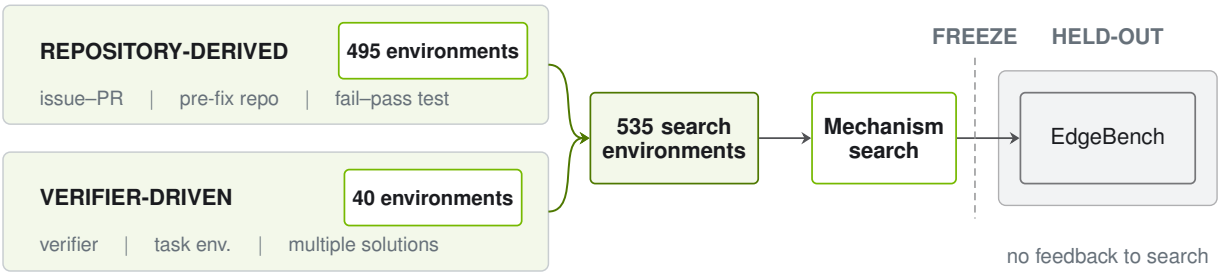


Figure 3 | **Two environment families support mechanism discovery while keeping EdgeBench held out.** Repository-derived environments use pre-fix repositories and hidden regression tests verified to fail before the accepted patch and pass afterward. Verifier-driven environments allow multiple solution paths under executable success criteria. EdgeBench remains isolated from search and is used only after the harness is frozen.

2.4. Discovered Harness Mechanisms

The search yields four reusable mechanisms that pass capability-constrained selection: Action Fusion, Online Context Compact, ObservationPack, and Evidence-Preserving Reducer. They target action execution, context management, observation storage, and delegated reading, respectively, reducing repeated work while preserving information needed for later decisions. Figure 4 shows their execution paths and fallback boundaries.

Action Fusion. Base Pi often edits a file and then issues a separate command to test, build, or run it. Action Fusion combines both actions into one tool request and returns their outcomes in a single observation, eliminating an intermediate model round trip. Commands that require inspecting the mutation result remain separate.

Online Context Compact. Online Context Compact uses plan-step completion to reconsider when to compact the context. The agent maintains its plan through `update_plan`. At each completion boundary, the harness estimates remaining model requests from the observed requests between completed steps and the number of unfinished steps. It caps this estimate by the requests that would fill the current context window at the observed growth rate. The cost gate compares projected input savings with the estimated extra cost of rewriting the prompt cache. Later compactions also account for unrecovered rewrite costs and require a larger savings margin. At these boundaries, the harness invokes Pi's native compaction when this gate passes or when context usage approaches the window limit, provided compaction can shorten the context.

ObservationPack. Large tool outputs can recur in later requests even when little of their content remains relevant. ObservationPack locally archives results exceeding the threshold (10 KiB) and sends them in full for the next two provider requests. From the third request onward, it substitutes a stable handle, the original size, and a short excerpt of complete head and tail lines. The agent can retrieve exact pages through the handle as needed. Smaller results remain unchanged.

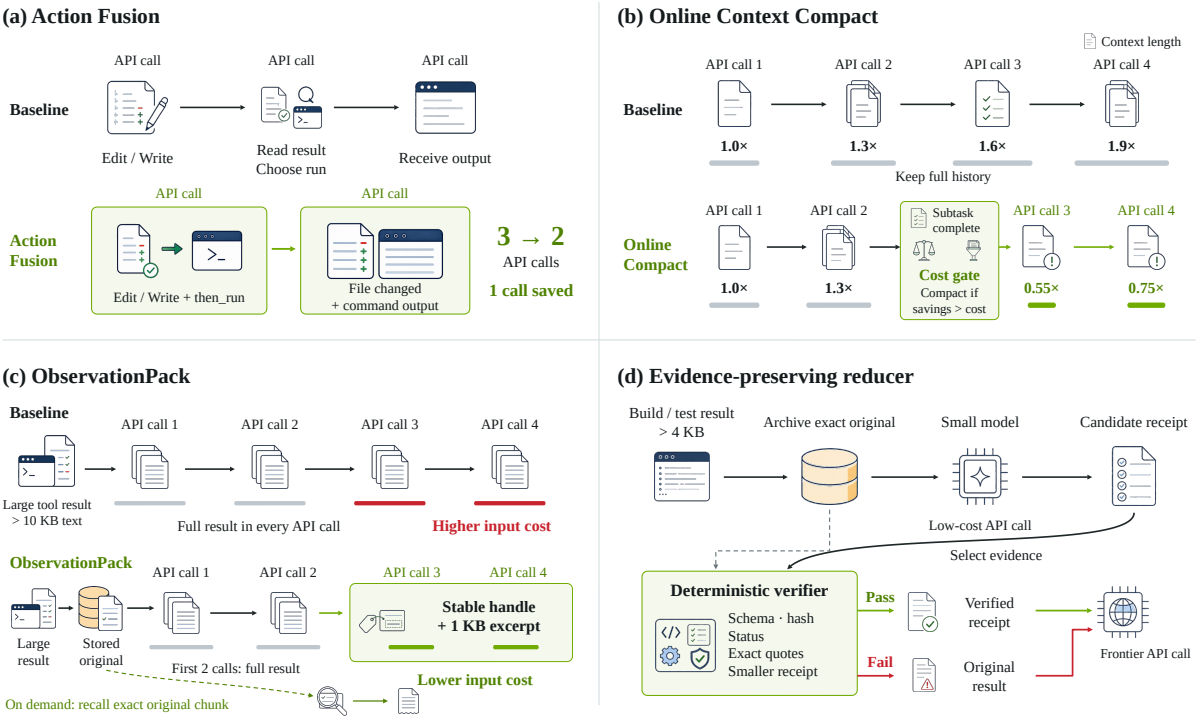


Figure 4 | **The four retained mechanisms act at different points in the agent–environment loop.** (a) Action Fusion combines a mutation and its follow-up command into one request, reducing API calls from three to two. (b) Online Context Compact evaluates compaction at subtask completion and applies it only when projected savings exceed the rewrite cost. (c) ObservationPack sends large results in full for the first two provider requests, then replaces them with a stable handle and a 1 KB excerpt. Exact original chunks remain available on demand. (d) Evidence-Preserving Reducer uses a low-cost model to compact results, with deterministic verification and fallback to the original on failure. Grey marks baseline traffic, green marks mechanism paths and savings, and red marks excess cost or verification failure.

