EVOCOST: COST-EFFECTIVE AGENTS VIA HARNESS SELF-EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents can incur substantial inference cost on complex tasks that require long interaction trajectories. Existing cost-saving methods mainly compress accumulated history or control execution stopping, but do not change the behavior that generates unnecessary interactions. We introduce EVOCOST, a framework for *cost-effective harness self-evolution* that learns from execution experience to reduce unnecessary interactions in future executions while preserving task performance. EVOCOST first performs *cost-saving opportunity discovery* by reconstructing execution dependencies and combining them with execution outcomes to identify interactions that can be safely skipped, collapsed, batched, or redirected. It then performs *validated harness evolution*, translating these opportunities into reusable harness edits and retaining updates that reduce inference cost while meeting stage-dependent task-performance requirements. Across four benchmarks and three model backbones, EVOCOST reduces inference cost relative to the original harness in all 12 settings by 22.7% on average, while improving or preserving task performance in 10 of them. These results demonstrate that harness self-evolution can provide a practical path toward more cost-effective agent execution[1].

1 INTRODUCTION

LLM agents can solve increasingly complex tasks through repeated reasoning, tool use, and environment interaction, but this capability comes at a growing inference cost (Yao et al., 2023; Yang et al., 2024). The cost becomes particularly significant for long-horizon tasks that require many interaction steps (Huang et al., 2026): in a 2026 Stack Overflow survey, 38% of developers reported cost as a barrier to using AI agents (Stack Overflow, 2026). This raises a key question for practical agent deployment: *how can we substantially reduce inference cost while preserving task performance?*

Existing methods mainly reduce inference cost *after* agent interactions have been generated, either by compressing accumulated history (Xiao et al., 2026; Pan et al., 2026; Ren et al., 2026) or terminating iterations with little expected benefit (Guo et al., 2026). Such methods can reduce subsequent computation, but cannot recover the cost of interactions that have already been incurred. Moreover, history compression may discard information needed for later decisions (Pan et al., 2026), while rewriting accumulated context can reduce prefix cache reuse (Zheng et al., 2024). A more fundamental alternative is to avoid unnecessary interactions *before* they are generated by changing the agent behavior that produces them.

The behavior of an agent is largely governed by its *harness*, which comprises reusable skills, tools, instructions, and executor code that collectively determine how the agent completes a task (Wang et al., 2024; Yang et al., 2024; Lee et al., 2026). We refer to modifications of these reusable components as *harness edits*. *Harness evolution* iteratively improves an agent by proposing, evaluating, and retaining harness edits using execution traces and outcome feedback (Khattab et al., 2024; Hu et al., 2025; Zhang et al., 2025; Agrawal et al., 2026; Lee et al., 2026; Xu et al., 2026). Unlike post-hoc cost reduction, which acts after inefficient interactions have already occurred, harness evolution can reshape future agent behavior and prevent unnecessary interactions before their cost is incurred. However, existing harness evolution methods primarily optimize task performance, leaving inference cost largely unexplored.

[1]Code are available in supplementary materials.

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.

Our Proposal. Optimizing a harness for cost is fundamentally different from optimizing it for task performance. It requires identifying which interactions can be safely avoided and translating these opportunities into reusable harness edits. To this end, we propose EvoCost, a framework for cost-effective harness self-evolution.

The first step is *cost-saving opportunity discovery*. Rather than relying on local cost signals alone, EvoCost reasons about execution dependencies to distinguish costly interactions from those necessary for downstream decisions. Specifically, EvoCost constructs a support directed acyclic graph (DAG) from execution traces to capture how intermediate outputs support subsequent decisions, and combines these dependencies with execution outcomes to identify interactions that can be safely skipped, collapsed, batched, or redirected. The resulting opportunities provide concrete evidence for generating reusable harness edits.

The second step is *validated harness evolution*. Individually promising harness edits may interact or lose their benefits when combined, so EvoCost validates edits at both the individual and composed levels. EvoCost uses Parallel Patch Generation to produce independent harness edits from the same harness snapshot, and then employs a merging agent to reconcile compatible edits across harness components. EvoCost further applies Hybrid Gating to filter unsupported edits and compare the merged harness against the current one under stage-dependent cost and task-performance thresholds. Only validated updates are retained, and the resulting execution trajectories initiate the next round of cost-saving opportunity discovery, closing the self-evolution loop.

The contributions can be summarized as below.

- We introduce *cost-effective harness self-evolution* as a new approach to agent cost reduction, shifting optimization from post-hoc trace reduction to proactive improvement of future agent behavior.
- We propose EvoCost, which combines *cost-saving opportunity discovery* with *validated harness evolution* to identify safely avoidable interactions and convert them into reusable harness edits.
- We evaluate EvoCost across four benchmarks and three model backbones. EvoCost reduces inference cost in all 12 settings by 22.7% on average, while improving or preserving task performance in 10 of them.

2 Problem Formulation

An LLM agent combines a language model with a *harness* that governs how the model interacts with tools and the environment (Yao et al., 2023; Yang et al., 2024). We represent the harness as $\mathcal{H} = (I, S, \Omega, E)$, consisting of instructions, procedural skills, tool interfaces, and executor code. The harness constructs model inputs, executes actions, and determines when the agent terminates.

For a task x, the agent produces an execution trajectory $\tau(x; \mathcal{H}) = ((a_t, o_t))_{t=1}^{T}$, where a_t denotes the agent action at step t, including any tool calls, and o_t is the resulting observation. *Harness evolution* iteratively updates $\mathcal{H}$ between executions using previous trajectories and outcome feedback. Each update produces a new harness $\mathcal{H}'$, while the underlying language-model parameters remain fixed. The goal is therefore not to modify the model itself, but to improve future agent behavior through reusable changes to the harness.

Let $\mathcal{U}(\tau, x)$ be the benchmark-verified task score. Inference cost includes all primary and auxiliary calls, distinguishing uncached-input, cached-input, and output tokens:

$$\mathcal{C}(\tau) = \sum_{t=1}^{T} \left(p_{\mathrm{u}} n_t^{\mathrm{u}} + p_{\mathrm{c}} n_t^{\mathrm{c}} + p_{\mathrm{o}} n_t^{\mathrm{o}} \right), \tag{1}$$

where n_t^q and p_q are the token count and per-token price for category q. This distinction matters because a shorter history need not be cheaper if rewriting it loses cached prefixes (Zheng et al., 2024). Our objective over deployment tasks $\mathcal{D}$ is

$$\min_{\mathcal{H}} \ \mathbb{E}_{x \sim \mathcal{D}}[\mathcal{C}(\tau(x; \mathcal{H}))] \quad \text{s.t.} \quad \mathbb{E}_{x \sim \mathcal{D}}[\mathcal{U}(\tau(x; \mathcal{H}), x)] \geq U_{\min}, \tag{2}$$

where $U_{\min}$ is the minimum acceptable task score. The objective concerns future execution, not merely shorter recorded trajectories. We use finite samples to guide the search (Dwork et al., 2015).

