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

Anonymous authors
Paper under double-blind review

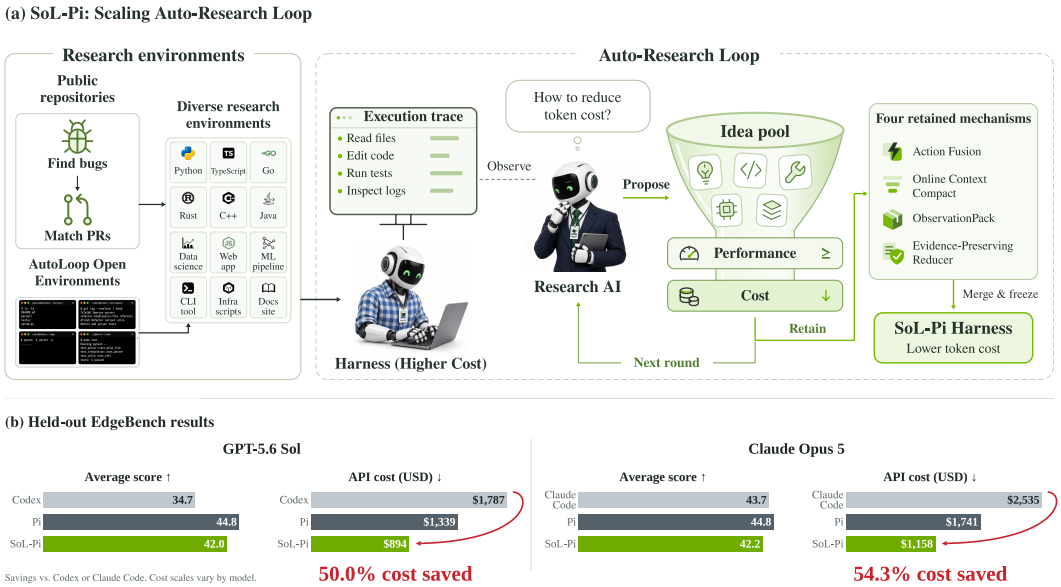


Figure 1: **SoL-Pi discovers a token-efficient harness through automated research.** (a) We assemble verifiable research environments from 495 issue–PR tasks in public GitHub repositories and 40 open-ended problems built through auto-research loops. Research agents inspect development traces, propose changes, and apply capability and efficiency gates. Four retained mechanisms are integrated before the harness is frozen for held-out evaluation. (b) On EdgeBench, the complete stack 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.

Abstract

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 (SoL stands for Speed of Light), spanning action execution, context compaction, observation handling, and delegated reading. On the 51-task EdgeBench evaluation, SoL-Pi achieves performance comparable to Pi-baseline 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 versus Pi-baseline.

1 Introduction

Advances in foundation models enable agents to tackle increasingly open-ended tasks over longer horizons with less supervision (Chen et al., 2021; Yang et al., 2024; Kwa et al., 2026; Lin, 2026). This shift supports applications such as autonomous research (Yamada et al., 2025), software en-

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.

gineering agents (OpenAI, 2026b), self-evolving personal assistants, and early forms of recursive self-improvement (RSI) (Zhang et al., 2026a;b). As agents operate over longer horizons, task-level token efficiency becomes a first-order systems concern (Anthropic, 2025; Ye et al., 2025). Existing efficiency work has primarily focused on lowering the cost per token through faster attention kernels and serving infrastructure (Dao, 2024; Kwon et al., 2023), model compression techniques such as quantization (Xiao et al., 2023; Frantar et al., 2022), or the use of cheaper models (Chen et al., 2023; Ong et al., 2024). 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 (Lee et al., 2026b; Lin et al., 2026). 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 Pi agent harness (denoted Pi-baseline), 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 (Lee et al., 2026b), while Recursive Harness Self-Improvement (RHI) iteratively refines prompt-level specifications of the agent loop for individual tasks (Lee et al., 2026a). 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 (Wang et al., 2026). These findings motivate a clear separation between search feedback and final evaluation (Gao et al., 2023). 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. Extended related work is provided in appendix A. 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 $\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 (Zhu et al., 2026), SoL-Pi achieves performance comparable to Pi-baseline 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

We frame harness improvement as an RSI-inspired search for reusable efficiency mechanisms. Research agents analyze base-harness trajectories, propose changes targeting recurring overhead, and evaluate them in development environments. Acceptance requires efficiency gains beyond a single environment while preserving task-completion capability.

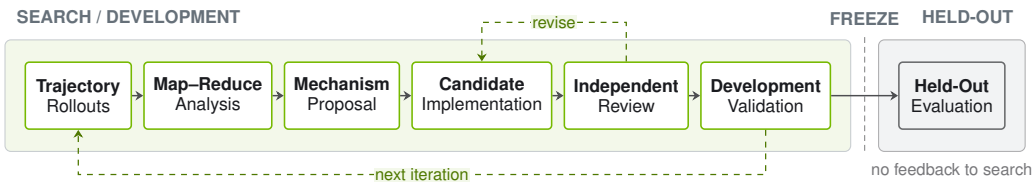


Figure 2: **Development feedback and held-out evaluation remain separate throughout the search.** Execution trajectories guide mechanism proposals and implementation, while independent review and development validation inform revisions. Held-out evaluation occurs only after the candidate is frozen.