Evidence-Preserving Reducer. Evidence-Preserving Reducer compresses build and test logs of at least 4 KiB from a predefined set of commands. File reads and search results bypass the reducer. The harness archives the exact output and asks a lower-cost model to extract key evidence into a compact receipt. A deterministic verifier checks the receipt's schema, source hash, exit status, exact quotes, and size. The harness falls back to the original log if verification fails, credentials are suspected, or the receipt provides no size reduction. The reducer processes tool results before ObservationPack projects the model context; ObservationPack recognizes the reducer's receipt marker and skips those results to preserve the verified evidence. The auxiliary model extracts evidence, while the main agent retains responsibility for diagnosis and action selection.

The four mechanisms support different stages of the agent's workflow. When the agent edits code, Action Fusion combines the edit with a follow-up command. When the environment returns output, Evidence-Preserving Reducer extracts verified evidence from build and test logs, while ObservationPack avoids repeatedly sending large results in full. When the agent completes a plan step, Online Context Compact checks whether compacting the accumulated context would save tokens. These mechanisms therefore address complementary sources of overhead. We evaluate the combined harness end to end to verify their joint effect on capability and efficiency.

2.5. Implementation Details and Backend Setup

We implement all four mechanisms as extensions to Pi. Action Fusion adds an optional follow-up command to file-mutation tools and returns both outcomes in one observation. Online Context Compact checks the cache-cost gate at plan-step completion and also supports compaction near the context limit. The gate estimates cache-rewrite overhead from the context size and the cache-write/read price ratio; it does not separately price the summarization call. ObservationPack archives large outputs locally and provides stable handles for exact retrieval. Evidence-Preserving Reducer uses GPT-5.6 Luna at `high` to extract evidence and verifies the resulting receipt before passing it to the main

agent.

Before held-out evaluation, we freeze the mechanism source, configuration, metrics, capability tolerances, and acceptance rule. All EdgeBench outputs remain outside the search loop. Of EdgeBench's 51 public tasks, 11 are used for one-way acceptance of frozen candidates; the remaining 40 are reserved for final evaluation of generalization.

3. Experimental Evaluation

We evaluate SoL-Pi on EdgeBench [22], Terminal-Bench 4 [26], IMO 2026 [27], and a kernel-optimization benchmark [28]. We report average score, token traffic (in billions), and API cost. Token efficiency is measured as API cost per unit of aggregate task score. Sec. 3.1 compares SoL-Pi with native and third-party harnesses and tests whether it transfers from GPT-5.6 Sol to Opus 5 without modification. Sec. 3.2 reports Terminal-Bench 4 completion rates and IMO 2026 Lean 4-verified problem counts, together with costs for both. Sec. 3.3 evaluates SoL-Pi in a coordinated agent swarm. Sec. 3.4 analyzes individual and combined mechanism effects and activation patterns across backends and configurations. Sec. 3.5 traces the development and selection of Action Fusion.

3.1. Overall Comparison

EdgeBench currently releases 51 of its 134 tasks as open source, and our evaluation uses this public set [22]. Table 1 compares eight evaluated configurations under GPT-5.6 Sol and lists each primary backend; the EdgeBench official GPT-5.5 row is an unranked score-only reference. We report two SoL-Pi operating points. SoL-Pi [Efficiency] is the fixed complete four-mechanism stack, reported as the token-efficiency-oriented point. SoL-Pi [Performance] is the single-mechanism configuration with the highest average score, selected separately for each backend from Table 4; under GPT-5.6 Sol, it corresponds to ObservationPack.

The Efficiency point uses a total of 1.10 B tokens, 49.0% fewer than Pi, while retaining 93.7% of Pi's average score (42.0 vs. 44.8). Its token cost is 33.2% lower than Pi's. The Performance point raises average score from 44.8 to 47.2, a 5.3% gain, while reducing token traffic by 6.1% and improving token efficiency by 9.8%.

Table 1 | **Harness comparison on EdgeBench.** SoL-Pi [Efficiency] is the fixed complete-stack point selected for token efficiency, while SoL-Pi [Performance] is the GPT-5.6 Sol single-mechanism configuration with the highest average score in Table 4. Best and next-distinct values among the eight measured rows are bold and underlined. The EdgeBench official GPT-5.5 @2h checkpoint (31.2) is unranked [29]; dashes denote unreported token traffic and token cost.