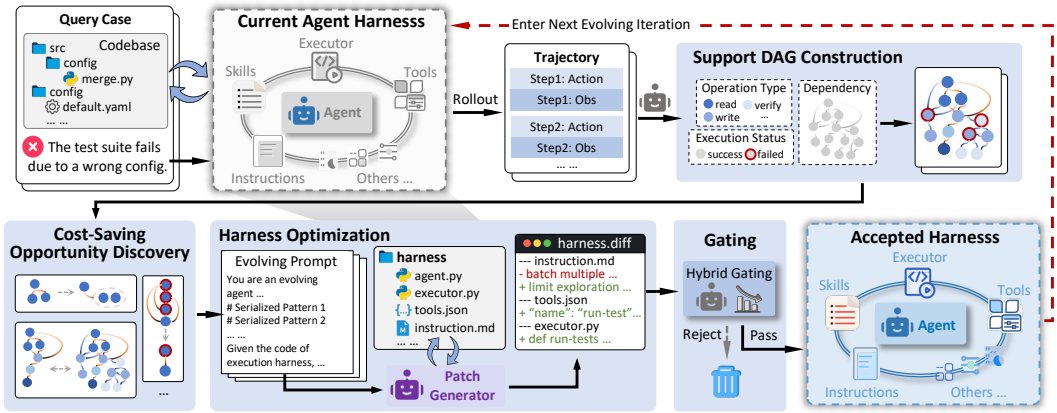


Figure 1: Overview of EvoCost.

3 EvoCost

EvoCost alternates between cost-saving opportunity discovery and validated harness evolution. As shown in Figure 1, accepted harnesses generate the trajectories used in the next cycle.

3.1 Cost-Saving Opportunity Discovery

Reducing inference cost requires distinguishing avoidable work from interactions that support task completion. Agents select tools and revise plans in response to feedback rather than follow a fixed workflow. Execution order and repeated calls alone do not reveal which results later decisions require. EvoCost reconstructs these dependencies to identify potential savings.

Support DAG Construction. EvoCost annotates each interaction with its operation type, execution status, touched resources, and outcome signature. We use a language-model annotator that observes only the task and model-visible trajectory prefix, linking an earlier node v_s to v_t when information or an artifact produced by v_s is directly used at v_t. Detailed annotation fields and construction rules are provided in Appendix A.

For answer-submission or terminal-validation nodes $\mathcal{V}_{\text{term}}$, the dependency closure is

$$\mathcal{K}(\tau) = \mathcal{V}_{\text{term}} \cup \text{Anc}_{G_\tau}(\mathcal{V}_{\text{term}}), \tag{3}$$

where $\text{Anc}_{G_\tau}(\cdot)$ returns the ancestors of the supplied nodes in G_τ. Cost identifies where execution is expensive, while dependency structure indicates which outputs support later decisions. An interaction inside the closure may still contain redundant work. The resulting edit should preserve its useful outputs while narrowing or combining the operations that produce them.

Intervention-oriented patterns. Figure 2 shows how trajectory evidence guides candidate edits. *Skippable* segments contain at least two consecutive interactions outside $\mathcal{K}(\tau)$, suggesting reusable conditions for skipping similar interactions. *Collapsible* paths contain successful serial operations whose intermediate results are consumed only within the path. A bounded composite tool may replace them while preserving the final state and downstream information. *Mergeable* groups contain two to four successful operations with identical direct predecessors and no recorded interdependence. Compatible outputs and disjoint write resources support batched execution. *Failure-to-Pivot* segments contain consecutive failures followed by recovery, motivating earlier redirection to a different approach under similar conditions. These patterns guide edits, which are extendable instead of a complete classification.

In Figure 2(b), the reads of files A and B share a predecessor and have no recorded dependency on each other, suggesting a batched read. In contrast, a read that discovers the next file's path introduces a serial dependency that batching must respect. The representation thus suggests a particular harness intervention, not merely a list of expensive steps. Failure-to-Pivot retains failed attempts and recovery to recognize stalled approaches rather than impose a task-independent retry limit.

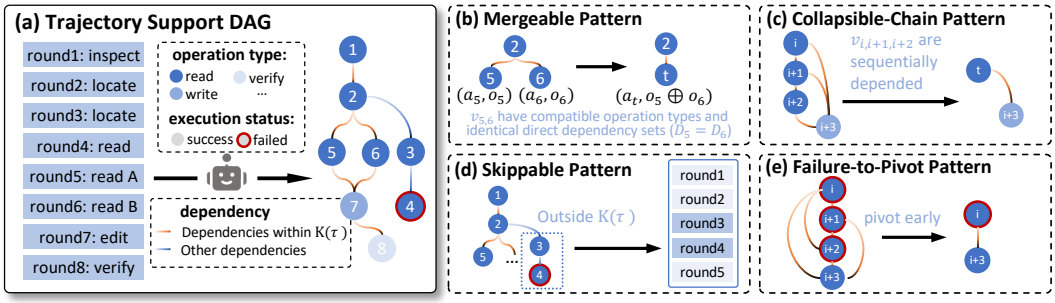


Figure 2: Dependency and outcome evidence for candidate harness interventions.

To guide reusable edits, EvoCost groups recurring instances into evidence batches that retain their source interactions, dependencies, and outcomes. Cost-saving opportunity discovery supplies evidence for candidate edits. During validated harness evolution, Hybrid Gating tests whether the combined harness reduces inference cost while meeting the task-performance criterion.

3.2 Parallel Patch Generation

Validated harness evolution translates the discovered opportunities into compatible edits across harness components. For example, batching file reads requires an instruction specifying when to batch, a compatible tool schema, and executor code that returns the requested results together. Each proposal translates an observed opportunity into a reusable condition and a corresponding harness change. For batching, the condition is that the required inputs are already known and the operations are independent. The edit encodes when to batch, rather than which files to read in one recorded task. Sequential editing makes later proposals depend on earlier modifications. Independent generation removes this dependence, but patches may still conflict. EvoCost therefore separates proposal generation from integration. Each generator edits the same snapshot, and a merger checks compatibility among retained proposals.

For evidence batches $\mathcal{P}_j$, generators produce isolated diffs Δ_j from the current harness $\mathcal{H}$. A merger receives the retained indices $\mathcal{J}$ after patch screening and constructs the candidate:

$$\Delta_j = A_{\text{evo}}(\mathcal{H}, \mathcal{P}_j), \qquad \widetilde{\mathcal{H}} = \mathcal{H} \oplus A_{\text{merge}}(\mathcal{H}, \{(\Delta_j, \mathcal{P}_j)\}_{j \in \mathcal{J}}), \tag{4}$$

where $\oplus$ applies the merged diff. No generator modifies the shared snapshot, and each proposal remains linked to the evidence that motivated it.

The merger reconciles overlapping source edits, checks agreement across instructions, skills, tool schemas, and executor code, and removes duplicated or superseded behavior. If proposals cannot all be reconciled, it retains a compatible subset. Figure 3 illustrates the integration process.

Structural compatibility alone does not establish that an edit will save inference cost or retain task performance. We therefore need a screening and validation mechanism that checks both the evidence for each proposal and the behavior of the merged harness.

3.3 Harness Validation

A plausible edit can fail beyond its source trajectory, and individually reasonable patches can interact after merging. Yet executing every proposal separately spends rollout budget without testing the combined harness. Hybrid Gating separates these concerns. Patch-level screening rejects invalid or narrowly targeted edits without environment rollouts, while iteration-level validation measures the merged candidate's cost and task performance.