2.1 Harness Auto-Research for Token Efficiency

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. As part of the auto-research loop in Figure 1(a), we integrate the mechanisms retained from independent exploration into SoL-Pi. We then refine the harness's 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, detailed in Sec. 2.4, 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 auto-research cycle: propose a change, implement it, run a fixed experiment, inspect the result, and retain, revise, or discard the candidate (Karpathy, 2026). We extend this cycle with an iterative implementation loop based on the Ralph Loop introduced by Huntley (Huntley, 2025) and implemented in the Claude Code plugin (Anthropic, 2026c). 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.

Each independent search uses a disposable copy of a shared skill template with a minimal research loop and operating instructions. After configuring and running the experiment, we retain its candidate and evidence but discard modified orchestration code. More isolated loops provide breadth; iterative refinement provides depth.

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-derived environments built

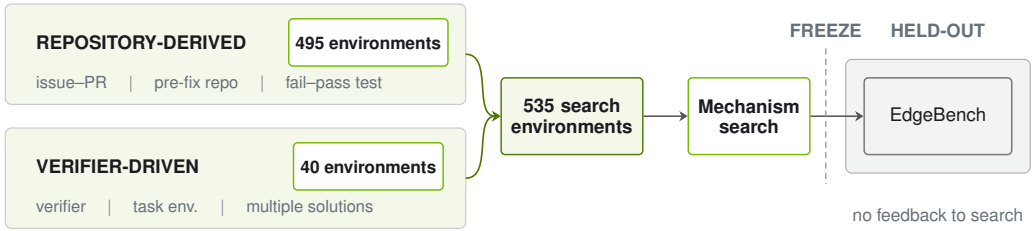


Figure 3: **Search environments and held-out evaluation.** The 495 repository-derived and 40 verifier-driven synthetic environments support mechanism discovery. Candidates are frozen before EdgeBench evaluation.

from public GitHub issue–pull request pairs and 40 verifier-driven synthetic environments from AutoLoop Open Environments. These correspond to "Public repositories" and "AutoLoop Open Environments" in Figure 1(a), respectively. Together, they support automatically evaluated search grounded in real software changes and open-ended problem solving.

Repository-derived environments. The "Find bugs" and "Match PRs" labels in Figure 1(a) depict the issue–pull request construction path. 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 synthetic 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 (Merrill et al., 2026), extending the search beyond repository histories while preserving automatic evaluation. Figure 3 summarizes both construction paths and their separation from held-out evaluation.

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. Pi-baseline 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 subtask completion to reconsider when to compact the context. Here, a subtask corresponds to a step in the plan maintained through `update_plan`. At each completion boundary, the harness estimates remaining model requests from the observed requests between completed subtasks and the number of unfinished subtasks. 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-baseline'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.

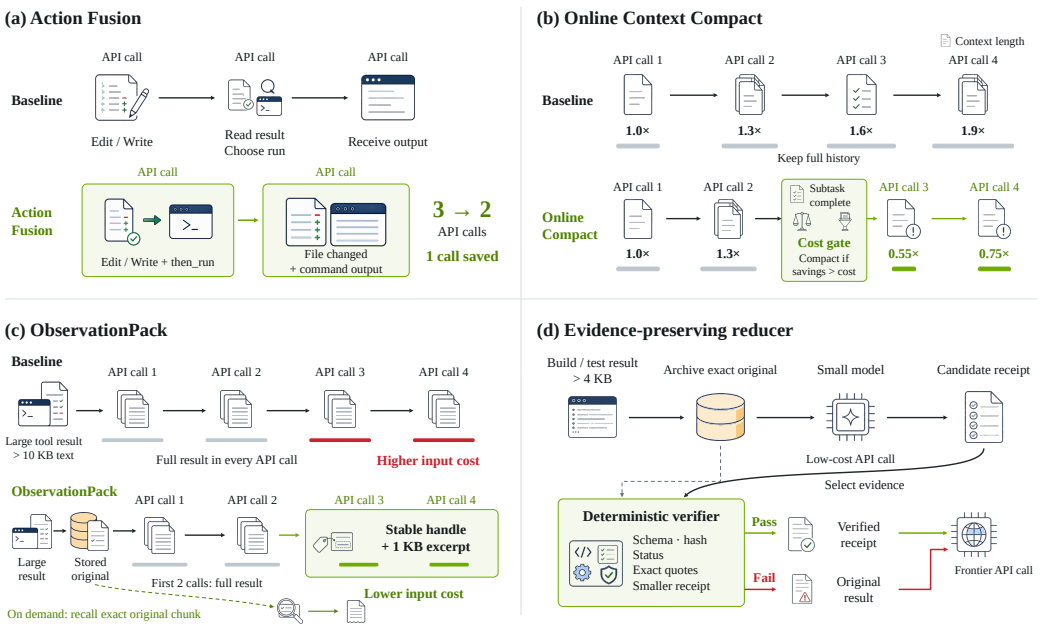


Figure 4: **Four mechanisms in the agent–environment loop.** (a) Action Fusion joins a mutation and its follow-up command in one request, reducing API calls from three to two. (b) Online Context Compact checks compaction at subtask completion, applying it only when projected savings exceed rewrite cost. (c) ObservationPack sends large results in full for two provider requests, then substitutes a stable handle and a 1 KB excerpt; exact original chunks remain retrievable. (d) Evidence-Preserving Reducer uses a low-cost model to compact results, with deterministic verification and fallback to the original on failure. Grey: baseline traffic; green: mechanism paths and savings; red: 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.