Harness	Backend	Recorded Token Traffic (B)					Token Cost ($)↓	Avg. Score↑	Token Eff. ($/score)↓
		Input↓	Cache R.↓	Cache W.↓	Output↓	Total↓			
EdgeBench official @2h	GPT-5.5	–	–	–	–	–	–	31.2	–
Codex [30]	GPT-5.6 Sol	0.0053	3.0287	0.0145	0.0052	3.0537	1,787	34.738	1.0086
OpenSquilla [31]	GPT-5.6 Sol	**0.0001**	1.2533	0.0776	**0.0044**	1.3353	1,243	24.506	0.9945
Oh-My-Pi [32]	GPT-5.6 Sol	0.1135	2.0448	0.0484	0.0168	2.2235	1,832	26.921	1.3347
OpenCode [33]	GPT-5.6 Sol	0.0012	2.2625	0.2865	0.0164	2.5668	3,422	29.552	2.2704
Oh-My-Opencode [34]	GPT-5.6 Sol	0.0044	2.4373	0.1286	0.0121	2.5825	2,678	38.523	1.3633
Pi [35]	GPT-5.6 Sol	0.0011	2.1326	**0.0141**	0.0059	2.1538	1,339	44.833	0.5855
SoL-Pi [Efficiency]	GPT-5.6 Sol	0.0009	**1.0605**	0.0316	0.0061	**1.0990**	**894**	42.003	**0.4174**
SoL-Pi [Performance]	GPT-5.6 Sol	0.0002	2.0005	0.0160	0.0056	2.0224	1,271	**47.208**	0.5280

Note. API prices change over time; all token costs in this paper use the API prices as of August 17, 2026.

Table 2 compares the native harness, Pi, SoL-Pi [Efficiency], and SoL-Pi [Performance] on GPT-5.6 Sol and Opus 5. To test transfer, we apply SoL-Pi, developed with GPT-5.6 Sol, to Opus 5 without further search or adaptation. On Opus 5, it retains 94.3% of Pi's average score while reducing token traffic by 44.7% and API cost by 33.5%, relative to Pi's point estimates. These results, alongside the activation patterns in Sec. 3.4, provide preliminary evidence of transfer to an unseen LLM backend.

Figure 1(b) summarizes the native-harness, Pi, and complete-stack SoL-Pi results on EdgeBench across the two backends. These bars are example results from our broader benchmark evaluation; Terminal-Bench 4 and IMO 2026 are reported in Sec. 3.2.

Table 2 | **SoL-Pi configurations and transfer across models on EdgeBench.** SoL-Pi [Efficiency] combines all four mechanisms and transfers from GPT-5.6 Sol to Opus 5 without further search or adaptation. SoL-Pi [Performance] uses the highest-scoring single mechanism for each model in Table 4: ObservationPack for GPT-5.6 Sol and Action Fusion for Opus 5. Within each model block, best values are **bold** and next-distinct values are underlined.

Harness	Recorded Token Traffic (B)					Token Cost ($)↓	Avg. Score↑	Token Eff. ($/score)↓
	Input↓	Cache R.↓	Cache W.↓	Output↓	Total↓			
GPT-5.6 Sol (Search Backend)								
Codex [30]	0.0053	3.0287	0.0145	**0.0052**	3.0537	1,787	34.738	1.0086
Pi [35]	0.0011	2.1326	**0.0141**	0.0059	2.1538	1,339	44.833	0.5855
SoL-Pi [Efficiency]	0.0009	**1.0605**	0.0316	0.0061	**1.0990**	**894**	42.003	**0.4174**
SoL-Pi [Performance]	**0.0002**	2.0005	0.0160	0.0056	2.0224	1,271	**47.208**	0.5280
Opus 5 (Held-Out Backend)								
Claude Code [36]	0.0002	1.8116	0.1701	0.0226	2.0045	2,535	43.689	1.1377
Pi [35]	**0.0000**	2.3203	**0.0348**	0.0145	2.3697	1,741	44.756	0.7625
SoL-Pi [Efficiency]	**0.0000**	**1.2626**	0.0352	**0.0123**	**1.3101**	**1,158**	42.224	**0.5376**
SoL-Pi [Performance]	**0.0000**	2.0520	0.0352	0.0144	2.1016	1,605	**50.482**	0.6235

Table 3 | **Harness comparison on Terminal-Bench 4 and IMO 2026.** Terminal-Bench 4 uses 63 CPU-only tasks; the IMO evaluation covers six problems. All IMO runs use GPT-5.6 Sol (`xhigh`). Costs are in USD, and best values within each benchmark are **bold**.

Harness	Terminal-Bench 4			IMO 2026		
	Solved Tasks (out of 63)↑	Total Model Cost ($)↓	Cost / Solved Task ($)↓	Pass (out of 6)↑	Total Model Cost ($)↓	Cost / Passed Problem ($)↓
Codex [30]	**18**	272.35	15.13	**5**	114.47	22.89
Pi [35]	**18**	286.45	15.91	3	75.95	25.32
SoL-Pi	15	**211.12**	**14.07**	3	**62.69**	**20.90**

Note. GPU-dependent Terminal-Bench 4 tasks were excluded due to infrastructure limits. The IMO evaluation uses AxiomMath/IMO2026's formal statements and follows Humanfia's verification setup [37, 38]. Each problem is capped at 150 minutes to limit unproductive looping, and costs include all model activity within this budget.