Patch-level screening. Patch-level screening removes proposals that are either invalid harness modifications or insufficiently reusable beyond their source trajectories. Deterministic checks reject patches that cannot be applied or parsed, while an LLM judge (Zheng et al., 2023) filters task-specific or weakly supported edits by assessing whether the proposed behavior is reusable and justified by the source evidence. Only retained patches proceed to joint merging and rollout-based evaluation. Screening considers both the proposed savings and the overhead introduced by the edit. Additional

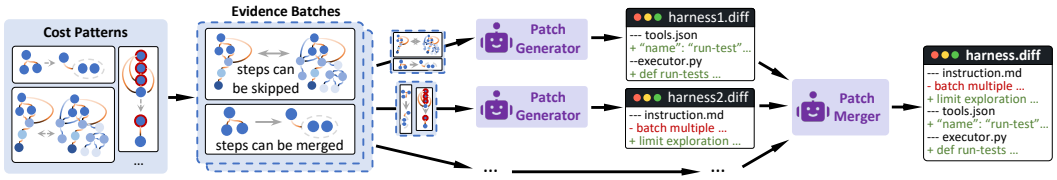


Figure 3: Parallel patch generation from a shared harness snapshot, followed by joint merging.

instructions and tool definitions are processed in future model calls, so removing an interaction does not necessarily reduce inference cost.

Iteration-level validation. The candidate $\mathcal{H}_1$ and current harness $\mathcal{H}_0$ run on identical validation tasks with matched model configurations, environments, and resource limits. Mean task score $\widehat{\mathcal{U}}$ includes every task, assigning zero to execution errors or missing runs. Mean cost $\widehat{\mathcal{C}}$ is computed on their common error-free subset $\mathcal{O}$, preventing infrastructure failures from appearing as savings. At stage r, acceptance requires

$$|\mathcal{O}| > 0, \qquad \widehat{\mathcal{U}}(\mathcal{H}_1) \geq p_r \widehat{\mathcal{U}}(\mathcal{H}_0), \qquad \frac{\widehat{\mathcal{C}}(\mathcal{H}_1) - \widehat{\mathcal{C}}(\mathcal{H}_0)}{\widehat{\mathcal{C}}(\mathcal{H}_0)} \leq \gamma_r, \tag{5}$$

where $p_r \in (0, 1]$ is the score-retention floor and $\gamma_r < 0$ the relative cost-change threshold.

The staged thresholds balance exploration with stricter acceptance as the harness improves. Early stages permit limited score loss to explore savings. As accepted updates accumulate, semantic screening, score retention, and cost-reduction requirements tighten, so later edits must justify their additional changes with stronger validation results. A rejected candidate leaves both the harness and active stage unchanged, so merely attempting more edits does not advance the acceptance schedule. An accepted harness supplies new trajectories for discovery, and evolution ends after final-stage acceptance or exhaustion of the cycle budget. These local checks guide the search rather than guarantee held-out performance: tolerated losses can accumulate across accepted updates.

4 Experiments

4.1 Experimental Setup

Benchmarks. We evaluate four agent benchmarks through Harbor (Harbor Framework Team, 2026). Table 1 summarizes the task pools used for harness development and held-out evaluation. SWE-bench Verified (SB), referred to as SWE-bench, evaluates repository issue resolution using instance-specific tests (Jimenez et al., 2024). DeepSWE v1 (DS) contains long-horizon software-engineering tasks drawn from 91 repositories and five programming languages (Huang et al., 2026). The Data Agent Benchmark (DAB) evaluates questions over heterogeneous data systems using hidden reference answers and validators (Ma et al., 2026). Terminal-Bench 2.1 (TB) evaluates containerized terminal workflows spanning software engineering, system administration, and data science (Merrill et al., 2026; Terminal-Bench Team, 2026). For each benchmark, Reserved denotes the tasks used for both trajectory discovery and iteration-level validation during harness development, whereas Held-out denotes the evaluation pool used to assess the resulting harness.

Table 1: Benchmark task pools for harness development and held-out evaluation.

Benchmark	Reserved	Held-out
SWE-bench	16	484
DeepSWE	16	97
DAB	16	88
Terminal-Bench 2.1	16	73

Baselines. We compare EvoCost against the Original harness, three runtime cost-control methods, and two harness-evolution methods. Original is the unmodified `mini-swe-agent` harness (SWE-agent Team, n.d.). AgentDiet removes redundant trajectory content through reflection (Xiao et al., 2026). ZipAct maintains a compact state representation instead of retaining the complete interaction

Table 2: Task performance and inference cost across four benchmarks. Scores are percentages and costs are in $/task. The second EvoCost row gives changes relative to Original. Blue denotes improvement and orange denotes deterioration.

Method	SWE-bench		DeepSWE v1.0		DAB		Terminal-Bench 2.1	
	Resolved ↑	Cost ↓	Resolved ↑	Cost ↓	Accuracy ↑	Cost ↓	Pass ↑	Cost ↓
(a) DeepSeek-V4-Flash								
Original	68.18	0.0391	**11.34**	0.2963	**53.41**	0.0224	49.32	0.0478
AgentDiet	64.05	0.0453	0.00	0.3714	38.64	0.0498	47.95	0.0517
EET	67.77	0.0355	0.00	0.5024	50.00	0.0195	50.68	0.0501
ZipAct	6.20	0.0529	0.00	0.4024	1.14	0.0303	4.11	0.0648
GEPA	71.69	0.0414	5.15	0.2305	50.00	0.0214	52.05	0.0471
Meta-Harness	72.93	0.0464	5.15	0.2726	48.86	0.0192	53.42	0.0533
EvoCost	**74.38**	**0.0347**	**11.34**	**0.1306**	52.27	**0.0173**	**56.16**	**0.0467**
vs. Original	+6.20	−11.31%	+0.00	−55.91%	−1.14	−23.14%	+6.85	−2.40%
(b) GPT-5.6-Terra								
Original	**76.65**	0.3479	30.93	1.0210	45.45	0.2243	64.38	0.3272
AgentDiet	71.28	0.5298	20.62	1.4360	40.91	0.3713	60.27	0.4601
EET	68.60	0.2198	22.68	0.6365	**50.00**	0.2126	65.75	0.4031
ZipAct	61.16	0.7186	0.00	1.5058	31.82	0.8723	45.21	0.6479
GEPA	71.07	0.2107	19.59	0.7526	45.45	0.1896	**68.49**	0.4870
Meta-Harness	69.42	**0.1848**	26.80	0.9906	47.73	0.2104	67.12	0.4979
EvoCost	72.93	0.1950	**31.96**	**0.6042**	45.45	**0.1695**	67.12	**0.2613**
vs. Original	−3.72	−43.93%	+1.03	−40.82%	+0.00	−24.45%	+2.74	−20.16%
(c) GPT-5.5								
Original	70.45	1.2708	47.42	6.4541	63.64	1.1426	65.75	0.9993
AgentDiet	65.70	1.3974	37.11	5.8077	63.64	1.5991	67.12	0.8966
EET	60.95	1.0872	46.39	6.9893	**65.91**	1.2190	**69.86**	1.1219
ZipAct	43.80	6.8372	5.15	**2.3463**	25.00	5.9259	43.84	0.9765
GEPA	75.62	1.4602	**49.48**	8.8095	62.50	1.1854	65.75	1.0786
Meta-Harness	74.38	1.3626	48.45	7.0791	62.50	1.2601	**69.86**	1.2949
EvoCost	**76.03**	**1.0663**	48.45	5.6965	63.64	**1.0466**	**69.86**	**0.8597**
vs. Original	+5.58	−16.09%	+1.03	−11.74%	+0.00	−8.40%	+4.11	−13.97%

history (Pan et al., 2026). EET uses retrieved experience and confidence-based checks for early termination (Guo et al., 2026). We evaluate Harbor-compatible implementations of these methods.