2.5 Implementation Details and Backend Setup

All four mechanisms are Pi-baseline extensions. Evidence-Preserving Reducer uses GPT-5.6 Luna with reasoning effort set to `high`, with verified receipts passed to the main agent. Before held-out evaluation, we freeze the mechanism source, configuration, metrics, capability tolerances, and acceptance rule. 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 (Zhu et al., 2026) for harness comparison and backend transfer. Although EdgeBench is excluded from harness search, 11 of its tasks serve as a validation gate for frozen candidates. We assess generalization on Terminal-Bench 4 (Marten, 2026) and IMO 2026 (International Mathematical Olympiad, 2026), which are excluded from both harness search and this validation gate, and evaluate agent swarms on a kernel-optimization benchmark (Anthropic, 2026d). We report average score, token traffic (in billions), API cost, and API cost per unit of aggregate task score ($C/\sum_i s_i$). API cost calculations are detailed in appendix C. 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 tests generalization to Terminal-Bench 4 and IMO 2026, reporting task success and API costs. Sec. 3.3 evaluates SoL-Pi in a coordinated agent swarm. Sec. 3.4

Table 1: **EdgeBench harness comparison.** Official GPT-5.5 @2h is a reference (ByteDance Seed, 2026). Bold/underline: best/next-distinct measured values. Token breakdowns: Table 6 (appendix C).

Harness	Tokens (B)↓	Cost ($)↓	Avg. score↑	$/score ↓
Official @2h	–	–	31.200	–
GPT-5.6 Sol (measured)				
Codex	3.0537	1,787	34.738	1.0086
OpenSquilla	1.3353	1,243	24.506	0.9945
Oh-My-Pi	2.2235	1,832	26.921	1.3347
OpenCode	2.5668	3,422	29.552	2.2704
Oh-My-Opencode	2.5825	2,678	38.523	1.3633
Pi-baseline	2.1538	1,339	44.833	0.5855
SoL-Pi [Efficiency]	**1.0990**	**894**	42.003	**0.4174**
SoL-Pi [Performance]	2.0224	1,271	**47.208**	0.5280

Table 2: **EdgeBench backend transfer.** Bold/underline: best/next-distinct values within each backend. Detailed token breakdowns are in Table 6 (appendix C).

Harness	Tokens (B)↓	Cost ($)↓	Avg. score↑	$/score ↓
GPT-5.6 Sol (search backend)				
Codex	3.0537	1,787	34.738	1.0086
Pi-baseline	2.1538	1,339	44.833	0.5855
SoL-Pi [Efficiency]	**1.0990**	**894**	42.003	**0.4174**
SoL-Pi [Performance]	2.0224	1,271	**47.208**	0.5280
Opus 5 (held-out backend)				
Claude Code	2.0045	2,535	43.689	1.1377
Pi-baseline	2.3697	1,741	44.756	0.7625
SoL-Pi [Efficiency]	**1.3101**	**1,158**	42.224	**0.5376**
SoL-Pi [Performance]	2.1016	1,605	**50.482**	0.6235

analyzes individual and combined mechanism effects and activation patterns across backends and configurations. Sec. 3.5 details the RSI process, using Action Fusion as an example.

3.1 Overall Comparison

EdgeBench currently releases 51 of its 134 tasks as open source, and our evaluation uses this public set (Zhu et al., 2026). Table 1 compares eight evaluated configurations under GPT-5.6 Sol; the EdgeBench official GPT-5.5 row is a 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-baseline, while retaining 93.7% of Pi-baseline's average score (42.0 vs. 44.8). Its token cost is 33.2% lower than Pi-baseline'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%.

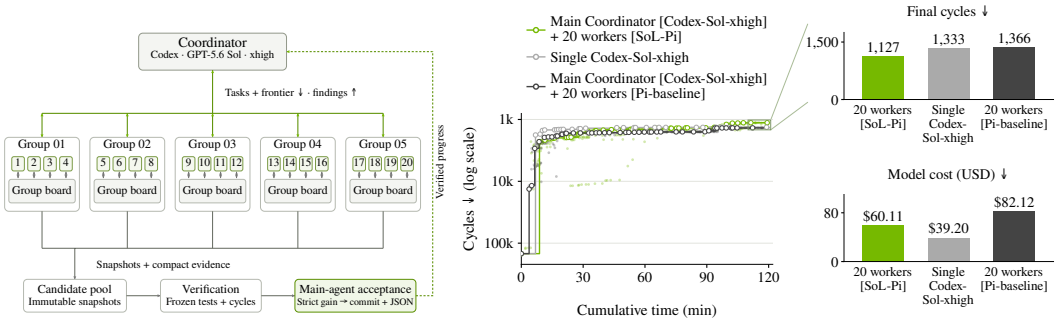
Table 2 compares the native harness, Pi-baseline, SoL-Pi [Efficiency], and SoL-Pi [Performance] on GPT-5.6 Sol and Opus 5. To test transfer, we apply SoL-Pi, developed with GPT-5.6 Sol, to Opus 5 without further search or adaptation. Only the primary backend changes: the reducer still uses GPT-5.6 Luna with reasoning effort set to `high`; full configurations are in appendix B.2. On Opus 5, SoL-Pi retains 94.3% of Pi-baseline's average score while reducing token traffic by 44.7% and API cost by 33.5%, compared with Pi-baseline. These results, alongside the activation patterns in Sec. 3.4, provide preliminary evidence of transfer to an unseen LLM backend.