3.2. Evaluation on Terminal-Bench 4 and IMO 2026

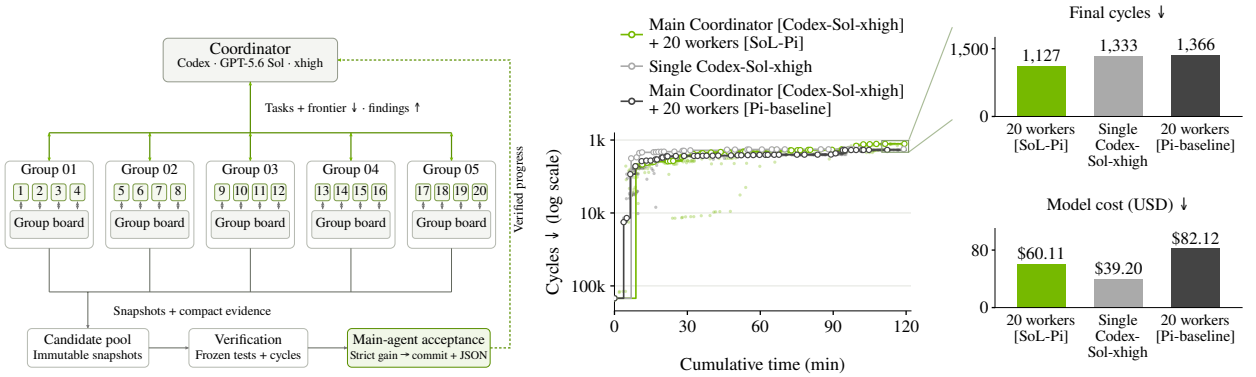
We also compare Codex, Pi, and SoL-Pi on 63 CPU-only tasks from Terminal-Bench 4 [26]. Table 3 reports the number of solved tasks, total model cost, and cost per solved task, with all costs reported as API costs. Codex and Pi each solve 18 tasks, while SoL-Pi solves 15. Compared with Pi, SoL-Pi reduces total model cost by 26.3% ($211.12 vs. $286.45) and cost per solved task by 11.6% ($14.07 vs. $15.91). These results suggest that the harness's efficiency gains generalize beyond EdgeBench, with lower total cost.

We further evaluate the three harnesses on IMO 2026 [27] using GPT-5.6 Sol (`xhigh`), requiring each solution to be formalized and verified in Lean 4. SoL-Pi passes three of six problems at a total model cost of $62.69, achieving the lowest cost per passed problem ($20.90), compared with $22.89 for Codex and $25.32 for Pi.

3.3. Efficient Agent Swarms

We evaluate SoL-Pi in a multi-agent kernel-optimization experiment measured in simulated machine cycles [28]. We compare three configurations in one two-hour run each: a single Codex agent, a Codex coordinator with 20 Pi baseline workers, and a Codex coordinator with 20 SoL-Pi workers using all four mechanisms. The single agent and coordinators use GPT-5.6 Sol at `xhigh`; all workers use GPT-5.6 Luna at `xhigh`. Every run starts from the same frozen starter, which requires 147,734 cycles, with fresh sessions and no solutions or notes from earlier runs.

In both swarms, workers form five groups of four, with independent workspaces and a shared evidence board within each group. Workers exchange notes to reproduce or combine promising findings, while the coordinator relays findings across groups. The shared best result is updated only when the coordinator submits an immutable candidate snapshot and independent verification confirms a strict improvement.



(a) Swarm architecture

(b) Verified search progress and two-hour results

Figure 5 | **Agent swarm architecture and two-hour search results.** (a) Group boards support local collaboration, and the coordinator shares findings across groups and submits candidates for independent verification. (b) Filled points show correct submitted candidates, open circles mark accepted commits, and step lines track the best accepted result. The cost chart shows cumulative API cost.

Figure 5(b) shows that the SoL-Pi swarm reaches 1,127 cycles at $60.11, compared with 1,333 cycles at $39.20 for the single agent and 1,366 cycles at $82.12 for the Pi baseline swarm. Under the same execution budget, the SoL-Pi swarm reduces API cost by 26.8% relative to the Pi baseline swarm, while the single agent remains least expensive. All final candidates pass the official correctness check. The SoL-Pi swarm and single agent pass all eight speed thresholds, whereas the Pi baseline swarm passes seven, missing the final threshold of fewer than 1,363 cycles. These results suggest that the value of SoL-Pi may extend beyond individual agents: a more efficient harness could help agent swarms and multi-agent systems turn a fixed budget into more effective collective exploration.

3.4. Learned Mechanisms and Backend Behavior

We assess the standalone contribution of each independently learned mechanism through an add-one evaluation. Table 4 compares the Pi baseline, four variants that each add a single mechanism to Pi, and the complete SoL-Pi stack under both model backends. Every component reduces the total token count under both backends. ObservationPack records the highest average score under GPT-5.6 Sol, while Action Fusion does so under Opus 5. The complete stack represents the efficiency-oriented point and has the lowest total token count and token cost in both backend blocks.

Figure 6 shows that mechanism activation varies across backends. Trigger rate is the fraction of tasks on which a mechanism activates, while trigger intensity is the mean number of activations per triggered task. Both are lower under Opus 5, which may reflect the harness being optimized exclusively on GPT-5.6 Sol trajectories. Nevertheless, every configuration evaluated under Opus 5 improves token efficiency on its triggered tasks, and the complete stack preserves a similar aggregate score–efficiency trade-off across the two backends (Table 2).

To assess the effects of combining mechanisms, Figure 7 compares each standalone configuration with the full stack. ObservationPack becomes more selective in the full stack, potentially due to overlap with Evidence-Preserving Reducer on observation-heavy trajectories. For every mechanism, the full stack achieves a larger token-efficiency gain than the standalone configuration on their respective triggered-task subsets. This pattern is consistent with complementarity in the evaluated stack, although comparisons on each configuration's own triggered-task subset do not isolate interaction effects.

Table 4 | **Single-component evaluation on EdgeBench.** Each component row adds one mechanism to Pi; SoL-Pi [Efficiency] combines all four. Light green and † identify the configuration selected as SoL-Pi [Performance] for each backend; dark green highlights SoL-Pi [Efficiency]. Token cost and token efficiency are computed using fixed API prices. Best values are **bold**, and the next-best distinct values are underlined.