The harness-evolution baselines are GEPA, which uses reflective mutation and Pareto-based selection (Agrawal et al., 2026), and Meta-Harness, whose proposer conditions on historical code, scores, and execution traces (Lee et al., 2026). We adapt both methods to optimize inference cost together with task performance. For a fair comparison, we control the external search conditions rather than equalizing the internal optimization mechanisms of the methods. Within each benchmark and backbone setting, GEPA, Meta-Harness, and EvoCost use the same initial harness, the same 16 reserved tasks, the same editable harness artifacts, and identical upper limits on evolution cycles. Candidate generation, parent selection, and acceptance remain method-specific parts of each evolution algorithm. Appendix B.1 details these adaptations and protocols.

The main experiments use DeepSeek-V4-Flash (DeepSeek-AI, 2026), GPT-5.5 (OpenAI, 2026a), and GPT-5.6-Terra (OpenAI, 2026b). Within each comparison, we fix the model version, reasoning configuration, token limits, sampling settings, environment, and resource limits.

Metrics. Task performance is measured by resolved-task percentage on SWE-bench and DeepSWE, validator accuracy on DAB, and pass rate on Terminal-Bench 2.1. Inference cost is the mean expenditure on model API calls per held-out task ($/task), computed from all model calls executed during task rollout according to Equation 1. We account separately for uncached-input, cached-input, and output tokens based on the official price listed in Appendix B.3. Failed tasks remain in the score denominator, and calls made before failure count toward inference cost.

4.2 Overall Comparison

Table 2 compares EvoCost with the Original harness and five cost-control or harness-evolution baselines across four benchmarks and three model backbones. The results evaluate whether cost-effective harness self-evolution reduces inference cost while retaining task-solving capability.

Task Performance. EvoCost achieves the highest available task score in six of the twelve settings, including four strict wins and two ties. The advantage is most consistent with DeepSeek-V4-Flash. On SWE-bench, EvoCost reaches a resolved rate of 74.38%, exceeding Meta-Harness by 1.45 points. On Terminal-Bench 2.1, it achieves 56.16%, which is 2.74 points above Meta-Harness. EvoCost also remains competitive with other LLMs. On SWE-bench, EvoCost obtains the highest resolved rate of 76.03% with GPT-5.5, compared with 75.62% for GEPA. With GPT-5.6-Terra, EvoCost achieves the best DeepSWE score. These results show that the lower inference cost of EvoCost does not come from systematically lower task performance than the evolution baselines.

Inference Cost. Cost reduction is substantially more consistent. As reported in Table 2, EvoCost reduces mean inference cost relative to Original across all twelve settings and has the lowest reported cost among the compared methods in ten settings. The unweighted mean and median relative cost reductions across these settings are 22.69% and 18.12%, respectively. The largest reduction occurs on DeepSWE with DeepSeek-V4-Flash, where cost falls by 55.91% while the resolved rate remains unchanged. With GPT-5.5, EvoCost reduces SWE-bench cost while improving the resolved rate by 5.58 points. The cost improvements also extend to GPT-5.6-Terra. On DeepSWE, it reduces cost by 40.82% while achieving the best task score. These results demonstrate that the benefit of EvoCost is not limited to a particular model scale or benchmark, although the resulting cost and performance trade-off varies across settings.

4.3 Ablation Analysis

We study Parallel Patch Generation and Hybrid Gating within validated harness evolution using GPT-5.5 on SWE-bench and DeepSWE. Within each benchmark, the compared variants share the initial harness, development tasks, and proposal budget. Figure 4 summarizes statistics collected during the evolution process, while Table 3 reports the final held-out evaluations. Acceptance rate is the fraction of evolution cycles that yield an accepted harness update. Mean score gain and mean cost reduction are computed over accepted updates relative to the current harness immediately before each update. The Parallel and Hybrid labels correspond to the same full-method runs in the two comparisons. Each variant is evolved once per benchmark.

Table 3: Task score and inference cost in ablations with GPT-5.5. Annotations show changes relative to EvoCost.

Benchmark	Variant	Score (%) ↑	Cost ($/task) ↓
SWE-bench	EvoCost	**76.03**	**1.0663**
	With Sequential	71.28 (−4.75)	1.0935 (+2.55%)
	With Hard	74.79 (−1.24)	1.2906 (+21.03%)
DeepSWE	EvoCost	**48.45**	**5.6965**
	With Sequential	46.39 (−2.06)	5.8180 (+2.13%)
	With Hard	47.42 (−1.03)	6.0101 (+5.51%)

Effect of Parallel Patch Generation. To evaluate Parallel Patch Generation, we compare the full method with a sequential variant while keeping Hybrid Gating unchanged. The full method generates proposals independently from the same harness snapshot and reconciles them through joint merging, whereas the sequential variant immediately applies each retained edit so that later proposals are conditioned on earlier modifications. In Figure 4, Parallel Patch Generation increases update acceptance from 15% to 25% and yields larger mean score gains and cost reductions among accepted updates. Table 3 shows corresponding held-out improvements on both benchmarks. On SWE-bench, EvoCost raises the resolved rate from 71.28% to 76.03% while lowering cost.

Effect of Hybrid Gating. We keep Parallel Patch Generation fixed and replace Hybrid Gating with a hard gate that accepts a candidate whenever its task score does not decrease and its cost strictly decreases. This variant removes both patch-level semantic screening and the staged iteration-level acceptance margins. In Figure 4, Hybrid Gating accepts fewer updates than the hard gate, but their

mean cost reduction is larger at 26.01% versus 9.99%, alongside larger score gains. It accepts no update with less than 5% cost reduction, whereas the hard gate accepts three such updates.

The final evaluations also favor the full gating design on both benchmarks. On SWE-bench, EvoCost improves task performance by 1.24 points and reduces cost by 17.38% relative to Hard. These results indicate that the combined screening and staged-validation procedure produces a stronger final harness in the reported runs.

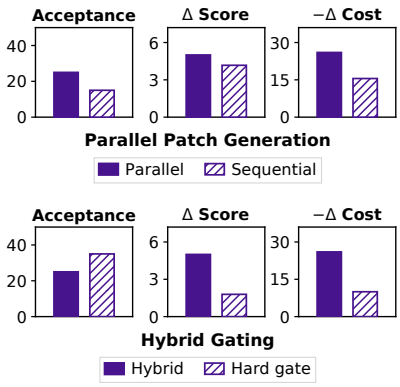


Figure 4: Evolution-process ablations with GPT-5.5. Upper and lower rows compare Parallel Patch Generation and Hybrid Gating.

Evidence Representation. We next examine whether the benefit of EvoCost comes merely from organizing trajectory information and exposing cost hotspots, or from explicitly modeling downstream dependencies. Raw supplies chronological actions and observations. StrucAno adds operation types, execution status, resources, outcome summaries, cost information, and ordinary trajectory-level analysis, but no dependency edges or dependency-derived patterns. DAG-based evidence further models direct dependencies and derives dependency-aware patterns from these annotations. We fix the runtime, initial harness, trajectories, development tasks, optimization components, and ten-cycle budget.