On DeepSeek-V4.1-Flash, SoL-Pi improves over Pi-baseline in both task performance and efficiency, increasing the average score from 34.531 to 35.170 while reducing token traffic by 28.3% and API cost by 12.2% (Table 7).

3.2 Generalization to Held-Out Benchmarks

To assess generalization beyond the EdgeBench validation gate, we compare Codex, Pi-baseline, and SoL-Pi on 63 CPU-only tasks from Terminal-Bench 4 (Marten, 2026). Table 3 reports the number of solved tasks, total model cost, and cost per solved task, with all costs reported as API costs. SoL-Pi achieves a solve rate of 23.8%, compared with 28.6% for Pi-baseline, corresponding to a difference of three solved tasks out of 63. Meanwhile, it reduces total API cost by 26.3% and API cost per solved task by 11.6% ($14.07 vs. $15.91).

We further evaluate the three harnesses on IMO 2026 (International Mathematical Olympiad, 2026) using GPT-5.6 Sol with reasoning effort set to `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-baseline. We use the formal problem statements released by Axiom Math (2026) and the verification



(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. Lower cycle counts are better; costs are cumulative API costs.

setup of Humanfia (2026). Each problem has a 150-minute cap; costs include all model activity within that budget. Thus, SoL-Pi matches its baseline (Pi-baseline) in verified solve rate (3/6) while reducing both total API cost and cost per solved problem.

3.3 Efficient Agent Swarms via SoL-Pi

We evaluate SoL-Pi in a multi-agent kernel-optimization experiment, measuring kernel performance in simulated machine cycles, where lower cycle counts indicate better performance (Anthropic, 2026d). 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 with reasoning effort set to `xhigh`; all workers use GPT-5.6 Luna with reasoning effort set to `xhigh`. Every run starts from the same frozen starter, which requires 147,734 cycles, with fresh sessions.

Table 3: **Terminal-Bench 4 and IMO 2026.** Costs are USD; $/pass is cost per solved task or problem. Best values are **bold**.

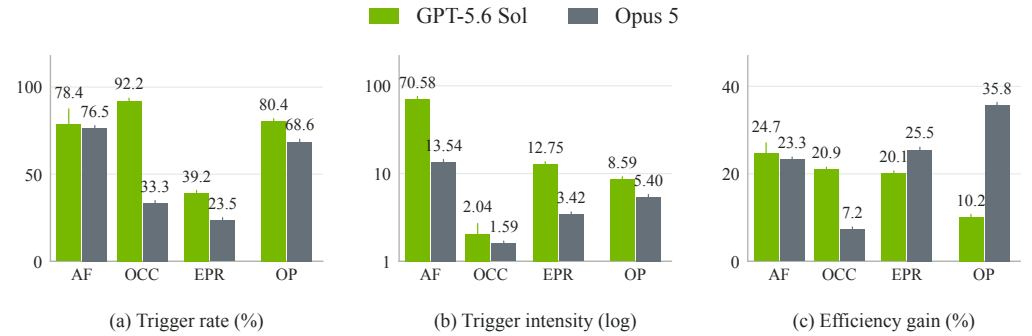
Harness	Pass↑	Cost↓	$/pass↓
Terminal-Bench 4 (63 tasks)			
Codex	**18**	272.35	15.13
Pi-baseline	**18**	286.45	15.91
SoL-Pi	15	**211.12**	**14.07**
IMO 2026 (6 problems)			
Codex	**5**	114.47	22.89
Pi-baseline	3	75.95	25.32
SoL-Pi	3	**62.69**	**20.90**

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.

Under the same two-hour search budget, the SoL-Pi swarm achieves the best kernel performance among the three evaluated configurations (Figure 5(b)). It reaches 1,127 cycles at $60.11, compared with 1,333 cycles at $39.20 for the single Codex agent and 1,366 cycles at $82.12 for the Pi-baseline swarm. This corresponds to 15.5% fewer cycles than the single agent and 17.5% fewer than the Pi-baseline swarm. The SoL-Pi swarm also reduces API cost by 26.8% relative to the Pi-baseline swarm. 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. 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. Additional controls are in appendix B.3.

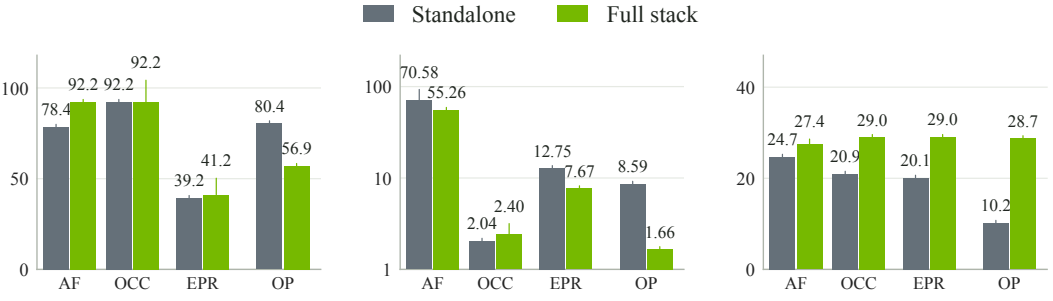
3.4 Learned Mechanisms and Backend Behavior

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-baseline, 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



(a) Trigger rate (%) (b) Trigger intensity (log) (c) Efficiency gain (%)

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. AF: Action Fusion; OCC: Online Context Compact; EPR: Evidence-Preserving Reducer; OP: ObservationPack.