Configuration	Recorded Token Traffic (B)					Token Cost ($)↓	Avg. Score↑	Token Eff. ($/score)↓
	Input↓	Cache R.↓	Cache W.↓	Output↓	Total↓			
GPT-5.6 Sol								
Pi Baseline [35]	0.0011	2.1326	**0.0141**	0.0059	2.1538	1,339	44.833	0.5855
+ Action Fusion	0.0011	1.8718	0.0180	0.0060	1.8968	1,235	46.664	0.5190
+ Online Context Compact	0.0008	1.2602	0.0217	**0.0055**	1.2881	935	41.993	0.4365
+ Evidence-Preserving Reducer	0.0051	1.9089	0.0181	**0.0055**	1.9375	1,200	44.630	0.5274
+ ObservationPack †	**0.0002**	2.0005	0.0160	0.0056	2.0224	1,271	**47.208**	0.5280
SoL-Pi [Efficiency]	0.0009	**1.0605**	0.0316	0.0061	**1.0990**	**894**	42.003	**0.4174**
Opus 5								
Pi Baseline [35]	**0.0000**	2.3203	0.0348	0.0145	2.3697	1,741	44.756	0.7625
+ Action Fusion †	**0.0000**	2.0520	0.0352	0.0144	2.1016	1,605	**50.482**	0.6235
+ Online Context Compact	**0.0000**	1.8618	0.0388	0.0145	1.9152	1,537	49.155	0.6130
+ Evidence-Preserving Reducer	**0.0000**	1.8685	**0.0315**	0.0131	1.9131	1,456	43.405	0.6578
+ ObservationPack	**0.0000**	1.3960	0.0356	**0.0102**	1.4418	1,176	47.047	**0.4899**
SoL-Pi [Efficiency]	**0.0000**	**1.2626**	0.0352	0.0123	**1.3101**	**1,158**	42.224	0.5376

Cache Reuse and Total Cost. Shortening context can reduce prompt-cache reuse when it changes a previously cached prefix [39]. Cached input also incurs a cost, so preserving a long prefix is not always the cheapest choice over an entire task. Online Context Compact and ObservationPack trade some prefix reuse for less repeated input. Table 4 shows this trade-off: with GPT-5.6 Sol, the complete stack reduces cache-read traffic from 2.1326 B to 1.0605 B tokens, while cache-write traffic increases from 0.0141 B to 0.0316 B. Despite the additional cache-write traffic, total model cost falls from $1,339 to $894. Every single-mechanism configuration also lowers cost per score point relative to Pi under both backends. These results support evaluating the full task cost alongside task quality, rather than cache reuse alone.

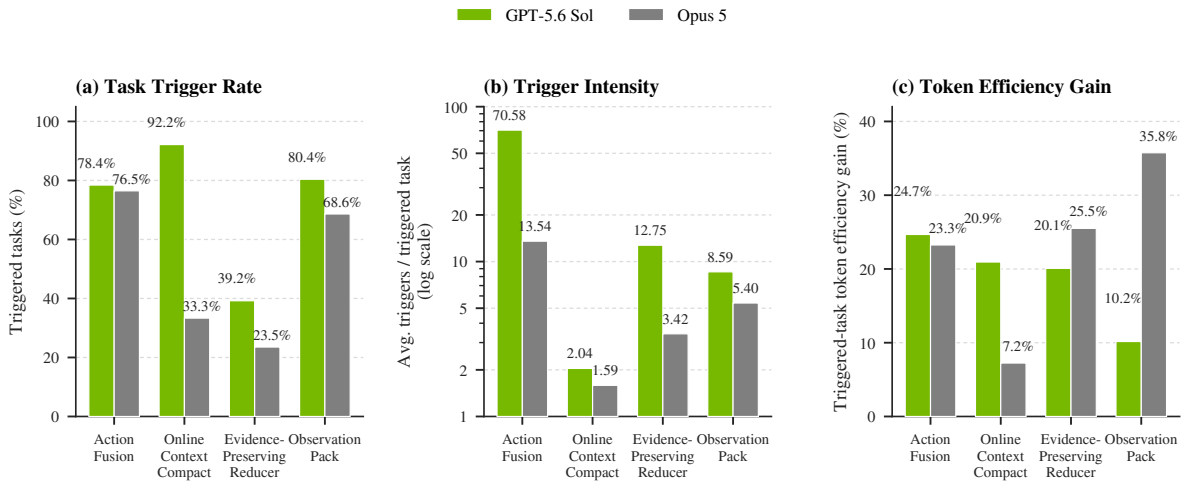


Figure 6 | **Backend-dependent activation of the retained mechanisms on EdgeBench.** Panels show trigger rate, trigger intensity on a logarithmic scale, and token-efficiency gain.