Figure 5 shows that StrucAno reduces cost relative to Raw by 6.36% on SWE-bench and 7.95% on DeepSWE, but its score falls from 72.93% to 72.73% on SWE-bench and rises from 44.33% to 45.36% on DeepSWE. DAG-based evidence improves both objectives relative to StrucAno, increasing scores by 3.30 and 3.09 points while reducing cost by a further 3.74% and 3.15%, respectively. It yields four accepted updates on each benchmark, compared with two for StrucAno on each and two and one for Raw.

These results suggest that trajectory organization and cost information alone do not explain the gain. Cost hotspots identify expensive interactions, but not whether their outputs remain useful downstream. Dependency evidence helps distinguish opportunities to narrow an operation from proposals to remove it entirely. Appendix D.3 illustrates this distinction in a DeepSWE run..

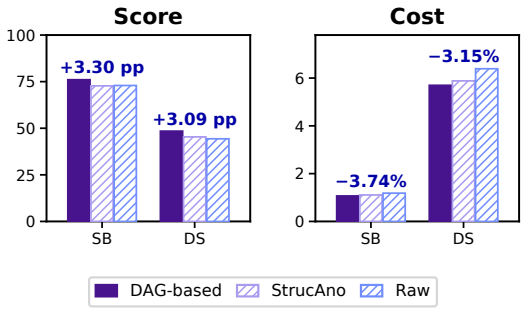


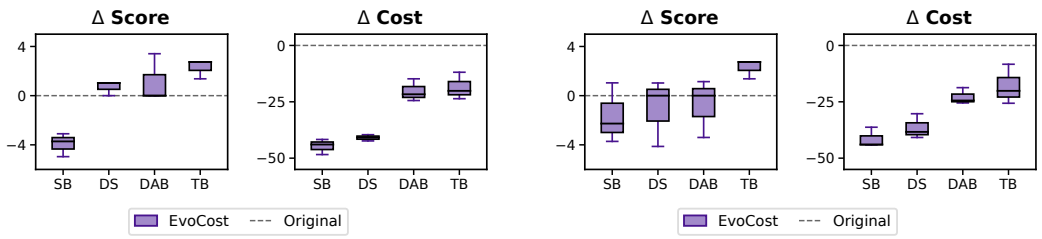
Figure 5: Evidence representations with GPT-5.5. Score is in % and cost in $/task. Blue labels compare DAG-based evidence with StrucAno.

4.4 Sensitivity Analysis

Figure 6 compares three seeds using fixed evolution cases in (a) and different evolution cases in (b). Both panels use the same axes and show changes relative to Original, in percentage points for score and percent for cost. Boxes show quartiles and medians, with whiskers spanning all three runs.

Stability Across Evolution Seeds. We evaluate three evolution seeds with GPT-5.6-Terra while keeping the evolution cases fixed. Figure 6(a) shows that every run reduces inference cost relative to Original on all four benchmarks. Scores match or exceed Original on DeepSWE, DAB, and Terminal-Bench across all seeds, whereas SWE-bench consistently trades lower cost for lower score. DAB exhibits the largest score spread, while DeepSWE varies little in either metric. The repeated runs support consistent cost savings under the fixed development set.

Sensitivity to Evolution Cases. We further vary both the evolution cases and the seed across three runs with GPT-5.6-Terra, sharing the reference run with the fixed-case study. Figure 6(b) shows lower



(a) Fixed evolution cases (b) Different evolution cases

Figure 6: Sensitivity to evolution seeds and cases with GPT-5.6-Terra.

inference cost than Original in every run on all four benchmarks. Score variation is larger than in the fixed-case study on SWE-bench, DeepSWE, and DAB, with outcomes on both sides of Original, whereas Terminal-Bench consistently retains a positive score gain. Cost savings persist across these runs. This comparison captures joint variation in development tasks and search randomness.

5 Related Work

Cost-Saving Methods. Cost-saving methods use batch prompting, model cascades, and context compression (Fan et al., 2023; Chen et al., 2023; Jiang et al., 2023; Pan et al., 2024). Efficient Agents selects model, planning, tool, and memory configurations to balance cost and performance (Wang et al., 2025). AgentDiet removes useless, redundant, and expired trajectory content through periodic reflection to reduce repeated processing of accumulated histories, while ZipAct maintains a compact goal, world, and constraint state (Xiao et al., 2026; Pan et al., 2026). SWE-Pruner selects code lines according to reading goals, and TACO combines task-level rule refinement with a cross-task rule pool under a fixed executor to retain useful terminal evidence (Wang et al., 2026; Ren et al., 2026). EET uses retrieved issue-resolution experience to terminate low-yield iterations (Guo et al., 2026). Our focus is to prevent avoidable interactions through reusable harness changes, complementing context reduction and stopping control.

Harness Evolution. Unlike agentic training, which changes model parameters (Fan et al., 2026; Schick et al., 2023), harness evolution revises agent components using execution feedback. GEPA combines rollout reflection with instance-level Pareto selection to preserve complementary prompts, while Meta-Harness retrieves historical code, scores, and complete traces to guide proposals (Agrawal et al., 2026; Lee et al., 2026). AutoHarness synthesizes action-checking programs to guard against illegal actions, and Life-Harness evolves environment contracts and execution interfaces (Lou et al., 2026; Xu et al., 2026). Self-Harness combines failure clustering with targeted edits and regression tests, while HarnessBank maintains diverse candidates under staged validation (Zhang et al., 2026; Luo et al., 2026). Harness-R1 trains an engineer through supervision and execution-feedback reinforcement learning, whereas HELIX connects verified interventions to model updates through typed composition and provenance records (Shao et al., 2026; Fan & Huang, 2026). EvoCost focuses on cost-effective harness self-evolution, combining cost-saving opportunity discovery with validated harness evolution to reshape future execution with a fixed model.

6 Conclusion

We introduced EvoCost, a framework for cost-effective harness self-evolution that shifts cost reduction from compressing past interactions to improving future agent behavior. Cost-saving opportunity discovery reconstructs execution dependencies and combines them with outcomes to distinguish potentially avoidable work from interactions that support later decisions. Validated harness evolution turns these opportunities into reusable edits and checks their individual and combined effects under stage-dependent cost and task-performance requirements. Together, these steps form a feedback loop that learns from execution experience without changing model parameters. Across four benchmarks and three model backbones, EvoCost reduces inference cost in all 12 settings by 22.7% on average, while improving or preserving task performance in 10. These results support proactive harness improvement as a practical path toward cost-effective agent execution.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, A. Dimakis, Ion Stoica, Dan Klein, Matei A. Zaharia, and O. Khattab. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. In *International Conference on Learning Representations*, 2026. doi: 10.48550/arXiv.2507.19457. URL https://arxiv.org/abs/2507.19457.

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

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. Generalization in adaptive data analysis and holdout reuse, 2015. URL https://arxiv.org/abs/1506.02629.

Meihao Fan, Xiaoyue Han, Ju Fan, Chengliang Chai, Nan Tang, Guoliang Li, and Xiaoyong Du. Cost-effective in-context learning for entity resolution: A design space exploration, 2023. URL https://arxiv.org/abs/2312.03987.

Meihao Fan, Ju Fan, Yuxin Zhang, Shaolei Zhang, Xiaoyong Du, Jie Song, Peng Li, Fuxin Jiang, Tieying Zhang, and Jianjun Chen. DeepPrep: An LLM-powered agentic system for autonomous data preparation, 2026. URL https://arxiv.org/abs/2602.07371.