(a) Trigger rate (%) (b) Trigger intensity (log) (c) Efficiency gain (%)

Figure 7: **Standalone mechanisms versus the complete stack on GPT-5.6 Sol.** Paired bars show trigger rate, trigger intensity, and token-efficiency gain on each configuration's own triggered-task subset, relative to its disabled baseline.

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.

Backend-dependent activation. Figure 6 reports trigger rate (the fraction of tasks with an activation), trigger intensity (activations per triggered task), and token-efficiency gain. Mechanisms trigger less often and less intensively on Opus 5, consistent with search using only GPT-5.6 Sol trajectories. Nevertheless, every tested configuration improves token efficiency on its triggered tasks under Opus 5; the fixed stack retains a similar aggregate score–efficiency trade-off across backends (Table 2).

Co-enablement. Figure 7 compares standalone mechanisms with the complete stack. ObservationPack triggers more selectively when combined, potentially reflecting overlap with Evidence-Preserving Reducer. All four mechanisms show larger token-efficiency gains in the stack on their respective triggered-task subsets. These results support the complementarity of the learned mechanisms.

Cache Reuse and Total Cost. Shortening context can reduce prompt-cache reuse when it

Table 4: **Single-component evaluation on EdgeBench.** Each mechanism is added to Pi-baseline; SoL-Pi combines all four. AF/OCC/EPR/OP: Action Fusion, Online Context Compact, Evidence-Preserving Reducer, ObservationPack. †: Performance selection. Bold/underline: best/next-distinct values. Token details: Table 5.

Harness	Tokens (B)↓	Cost ($)↓	Avg. score↑	$/score ↓
GPT-5.6 Sol				
Pi-baseline	2.1538	1,339	44.833	0.5855
+ AF	1.8968	1,235	<u>46.664</u>	0.5190
+ OCC	<u>1.2881</u>	<u>935</u>	41.993	<u>0.4365</u>
+ EPR	1.9375	1,200	44.630	0.5274
+ OP†	2.0224	1,271	**47.208**	0.5280
SoL-Pi [Efficiency]	**1.0990**	**894**	42.003	**0.4174**
Opus 5				
Pi-baseline	2.3697	1,741	44.756	0.7625
+ AF†	2.1016	1,605	**50.482**	0.6235
+ OCC	1.9152	1,537	<u>49.155</u>	0.6130
+ EPR	1.9131	1,456	43.405	0.6578
+ OP	<u>1.4418</u>	<u>1,176</u>	47.047	**0.4899**
SoL-Pi [Efficiency]	**1.3101**	**1,158**	42.224	<u>0.5376</u>

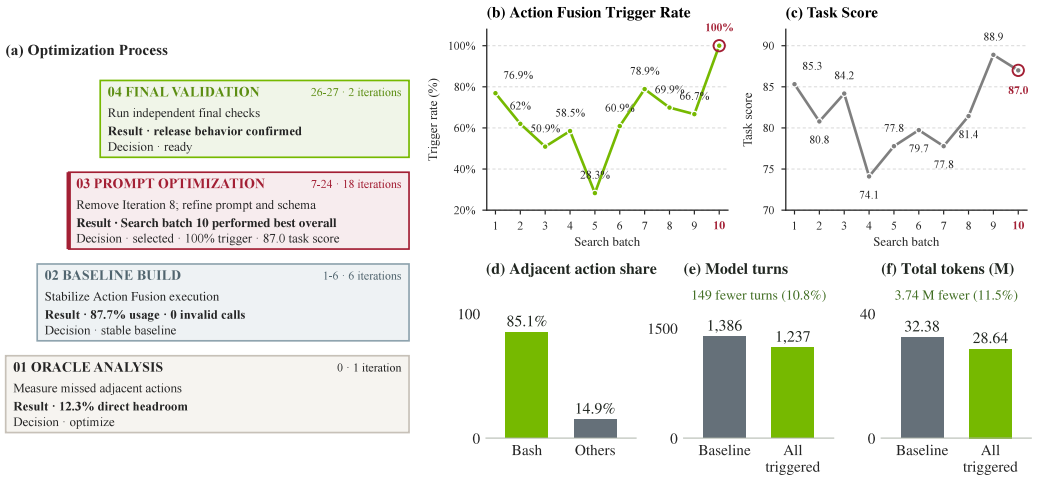


Figure 8: **Action Fusion from opportunity to retention.** (a) 27 recorded iterations across four stages. (b)–(c) Ten retained steps from 18 stage-03 prompt-optimization explorations: only the best candidate from each parallel batch is retained. (d) Observed adjacent-action composition. (e)–(f) Projected model turns and tokens under full triggering.

changes a previously cached prefix (Anthropic, 2026b). 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 5 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-baseline under both backends. These results show that reducing repeated context can improve overall cost efficiency, even at the expense of cache reuse.