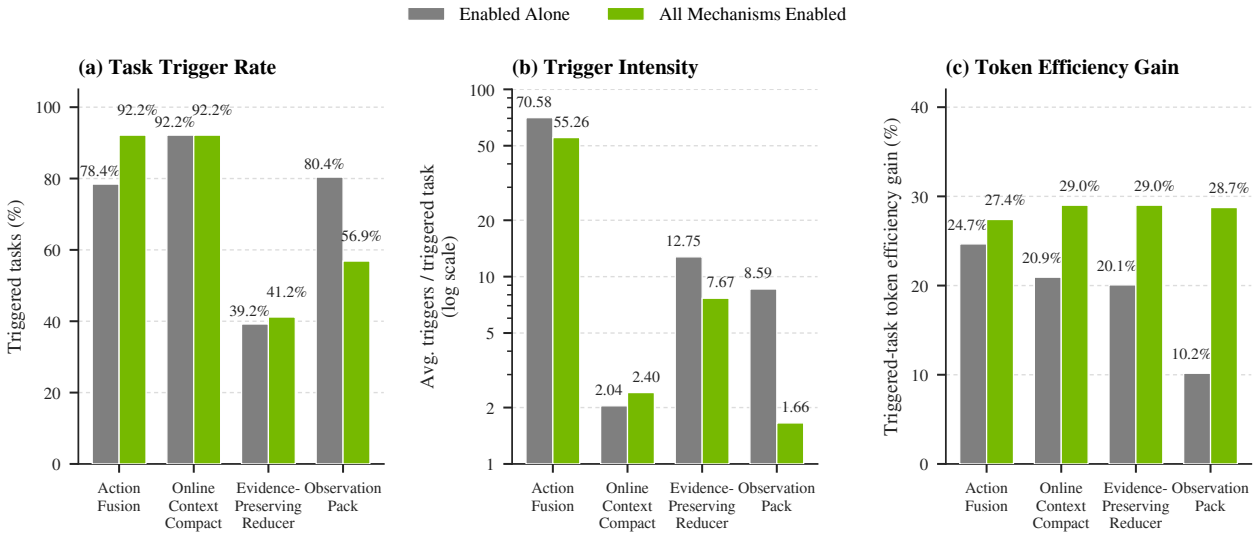


Figure 7 | **Merge behavior of independently explored mechanisms under GPT-5.6 Sol (`xhigh`).** Paired bars compare standalone and full-stack trigger rate, trigger intensity, and token-efficiency gain for each mechanism. Gains use each configuration's own triggered-task subset and corresponding disabled baseline, so comparisons are descriptive. ObservationPack becomes more selective in the full stack, while every mechanism shows a larger token-efficiency gain than in its standalone configuration, a pattern consistent with complementarity in the evaluated stack.

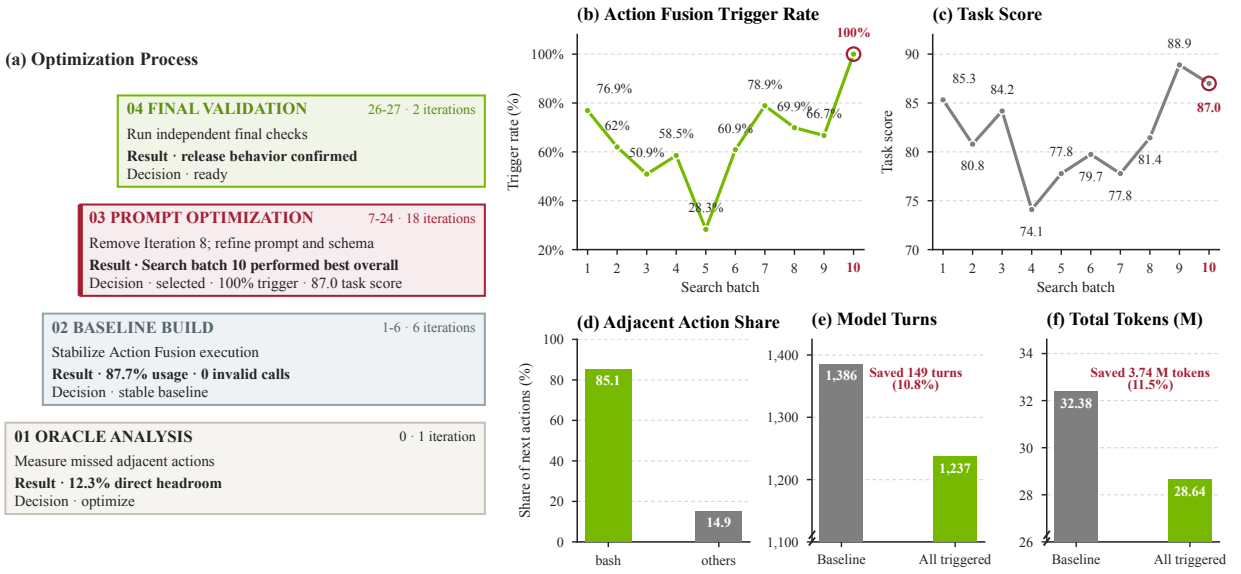


Figure 8 | **Action Fusion from opportunity to retention.** Panel (a) summarizes the 27 recorded iterations across four stages. Stage 03 records 18 prompt-optimization explorations, whereas panels (b)–(c) show ten retained steps: some steps evaluate multiple candidates in parallel, and only the best result from each parallel batch is retained. Panel (d) reports observed adjacent-action composition; panels (e)–(f) project model turns and tokens under full triggering.

3.5. From Observation to a Retained Mechanism

We use Action Fusion as a case study of how a candidate was discovered and retained. As shown in Figure 8, the recorded lineage spans 27 iterations across four stages: oracle analysis, baseline construction, prompt and tool-schema optimization, and final held-out validation.

The oracle-analysis panels (d)–(f) identify repeated adjacent actions and project an 11.5% token reduction under full triggering, motivating a dedicated Action Fusion lineage. During baseline construction, prompt-only triggering proved

unreliable. The agent therefore extended the tool schema to expose the fused action directly, establishing a stable baseline with no invalid calls. It then refined the prompt and schema on development tasks, introducing trigger rate alongside task score as an intermediate acceptance metric. The final configuration was selected using both metrics, frozen, and retained after held-out validation. This case illustrates how a lineage-local auto-research loop can stabilize a mechanism's interface and triggering behavior. The agent's introduction of trigger rate also demonstrates that auto-research can develop mechanism-specific intermediate metrics to guide optimization alongside end-task performance.

4. Related Work

4.1. Agent Harnesses and Automated Agent Design

An agent's behavior depends not only on its underlying model, but also on the harness that presents state, exposes actions, and processes feedback. SWE-agent showed that changing the agent–computer interface around a fixed model can materially affect repository-level software-engineering performance [2]. Subsequent work has examined interface design, context management, tool use, and delegation as system-level choices that shape agent behavior [40, 41]. These design choices become increasingly consequential in long-horizon tasks, where accumulated interaction histories and environment observations can increase context size and execution overhead [9, 22].