Tianyu Fan and Chao Huang. HELIX: Model-harness co-evolution for recursive self-improvement, 2026. URL https://arxiv.org/abs/2608.13951.

Yaoqi Guo, Ying Xiao, Jie M. Zhang, Mark Harman, Yiling Lou, Yang Liu, and Zhenpeng Chen. EET: Experience-driven early termination for cost-efficient software engineering agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 33008–33024. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.1652. URL https://aclanthology.org/2026.findings-acl.1652/.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, 2026. URL https://github.com/harbor-framework/harbor.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/36b7acf6f6010652b3f2a433774a66fe-Paper-Conference.pdf.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026. URL https://arxiv.org/abs/2607.07946.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LLMLingua: Compressing prompts for accelerated inference of large language models. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 13358–13376, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.825. URL https://aclanthology.org/2023.emnlp-main.825/.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

O. Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into state-of-the-art pipelines. 2024.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-End Optimization of Model Harnesses, 2026. URL https://arxiv.org/abs/2603.28052.

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, 2026. URL https://arxiv.org/abs/2603.03329.

Xiaotian Luo, Dizhan Xue, Fengxingyu Wang, Chuanrui Hu, and Yafeng Deng. HarnessBank: Semantic gene-bank search with gated verification for agent-harness self-evolution, 2026. URL https://arxiv.org/abs/2607.13683.

Ruiying Ma, Shreya Shankar, Ruiqi Chen, Yiming Lin, Sepanta Zeighami, Rajoshi Ghosh, Abhinav Gupta, Anushrut Gupta, Tanmai Gopal, and Aditya G. Parameswaran. Can AI agents answer your data questions? a benchmark for data agents. *arXiv preprint arXiv:2603.20576*, 2026. URL https://arxiv.org/abs/2603.20576.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

OpenAI. GPT-5.5 system card. Technical report, OpenAI, April 2026a. URL https://deploymentsafety.openai.com/gpt-5-5/gpt-5-5.pdf.

OpenAI. GPT-5.6 Terra Model. Official API documentation, 2026b. URL https://developers.openai.com/api/docs/models/gpt-5.6-terra. Accessed September 20, 2026.

Zhiming Pan, Junyu Luo, Zhiping Xiao, Kaize Ding, Xiao Luo, and Ming Zhang. ZipAct: Zipping interaction history into a compact state for efficient LLM agents. *Transactions on Machine Learning Research*, May 2026. ISSN 2835-8856. URL https://openreview.net/forum?id=ZssIalqqrz.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, H. Vicky Zhao, Lili Qiu, and Dongmei Zhang. LLMLingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 963–981, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.57. URL https://aclanthology.org/2024.findings-acl.57/.

Jincheng Ren, Siwei Wu, Yizhi Li, Kang Zhu, Shu Xu, Boyu Feng, Ruibin Yuan, Wei Zhang, Riza Batista-Navarro, Jian Yang, and Chenghua Lin. A self-evolving framework for efficient terminal agents via observational context compression, 2026. URL https://arxiv.org/abs/2604.19572.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 68539–68551. Curran Associates, Inc., 2023. doi: 10.52202/075280-2997. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/d842425e4bf79ba039352da0f658a906-Paper-Conference.pdf.

Shuai Shao, Kangning Zhang, Qingyao Li, Shijian Wang, Hao Wang, Wenxiang Jiao, Yuan Lu, Yi Guo, Weiwen Liu, and Weinan Zhang. Harness-R1: Learning to edit executable runtime harnesses from agent failure trajectories, 2026. URL https://arxiv.org/abs/2608.02276.

Stack Overflow. Agents on a leash: Agentic ai remains mostly single-agent and monitored at work, May 2026. URL https://stackoverflow.blog/2026/05/27/agents-on-a-leash-agentic-ai-remains-mostly-monitored-at-work/. Accessed September 26, 2026.

SWE-agent Team. mini-swe-agent. GitHub repository, n.d. URL https://github.com/SWE-agent/mini-swe-agent. Accessed September 22, 2026.

Terminal-Bench Team. Terminal-bench 2.1. Official release note, May 2026. URL https://www.tbench.ai/news/terminal-bench-2-1.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=ehfRiF0R3a.

Ningning Wang, Xavier Hu, Pai Liu, He Zhu, Yue Hou, Heyuan Huang, Shengyu Zhang, Jian Yang, Jiaheng Liu, Ge Zhang, Changwang Zhang, Jun Wang, Yuchen Eleanor Jiang, and Wangchunshu Zhou. Efficient agents: Building effective agents while reducing cost, 2025. URL https://arxiv.org/abs/2508.02694.

Yuhang Wang, Yuling Shi, Mo Yang, Rongrui Zhang, Shilin He, Heng Lian, Yuting Chen, Siyu Ye, Kai Cai, and Xiaodong Gu. SWE-Pruner: Self-adaptive context pruning for coding agents, 2026. URL https://arxiv.org/abs/2601.16746.

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. doi: 10.1145/3797084. URL https://doi.org/10.1145/3797084.

Tianshi Xu, Huifeng Wen, and Meng Li. Adapting the interface, not the model: Runtime harness adaptation for deterministic LLM agents, 2026. URL https://arxiv.org/abs/2605.22166.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-Harness: Harnesses that improve themselves, 2026. URL https://arxiv.org/abs/2606.09498.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, XiongHui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/5492ecbce4439401798dcd2c90be94cd-Paper-Conference.pdf.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph Gonzalez, and Ion Stoica. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46595–46623, 2023. doi: 10.52202/075280-2020. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/724be4472168f31ba1c9ac630f15dec8-Paper-Conference.pdf.

AI USE STATEMENT

In this work, we used generative AI tools only for final-stage language polishing and grammar refinement of the manuscript. We have not used generative AI tools for generating research ideas, designing methods, conducting experiments, implementing code, analyzing data, interpreting results, or producing scientific claims, and other generative AI-assisted tasks requiring disclosure are not applicable to this work. We have reviewed all AI-assisted revisions to ensure their correctness and consistency with the authors' intended content. We take responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We have made efforts to ensure the reproducibility of this work by providing detailed descriptions of the proposed method, experimental settings, implementation details, and evaluation protocols in the main paper and supplementary materials. The supplementary materials include the source code required to reproduce our experiments, and we will publicly release the code after the review process to further facilitate verification and future research. The content of this paper can be completely reproduced.

A DETAILED FORMULATION AND EVOLUTION RULES

This appendix specifies the trajectory representation, pattern detection rules, and harness acceptance criteria.

A.1 PRELIMINARIES

A.1.1 EXECUTION TRAJECTORY AND INFERENCE COST

For task x, let a_t denote the model response and o_t the environment observation at round t. A trajectory of T rounds is

$$\tau(x) = \big((a_1, o_1), \ldots, (a_T, o_T)\big), \tag{6}$$

with interaction history before round t given by

$$h_t = \big((a_1, o_1), \ldots, (a_{t-1}, o_{t-1})\big). \tag{7}$$

Let n_t^{u}, n_t^{c}, and n_t^{o} denote uncached-input, cached-input, and output tokens across all primary and auxiliary model calls in round t. With corresponding per-token prices p_{u}, p_{c}, and p_{o}, inference cost is

$$c_t = p_{\mathrm{u}} n_t^{\mathrm{u}} + p_{\mathrm{c}} n_t^{\mathrm{c}} + p_{\mathrm{o}} n_t^{\mathrm{o}}, \qquad \mathcal{C}(\tau) = \sum_{t=1}^{T} c_t. \tag{8}$$

Under exact-prefix caching, rewriting earlier content may incur uncached-input cost again (Zheng et al., 2024).