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 Conclusion

We introduce SoL-Pi, an RSI-inspired auto-research system for discovering and combining efficient agent-harness mechanisms. On EdgeBench, the highest-scoring single mechanisms improve average score by 5.3–12.8% and token efficiency by 9.8–18.2% relative to Pi-baseline. The fixed complete stack reduces token traffic by 44.7–49.0% and API cost by about one third while retaining 93.7–94.3% of Pi-baseline's average score. Transfer to Opus 5 provides preliminary evidence that harness improvements can generalize beyond the search backend. We present additional related work in appendix A and discuss limitations and future directions in appendix D.

AI USE STATEMENT

Experimental use. As described in Sec. 2, AI contributed to the experimental workflow in the following ways:

- **Harness search and implementation.** LLM agents proposed harness changes, implemented and revised candidates, and reviewed implementations.
- **Environment construction.** AI assisted in constructing verifier-driven synthetic tasks for mechanism discovery.
- **Trajectory analysis.** Independent agents analyzed development trajectories and summarized evidence to guide subsequent experiments.
- **Evaluation and agent execution.** GPT-5.6 Sol and Opus 5 served as primary backends, while GPT-5.6 Luna supported evidence reduction and swarm workers. DeepSeek-V4.1-Flash was used for the additional EdgeBench evaluation reported in Table 7. Roles and settings are specified in Secs. 2.5 and 3.3 and appendix B.2.

Candidate acceptance used executable evaluation with capability and efficiency gates. Human researchers reviewed the final evolved harness and its retained mechanisms.

Manuscript preparation. OpenAI Codex assisted with manuscript organization, LaTeX table and float formatting, and drafting the submission statements. The authors are responsible for the final manuscript, including AI-assisted text, code, claims, and artifacts.

ETHICS STATEMENT

This work studies automated software agents and harness optimization using repository-derived tasks, executable verifiers, and public benchmarks. More efficient agents can reduce the cost of useful software work, but can also make harmful automation easier to scale. Appropriate deployment should restrict tool permissions, isolate execution, preserve audit logs, and retain human oversight for consequential actions. We do not anticipate additional risks beyond those associated with existing agent and harness research.

REPRODUCIBILITY STATEMENT

We provide detailed descriptions of our method and experimental setup to support reproducibility. Sec. 2 describes the search loop, selection criteria, environment construction, implementation boundaries, and four retained mechanisms. Secs. 3 and 3.3 specify the evaluated systems, metrics, and swarm protocol. Appendices B and C provide additional backend settings, benchmark budgets, and the fixed price schedule; Tables 5 and 6 give complete token-category tables. Sec. 3.4 reports activation and co-enablement analyses, and Sec. 3.5 describes the Action Fusion development trace. We will open-source the SoL-Pi harness discovered in this work to support reproducibility.

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.

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

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.

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

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

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

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

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

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.

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

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.

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.

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.

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

DeepSeek. Models & Pricing, 2026. URL https://api-docs.deepseek.com/quick_start/pricing/. USD API off-peak rates accessed September 22, 2026.

DeepSeek-AI. DeepSeek Harness: Everything is a Plugin. GitHub repository, 2026. URL https://github.com/deepseek-ai/deepseek-harness. Accessed September 23, 2026.

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

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.

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*, pp. 10835–10866. PMLR, 2023. URL https://proceedings.mlr.press/v202/gao23h.html.

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

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.

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

Geoffrey Huntley. Ralph Wiggum as a "software engineer". Blog post, July 2025. URL https://ghuntley.com/ralph/. Published July 14, 2025; accessed September 23, 2026.

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

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.

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

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 software tasks. *Advances in Neural Information Processing Systems*, 38:92213–92266, 2026.

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*, pp. 611–626, 2023.

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.

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.

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

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*, pp. 20604–20617, 2026.

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.

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

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

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.

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.

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.

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

OpenAI. Introducing GPT-5.3-Codex. OpenAI blog post, 2026b. URL https://openai.com/index/introducing-gpt-5-3-codex/. Accessed 2026-08-17.

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.

Jürgen Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In Ben Goertzel and Cassio Pennachin (eds.), *Artificial General Intelligence*, pp. 199–226. Springer, 2007.

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, pp. 3841–3865, 2025.

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.

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

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.

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*, pp. 38087–38099. PMLR, 2023.

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.

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.

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.

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.

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, pp. 104223–104294, 2026a.

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

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

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, 2026c.

Deyao Zhu, Xin Zhou, Shengling Qin, Xuekai Zhu, Hangliang Ding, Shu Zhong, Zixin Wen, Zhonglin Xie, Chenhui Gou, Linxuan Ren, et al. Edgebench: Unveiling scaling laws of learning from real-world environments. *arXiv preprint arXiv:2607.05155*, 2026.

A Related Work

A.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 (Yang et al., 2024). Subsequent work has examined interface design, context management, tool use, and delegation as system-level choices that shape agent behavior (Ning et al., 2026; Lumer et al., 2026). These design choices become increasingly consequential in long-horizon tasks, where accumulated interaction histories and environment observations can increase context size and execution overhead (Zhu et al., 2026; Anthropic, 2025).