Automated agent design explores several optimization targets. GEPA improves prompts through reflection on execution trajectories [42], while ADAS formulates agent design as code search [43]. AFlow and AgentSquare search over workflow structures and modular agent components, respectively [44, 45]. Related research on recursive self-improvement has theoretical roots in the Gödel Machine [46]. Recent systems explore executable forms of self-modification: the Darwin Gödel Machine iteratively modifies and evaluates agent code [7], while Hyperagents also make the meta-level improvement procedure itself editable [8]. Our approach draws on this line of research to search for reusable efficiency improvements in harness mechanisms while keeping the underlying model fixed.

4.2. Automated Harness Optimization

Recent methods optimize harness code and configurations using execution feedback. AutoHarness synthesizes environment-specific code guards and, in some settings, complete code policies [47]. Recursive Harness Self-Improvement (RHI) refines prompt-level specifications of the agent loop for individual tasks [19]. Meta-Harness searches over executable harness programs using prior code, scores, and execution traces, maintaining a Pareto frontier over task performance and context cost [17]. AHE evolves modular coding-harness components from trajectory evidence and evaluates their transfer across tasks and models [18]. MemoHarness stores execution experience and uses it to adapt harness configurations to individual test cases at inference time [48].

The extent to which the improvements discovered generalize remains an important evaluation question. Wang et al. report limited gains on held-out tasks for the methods they evaluate and highlight the risk of overstating improvements when search and evaluation tasks overlap [20]. Our search pipeline explores candidates across diverse executable development environments, using a broad-to-deep funnel to refine promising changes while keeping final held-out evaluation separate from candidate development. The search targets token efficiency while checking that candidates maintain task performance. SoL-Pi combines four mechanisms selected through this process. Its held-out evaluation provides preliminary evidence that RSI-inspired search at the harness layer can discover mechanisms that generalize beyond their development environments.

4.3. Context and Token-Efficient Agents

Long-horizon agents can incur substantial token overhead as interaction histories and environment observations accumulate. LLM-as-Code limits context accumulation by moving deterministic control flow into executable code [49]. Other methods reduce the information retained during execution: AgentDiet removes redundant and outdated trajectory content [50], while ACON optimizes the compression of observations and interaction histories [51]. Context-Folding and AgentFold enable agents to manage their working context through trajectory folding and compression [10, 52], and Context as a Tool exposes context maintenance as an explicit agent action [53]. Agentic Context Engineering (ACE) focuses on accumulating and refining reusable strategies in context through execution feedback [54]. These approaches provide mechanisms for controlling context growth and organizing execution. Our approach studies automated discovery and selection across multiple harness components, seeking reusable changes that reduce token usage while maintaining

task performance. SoL-Pi combines the selected mechanisms across action execution, context compaction, observation handling, and delegated reading.

5. Conclusion

We introduce SoL-Pi, an RSI-inspired auto-research system for discovering and combining efficient agent-harness mechanisms. On EdgeBench, the best-performing candidates improve model performance by 5.3–12.8% and token efficiency by 9.8–18.2%, while the complete stack reduces token traffic by 44.7–49.0% and token cost by about one third at comparable performance. The complete stack's consistent results across GPT-5.6 Sol and Opus 5 demonstrate strong cross-model generalization within the evaluated setting, positioning SoL-Pi as a preliminary step toward scalable RSI systems.

5.1. Limitations and Future Directions

Pre-Training the Harness. Generalizing harness artifacts discovered through RSI remains a persistent challenge. Our results provide preliminary evidence that scaling auto-research loops can mitigate this challenge. Analogous to pretraining, the harness is exposed to many tasks and updated from the resulting trajectories. We hypothesize that scaling both executable environments and the diversity of research ideas can yield sustained gains; we call this long-term research direction *pretraining the harness*. We will continue to explore this direction.

Multi-Backend Training. Treating the harness as trainable suggests a complementary path: learning from multiple LLM backends. Our current harness was updated from trajectories generated by a single LLM; on the other evaluated backend, its mechanisms trigger less often and less intensively, although they still improve token efficiency when triggered. Training and validating the harness across multiple backends may improve robustness while preserving task quality and efficiency.

Recursive Efficient Improvement. Efficiency may also become recursive: a more efficient harness could lower the cost of the auto-research used to build its successor. We plan to use SoL-Pi as the starting harness for the next research cycle, where lower per-run costs could let a fixed budget cover more executable environments, trajectories, and research ideas. In this view, efficiency is both an outcome of harness research and a resource for expanding the search that follows, so a more efficient harness may help discover an even more efficient one. We call this possibility *recursive efficient improvement*; it is a long-term research vision rather than a compounding effect demonstrated by the present study.

Search Coverage and Cost. Running complete auto-research loops in our environment is computationally expensive, making controlled comparisons of search breadth and depth under a fixed budget particularly challenging. We view scaling laws along these dimensions as a promising research direction and leave their systematic investigation to future work.

References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[2] John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37: 50528–50652, 2024.

[3] Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, et al. Measuring ai ability to complete long tasks. *arXiv preprint arXiv:2503.14499*, 352, 2025.

[4] Wilson Lin. Towards self-driving codebases. Cursor research blog, 2026. URL https://cursor.com/blog/self-driving-codebases. Accessed 2026-08-17.

[5] OpenAI. Introducing GPT-5.3-Codex. OpenAI technical report, 2026. URL https://openai.com/index/introducing-gpt-5-3-codex/. Accessed 2026-08-17.

[6] Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist-v2: Workshop-level automated scientific discovery via agentic tree search. *arXiv preprint arXiv:2504.08066*, 2025.

[7] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pages 104223–104294, 2026.