The benchmark verifier returns task score $\mathcal{U}(\tau, x)$, which may be binary or partial.

A.1.2 EXECUTION HARNESS

For a fixed model f_θ, the harness consists of behavioral instructions I, reusable skills S, tool interfaces Ω, and executor E,

$$\mathcal{H} = (I, S, \Omega, E). \tag{9}$$

Let z_t denote the environment and executor state, $\Phi_{\mathcal{H}}$ the input constructor, $\Psi_{\mathcal{H}}$ the action executor, and d_t the termination indicator. Each round follows

$$\begin{aligned} a_t &\sim f_\theta\left(\cdot \mid \Phi_{\mathcal{H}}(x, h_t)\right), \\ (o_t, z_{t+1}, d_t) &= \Psi_{\mathcal{H}}(z_t, a_t). \end{aligned} \tag{10}$$

We denote the trajectory produced by model f_θ under harness $\mathcal{H}$ as $\tau(x; \mathcal{H})$.

In EvoCost, the model parameters remain fixed, the harness remains unchanged within each evaluated rollout, and recorded interactions are append-only.

A.1.3 Problem Formulation

Given deployment task distribution $\mathcal{D}$ and initial harness $\mathcal{H}^{(0)}$, we minimize expected inference cost subject to a minimum expected task score $U_{\min}$,

$$\begin{aligned} \min_{\mathcal{H}} \quad & \mathbb{E}_{x\sim\mathcal{D}}\left[\mathcal{C}\big(\tau(x;\mathcal{H})\big)\right], \\ \text{subject to} \quad & \mathbb{E}_{x\sim\mathcal{D}}\left[\mathcal{U}\big(\tau(x;\mathcal{H}), x\big)\right] \geq U_{\min}. \end{aligned} \tag{11}$$

Only (I, S, Ω, E) are optimized. The finite-sample acceptance tests in Section A.4.3 guide this search but do not guarantee the population constraint $U_{\min}$.

A.2 An Overview of EvoCost

Each cycle in Figure 1 converts trajectories from the current harness into evidence for candidate edits. Section A.3 specifies cost-saving opportunity discovery, and Section A.4 specifies validated harness evolution through patch generation, merging, and acceptance. Discovery and iteration-level validation use the same reserved development tasks. Accepted harnesses supply the next cycle's trajectories.

A.3 Cost-Saving Opportunity Discovery

This stage identifies cost-saving opportunities as groups of observed interactions that may be skipped, combined, or replaced by an earlier change of approach. A trajectory records temporal order, but that order alone does not establish which earlier results support a later decision. EvoCost reconstructs these dependencies and uses operation annotations to distinguish potentially unnecessary interactions, serial workflows, independent operations, and failed approaches. Figure 2 illustrates how dependency and outcome evidence supports these candidate interventions. The output is a collection of pattern instances with source evidence for the optimizer. Whether an opportunity produces actual savings is assessed through rollout validation in Section A.4.

A.3.1 Support DAG Construction

We construct one graph node for each interaction, retaining its action, observation, inference cost, and a local annotation of the operation. Let c_t be its round cost from Equation 8 and let α_t collect four annotation fields. The operation type ρ_t distinguishes reading, writing, execution, and verification. The status y_t records whether the operation succeeds, and the resource set $\mathcal{R}_t$ lists the files or database objects it touches. The outcome signature ϕ_t briefly describes the returned result so that related interactions can be compared. We represent node v_t as

$$v_t = (a_t, o_t, c_t, \alpha_t), \qquad \alpha_t = (\rho_t, y_t, \mathcal{R}_t, \phi_t). \tag{12}$$

A language-model annotator receives the task and the model-visible trajectory prefix ending at v_t. It generates the local annotation α_t and a direct dependency set D_t satisfying

$$D_t \subseteq \{v_1, \ldots, v_{t-1}\}. \tag{13}$$

A node v_s is included in D_t only if information or an artifact produced by v_s is directly used by interaction v_t. The annotator is not given hidden validation code or reference answers.

Let $\mathcal{V}_\tau$ be the interaction-node set and $\mathcal{E}_\tau$ the directed edges induced by the dependency sets. The support DAG G_τ is defined as

$$\begin{aligned} \mathcal{V}_\tau &= \{v_1, \ldots, v_T\}, \\ \mathcal{E}_\tau &= \{(v_s, v_t) : v_s \in D_t\}, \\ G_\tau &= (\mathcal{V}_\tau, \mathcal{E}_\tau). \end{aligned} \tag{14}$$

Since each dependency points from an earlier interaction to a later one, G_τ is acyclic. These edges describe dependencies in the observed rollout, so the absence of an edge does not establish that an interaction can be omitted on every execution path.

A.3.2 Cost Pattern Detection

Let $\mathcal{V}_{\text{term}}$ contain interactions that submit the final answer or perform terminal validation visible to the agent. The dependency closure $\mathcal{K}(\tau)$ contains these terminal nodes and their ancestors, where Anc_{G_τ} returns all ancestors in the support DAG:

$$\mathcal{K}(\tau) = \mathcal{V}_{\text{term}} \cup \text{Anc}_{G_\tau}(\mathcal{V}_{\text{term}}). \tag{15}$$

The closure identifies interactions that support the terminal result through the reconstructed dependencies. The patterns are organized by the evidence supporting an intervention, not by a single classification rule. Dependency structure primarily supports skipping, collapsing, and batching through closure membership and serial or independent relationships. Operation annotations and resource checks further constrain these candidates. Failure and recovery outcomes primarily support earlier pivoting, while the DAG retains the context needed to interpret them. The four patterns are neither exhaustive nor mutually exclusive. They describe actionable candidates under the available evidence, not all possible ways to reduce inference cost. The DAG provides the closure for Skippable, the internal data flow for Collapsible, and the predecessor relationships for Mergeable. Temporal order alone cannot establish these conditions.

Skippable Interactions. A skippable candidate is a maximal temporally contiguous segment of at least two interactions outside $\mathcal{K}(\tau)$. Requiring two interactions excludes isolated omissions from the closure and focuses the evidence on sequences of potentially unnecessary interactions. This heuristic does not eliminate dependency-annotation errors. Such a segment is a candidate for skipping in the observed trajectory because none of its nodes supports the terminal result through the reconstructed dependencies. The patch generator uses this pattern to infer a reusable condition under which similar interactions can be skipped in future tasks. The resulting harness edit is evaluated before deployment.

Collapsible Interactions. A collapsible candidate is a successful serial path inside $\mathcal{K}(\tau)$ whose intermediate outputs are consumed only by later nodes on the same path. If the final environment state and downstream information can be preserved by one bounded composite tool interaction, a path of $m \geq 2$ model rounds can potentially be reduced to one round. This structural condition guides composite-tool proposals from the supplied trajectory evidence. The generated patch must specify the operation, its output bounds, and its failure behavior.