Automated agent design explores several optimization targets. GEPA improves prompts through reflection on execution trajectories (Agrawal et al., 2026), while ADAS formulates agent design as code search (Hu et al., 2025). AFlow and AgentSquare search over workflow structures and modular agent components, respectively (Zhang et al., 2025; Shang et al., 2025). Related research on recursive self-improvement has theoretical roots in the Gödel Machine (Schmidhuber, 2007). Recent systems explore executable forms of self-modification: the Darwin Gödel Machine iteratively modifies and evaluates agent code (Zhang et al., 2026a), while Hyperagents also make the meta-level improvement procedure itself editable (Zhang et al., 2026b). Our approach draws on this line of research to search for reusable efficiency improvements in harness mechanisms while keeping the underlying model fixed.

A.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 (Lou et al., 2026). Recursive Harness Self-Improvement (RHI) refines prompt-level specifications of the agent loop for individual tasks (Lee et al., 2026a). Meta-Harness searches over executable harness programs using prior code, scores, and execution traces, maintaining a Pareto frontier over task performance and context cost (Lee et al., 2026b). AHE evolves modular coding-harness components from trajectory evidence and evaluates their transfer across tasks and models (Lin et al., 2026). MemoHarness stores execution experience and uses it to adapt harness configurations to individual test cases at inference time (Huang et al., 2026).

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 (Wang et al., 2026). 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.

A.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 (Qi et al., 2026). Other methods reduce the information retained during execution: AgentDiet removes redundant and outdated trajectory content (Xiao et al., 2026), while ACON optimizes the compression of observations and interaction histories (Kang et al., 2025). Context-Folding and AgentFold enable agents to manage their working context through trajectory folding and compression (Sun et al., 2025; Ye et al., 2025), and Context as a Tool exposes context maintenance as an explicit agent action (Liu et al., 2026). Agentic Context Engineering (ACE) focuses on accumulating and refining reusable strategies in context through execution feedback (Zhang et al., 2026c). 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.

B Additional Experimental Protocol

B.1 Search Environments and Evaluation Boundary

The search uses 535 executable environments: 495 repository-derived environments from public repositories and 40 verifier-driven synthetic environments from AutoLoop Open Environments. Figure 3 shows the construction paths. EdgeBench is excluded from search and environment synthesis. Of its 51 public tasks, 11 are used for one-way acceptance of frozen candidates and 40 for final evaluation. The tables report aggregates over all 51 tasks.

B.2 Benchmark and Backend Configuration

The main comparison uses GPT-5.6 Sol with reasoning effort set to `xhigh`. SoL-Pi's Evidence-Preserving Reducer uses GPT-5.6 Luna with reasoning effort set to `high`. In the Opus 5 transfer experiment only the primary backend changes; the auxiliary reducer remains unchanged. Baseline implementations are Codex (OpenAI, 2026a), Claude Code (Anthropic, 2026a), OpenSquilla (TokenRhythm, 2026), Oh-My-Pi (can1357, 2026), OpenCode (Anomaly, 2026), Oh-My-Opencode (code-yeongyu, 2026), and Pi-baseline (Earendil Works, 2026).

We evaluate 63 CPU-only tasks from Terminal-Bench 4, as GPU-dependent tasks require resources outside our evaluation setup. The IMO 2026 evaluation uses six formal problem statements released by Axiom Math (2026) and the verification setup of Humanfia (2026). All IMO runs use GPT-5.6 Sol with reasoning effort set to `xhigh`, require Lean 4 verification, and have a 150-minute cap per problem; costs include all model activity within this budget. The swarm protocol and its two-hour run budget are described in Sec. 3.3.

B.3 Agent-Swarm Controls

Each swarm has five groups of four workers, with independent workspaces and a shared evidence board within each group. Workers exchange notes to reproduce or combine promising findings; the coordinator relays findings across groups. All three configurations start from the same frozen starter, which requires 147,734 simulated machine cycles, with fresh sessions. Each configuration runs for two hours. The shared best result changes only when the coordinator submits an immutable candidate snapshot and independent verification confirms a strict improvement. All four mechanisms are enabled in the SoL-Pi workers. The single agent and coordinators use GPT-5.6 Sol with reasoning effort set to `xhigh`; all workers use GPT-5.6 Luna with reasoning effort set to `xhigh`.

C Token Accounting and Complete Breakdowns

Average score is the arithmetic mean of task scores. For GPT-5.6 Sol and Opus 5, total token traffic is the aggregate recorded input, cache-read, cache-write, and output traffic. For DeepSeek-V4.1-Flash, input includes cache-read and cache-write tokens, and total traffic is input plus output. Token efficiency divides API cost by aggregate task score, so on EdgeBench the denominator is 51 times the average score, not the average score itself. GPT-5.6 Sol and Opus 5 costs use a fixed Standard/global API price schedule checked on August 17, 2026. DeepSeek-V4.1-Flash costs correspond to the official off-peak API rates checked on September 22, 2026 (DeepSeek, 2026).

The frozen schedule assigns GPT-5.6 Sol input, cached input, cache write, and output prices of \$5.00, \$0.50, \$6.25, and \$30.00 per million tokens; the corresponding GPT-5.6 Luna prices are \$0.20, \$0.02, \$0.25, and \$1.20. For Opus 5, input, five-minute cache write, one-hour cache write, cache read, and output prices are \$5.00, \$6.25, \$10.00, \$0.50, and \$25.00. For DeepSeek-V4.1-Flash, the off-peak prices per million tokens are \$0.003 for cache-hit input, \$0.15 for cache-miss input, and \$0.60 for output.