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

[9] Anthropic. Effective harnesses for long-running agents. Anthropic Engineering, 2025. URL https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents. Accessed 2026-08-29.

[10] Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, et al. Agentfold: Long-horizon web agents with proactive context management. *arXiv preprint arXiv:2510.24699*, 2025.

[11] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations*, volume 2024, pages 35549–35562, 2024.

[12] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

[13] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International conference on machine learning*, pages 38087–38099. PMLR, 2023.

[14] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[15] Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023. URL https://arxiv.org/abs/2305.05176.

[16] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs with preference data. *arXiv preprint arXiv:2406.18665*, 2024. URL https://arxiv.org/abs/2406.18665.

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

[18] 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.

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

[20] Yike Wang, Huaisheng Zhu, Zhengyu Hu, Yige Yuan, Zhengyu Chen, Shakti Senthil, Hannaneh Hajishirzi, Yulia Tsvetkov, Pradeep Dasigi, and Teng Xiao. Rethinking the evaluation of harness evolution for agents. *arXiv preprint arXiv:2607.12227*, 2026.

[21] Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10835–10866. PMLR, 2023. URL https://proceedings.mlr.press/v202/gao23h.html.

[22] ByteDance Seed. EdgeBench: Scaling laws of environment learning, 2026. URL https://edge-bench.org/. Accessed 2026-08-17.

[23] Andrej Karpathy. autoresearch: The experiment loop. GitHub repository, 2026. URL https://github.com/karpathy/autoresearch/blob/master/program.md. Accessed 2026-08-17.

[24] Anthropic. Ralph wiggum plugin: Ralph loop command. GitHub repository, 2026. URL https://github.com/anthropics/claude-code/blob/main/plugins/ralph-wiggum/commands/ralph-loop.md. Accessed 2026-08-17.

[25] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026. URL https://arxiv.org/abs/2601.11868.

[26] Ryan Marten. Terminal-Bench 4.0. https://www.tbench.ai/news/terminal-bench-4-0, 2026. Accessed September 14, 2026.

[27] International Mathematical Olympiad. IMO 2026 Problems. https://www.imo-official.org/problems/2026/, 2026. Accessed September 14, 2026.

[28] Anthropic. Anthropic's original performance take-home. https://github.com/anthropics/original_performance_takehome. Accessed September 12, 2026.

[29] ByteDance Seed. EdgeBench official leaderboard. Official GitHub repository, 2026. URL https://github.com/ByteDance-Seed/EdgeBench. Official @2h score, commit a87350a; accessed 2026-08-25.

[30] OpenAI. Codex CLI. GitHub repository, 2026. URL https://github.com/openai/codex. accessed September 15, 2026.

[31] TokenRhythm. OpenSquilla. GitHub repository, 2026. URL https://github.com/TokenRhythm/opensquilla. accessed September 15, 2026.

[32] can1357. Oh My Pi. GitHub repository, 2026. URL https://github.com/can1357/oh-my-pi. accessed September 15, 2026.

[33] Anomaly. OpenCode. GitHub repository, 2026. URL https://github.com/anomalyco/opencode. accessed September 15, 2026.

[34] code-yeongyu. Oh My OpenCode. GitHub repository, 2026. URL https://github.com/code-yeongyu/oh-my-openagent. Repository now named oh-my-openagent; accessed September 15, 2026.

[35] Earendil Works. Pi: Coding Agent Toolkit. GitHub repository, 2026. URL https://github.com/earendil-works/pi. Formerly pi-mono; accessed September 15, 2026.

[36] Anthropic. Claude Code. GitHub repository, 2026. URL https://github.com/anthropics/claude-code. accessed September 15, 2026.

[37] Humanfia. Humanize Olympic Agents (HOA). https://humanfia.ai/projects/hoa, 2026. Accessed September 17, 2026.

[38] Axiom Math. AxiomProver at IMO 2026. https://github.com/AxiomMath/IMO2026, 2026. Accessed September 17, 2026.

[39] Anthropic. Prompt Caching. Claude API documentation, 2026. URL https://platform.claude.com/docs/en/build-with-claude/prompt-caching. Accessed September 17, 2026.

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

[41] Elias Lumer, Sahil Sen, Kevin Paul, and Vamse Kumar Subbiah. Recursive agent harnesses. *arXiv preprint arXiv:2606.13643*, 2026.

[42] 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, pages 8479–8565, 2026.

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

[44] 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, pages 34040–34077, 2025.

[45] Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. Agentsquare: Automatic llm agent search in modular design space. In *International Conference on Learning Representations*, volume 2025, pages 3841–3865, 2025.

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

[47] Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P Murphy. Autoharness: improving llm agents by automatically synthesizing a code harness. *arXiv preprint arXiv:2603.03329*, 2026.

[48] Yue Huang, Wenjie Wang, Han Bao, Yuchen Ma, Xiaonan Luo, Yi Nian, Haomin Zhuang, Zheyuan Liu, Yue Zhao, and Xiangliang Zhang. Memoharness: Agent harnesses that learn from experience. *arXiv preprint arXiv:2607.14159*, 2026.

[49] Junjia Qi, Zichuan Fu, Jingtong Gao, Wenlin Zhang, Hanyu Yan, Xian Wu, and Xiangyu Zhao. Llm-as-code agentic programming for agent harness. *arXiv preprint arXiv:2606.15874*, 2026.

[50] Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. Reducing cost of llm agents with trajectory reduction. *Proceedings of the ACM on Software Engineering*, 3(FSE):1241–1263, 2026.

[51] Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*, 2025.

[52] Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon llm agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025.

[53] Shukai Liu, Bo Jiang, Jian Yang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon swe-agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 20604–20617, 2026.

[54] 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, pages 86069–86100, 2026.