Both Oh-My-Opencode and Oh-My-Pi are evaluated with their subagent systems enabled. In our experiments, Oh-My-Pi occasionally spawned large numbers of subagents through recursive delegation. We therefore imposed a \$100 per-task budget on Oh-My-Pi to keep resource consumption bounded during evaluation.

Table 5: **Token accounting for single-component evaluations.** Recorded tokens (B). Dark green: SoL-Pi/Efficiency; light green and †: Performance selection. Scores and costs: Table 4.

Harness / configuration	Input	Cache read	Cache write	Output	Total
GPT-5.6 Sol					
Pi-baseline	0.0011	2.1326	**0.0141**	0.0059	2.1538
+ Action Fusion	0.0011	1.8718	0.0180	0.0060	1.8968
+ Online Context Compact	0.0008	1.2602	0.0217	**0.0055**	1.2881
+ Evidence-Preserving Reducer	0.0051	1.9089	0.0181	**0.0055**	1.9375
+ ObservationPack †	**0.0002**	2.0005	0.0160	0.0056	2.0224
SoL-Pi [Efficiency]	0.0009	**1.0605**	0.0316	0.0061	**1.0990**
Opus 5					
Pi-baseline	**0.0000**	2.3203	0.0348	0.0145	2.3697
+ Action Fusion †	**0.0000**	2.0520	0.0352	0.0144	2.1016
+ Online Context Compact	**0.0000**	1.8618	0.0388	0.0145	1.9152
+ Evidence-Preserving Reducer	**0.0000**	1.8685	**0.0315**	0.0131	1.9131
+ ObservationPack	**0.0000**	1.3960	0.0356	**0.0102**	1.4418
SoL-Pi [Efficiency]	**0.0000**	**1.2626**	0.0352	0.0123	**1.3101**

Table 6: **Token accounting for the harness comparisons.** Recorded tokens (B); dark/light green mark SoL-Pi/Efficiency and Performance. Scores and costs: Tables 1, 2 and 7.

Harness / configuration	Input	Cache read	Cache write	Output	Total
GPT-5.6 Sol					
Codex	0.0053	3.0287	0.0145	0.0052	3.0537
OpenSquilla	**0.0001**	1.2533	0.0776	**0.0044**	1.3353
Oh-My-Pi	0.1135	2.0448	0.0484	0.0168	2.2235
OpenCode	0.0012	2.2625	0.2865	0.0164	2.5668
Oh-My-Opencode	0.0044	2.4373	0.1286	0.0121	2.5825
Pi-baseline	0.0011	2.1326	**0.0141**	0.0059	2.1538
SoL-Pi [Efficiency]	0.0009	**1.0605**	0.0316	0.0061	**1.0990**
SoL-Pi [Performance]	0.0002	2.0005	0.0160	0.0056	2.0224
Opus 5					
Claude Code	0.0002	1.8116	0.1701	0.0226	2.0045
Pi-baseline	**0.0000**	2.3203	**0.0348**	0.0145	2.3697
SoL-Pi [Efficiency]	**0.0000**	**1.2626**	0.0352	**0.0123**	**1.3101**
SoL-Pi [Performance]	**0.0000**	2.0520	0.0352	0.0144	2.1016
DeepSeek-V4.1-Flash					
Pi-baseline	3.5037	3.4930	**0.0107**	0.0138	3.5175
DSH	4.1556	4.1395	0.0161	0.0225	4.1781
SoL-Pi	**2.5094**	**2.4940**	0.0154	**0.0135**	**2.5229**

Table 7: **Additional EdgeBench results with DeepSeek-V4.1-Flash.** Bold/underline: best/next-distinct values within this backend. Full token accounting is in Table 6.

Harness	Tokens (B)↓	Cost ($)↓	Avg. score↑	$/score↓
Pi-baseline	3.5175	20.38	34.531	0.0116
DSH	4.1781	28.33	**37.625**	0.0148
SoL-Pi	**2.5229**	**17.89**	35.170	**0.0100**

Tables 5 and 6 report token accounting for the single-component evaluations and harness comparisons, respectively. The DeepSeek-V4.1-Flash token breakdowns for Pi-baseline, DeepSeek Harness (DSH; DeepSeek-AI, 2026), and SoL-Pi are reported in Table 6.

C.1 Performance and Efficiency on DeepSeek-V4.1-Flash

Table 7 complements the GPT-5.6 Sol and Opus 5 comparisons with results for Pi-baseline, DSH, and SoL-Pi using DeepSeek-V4.1-Flash. Relative to Pi-baseline, SoL-Pi increases the average score

from 34.531 to 35.170 while reducing total token traffic by 28.3% (3.5175 B to 2.5229 B) and reported cost by 12.2% ($20.38 to $17.89). DSH achieves the highest average score, 37.625, whereas SoL-Pi retains 93.5% of that score with 39.6% less token traffic and 36.9% lower cost. Among these three harnesses, SoL-Pi has the lowest cost per aggregate score point, $0.0100, compared with $0.0116 for Pi-baseline and $0.0148 for DSH. These results demonstrate simultaneous gains in task performance and efficiency over Pi-baseline on DeepSeek-V4.1-Flash.

D 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 pre-training, 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 *Pre-Training 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 Opus 5, 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 direction *recursive efficient improvement* and will explore its potential to compound efficiency gains across successive research cycles.

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.