Mergeable Interactions. A mergeable candidate contains two to four successful tool interactions inside $\mathcal{K}(\tau)$ with identical direct dependency sets and no recorded dependency between group members. Their operation types and structured outputs must be compatible. For write operations, the touched resource sets must be non-empty and pairwise disjoint. The upper bound limits the number of operations represented by one example, while the proposed tool must separately bound its returned observations. These graph and resource checks suggest that batching is possible, but do not prove independence under unobserved execution conditions. For a group of interactions B, replacing the group with one bounded batched interaction can save up to $|B| - 1$ model rounds.

Failure-to-Pivot Interactions. A failure-to-pivot candidate begins with at least two consecutive failed interactions immediately followed by a successful interaction. The operation and resource annotations help the generator assess whether the recovery suggests an earlier change of approach. The evidence includes the segment's predecessors in the support DAG to retain the context of the failure. Failed nodes need not lie outside $\mathcal{K}(\tau)$, since observing a failure can directly support the successful next decision. Likewise, recovery need not share all predecessors with the failed attempt. The patch generator uses the failure and recovery evidence to propose an instruction, skill, or executor check that recognizes a similar stalled state earlier. The resulting behavior depends on the observed failure condition, without requiring a fixed retry limit for every task.

Pattern instances may overlap. For example, a failed retry segment outside the closure can support both skipping and an earlier pivot. We retain both interpretations with their source interactions rather than imposing a mutually exclusive label. Overlapping edits are reconciled during merging, and

their proposed savings are not added together. Only the cost of the combined harness is used for acceptance.

We group the trajectory evidence across evolution tasks before passing it to harness optimization. By presenting recurring behavior together, the batches give patch generators evidence beyond a single task-specific event. The observed round cost c_t describes the expense of the recorded execution, not the cost that a replacement would incur. Fewer model rounds need not yield lower inference cost if a composite tool returns more tokens or adds a large permanent schema. We therefore use structural opportunities to propose edits and measured inference cost to decide acceptance, without assuming a calibrated monetary saving for each pattern instance.

A.4 Validated Harness Evolution

This stage converts discovered cost-saving opportunities into reusable harness changes and tests whether they meet the optimization objective. The detected patterns specify behavior to revise, but an edit may need to span several harness components. Processing patterns one at a time and immediately applying each edit makes later proposals depend on the preceding changes. Repeated additions to the instructions can also accumulate redundant rules. EvoCost generates proposals independently and reconciles them through merging, as shown in Figure 3. Hybrid Gating screens patches before merging and evaluates the combined candidate through paired rollouts. The optimizer returns either an accepted harness for the next cycle or the unchanged current harness.

A.4.1 Parallel Patch Generation

At evolution cycle k, the pattern collection is divided into M_k bounded batches, with $\mathcal{P}_j^{(k)}$ denoting batch j. The patch generator A_{evo} receives the current harness snapshot $\mathcal{H}^{(k)}$ and one batch, then produces a diff $\Delta_j^{(k)}$:

$$\Delta_j^{(k)} = A_{\text{evo}}\left(\mathcal{H}^{(k)}, \mathcal{P}_j^{(k)}\right), \qquad j = 1, \ldots, M_k. \tag{16}$$

The output $\Delta_j^{(k)}$ is a diff over one or more components in (I, S, Ω, E). Each diff remains linked to the pattern batch that motivated it. This link is later used by the patch-level gate to determine whether the proposed edit is supported by trajectory evidence. No generator modifies the shared harness snapshot.

A.4.2 Patch Merging

Let $g_{\text{patch},j}^{(k)} \in \{0, 1\}$ denote the patch-level decision defined in Section A.4.3. We collect the retained proposal indices in $\mathcal{J}^{(k)}$ as follows:

$$\mathcal{J}^{(k)} = \left\{ j : g_{\text{patch},j}^{(k)} = 1 \right\}. \tag{17}$$

If $\mathcal{J}^{(k)}$ is empty, the cycle terminates without a candidate harness. Otherwise, the merger A_{merge} receives the current harness, retained diffs, and source evidence. It produces a merged diff $\overline{\Delta}^{(k)}$, which the application operator $\oplus$ installs to form the candidate harness $\widetilde{\mathcal{H}}^{(k+1)}$:

$$\begin{aligned} \overline{\Delta}^{(k)} &= A_{\text{merge}}\left(\mathcal{H}^{(k)}, \left\{ (\Delta_j^{(k)}, \mathcal{P}_j^{(k)}) \right\}_{j \in \mathcal{J}^{(k)}}\right), \\ \widetilde{\mathcal{H}}^{(k+1)} &= \mathcal{H}^{(k)} \oplus \overline{\Delta}^{(k)}. \end{aligned} \tag{18}$$

The merger removes duplicated or superseded behavior and resolves overlapping source edits. This consolidation also controls recurring prompt cost. Instructions, skill content, and tool schemas consume input tokens whenever they are included in a model call, even on tasks that do not benefit from the new capability. The generator is instructed to account for this overhead, and the merger keeps the combined harness concise rather than concatenating proposals. This is a qualitative cost regularization, not an additional numerical penalty in the objective. The resulting prompt overhead is included in the token-based inference cost of Equation 8.

The merger checks that instructions, skills, tool schemas, and executor code specify compatible behavior. If the retained changes cannot all be reconciled, the merger can keep a compatible subset or return an empty diff. Generating proposals from a common snapshot prevents them from inheriting earlier edits within the cycle. The merger checks their compatibility, and rollout validation evaluates the combined behavior.

A.4.3 HYBRID GATING

Hybrid Gating combines semantic screening without environment rollouts and empirical validation through paired execution. The patch-level gate uses deterministic checks and a language-model judge to screen proposals before merging. This screening allocates rollout budget to the candidates that survive the first check. It still incurs judge-call cost, but avoids paired benchmark execution for each individual proposal. The iteration-level gate then spends that execution budget on the combined harness, whose effects cannot be established by semantic judgment alone. In the gating ablation, the first check rejects 17 of 160 proposals before merging (Section 4.3).

Patch-Level Gate For a candidate diff Δ, let $d(\Delta) \in \{0, 1\}$ indicate whether the diff applies cleanly and whether all modified configuration files, schemas, and source files can be parsed or compiled. Proposals that fail this deterministic check are rejected directly.

A language-model judge evaluates each valid patch against its source pattern batch $\mathcal{P}$. It checks whether the proposed behavior can be reused across tasks and could remove or combine the identified interactions. The cost assessment weighs the proposed reduction in model decisions and redundant output against permanent instruction and tool-schema overhead. This is an evidence-based judgment of likely savings, not a monetary estimate or a substitute for rollout validation. The judge also assesses preservation of task behavior, agreement among modified components, and whether the evidence supports the scope of the edit.

Let $R_{\max}$ be the number of gating stages and η_r the minimum semantic score at stage r. We tighten screening as evolution advances, with $\eta_1 \le \eta_2 \le \cdots \le \eta_{R_{\max}}$. The active stage r_k depends on the number of accepted updates before cycle k, as defined in Equation 23. Using $\mathbf{1}[\cdot]$ for the indicator function, the patch decision is

$$g^{(k)}_{\text{patch},j} = \mathbf{1}\left[d(\Delta^{(k)}_j) = 1 \ \wedge \ s_{\text{judge}}(\Delta^{(k)}_j, \mathcal{P}^{(k)}_j) \ge \eta_{r_k}\right]. \tag{19}$$

The retained patches proceed to merging and paired rollout validation.

Iteration-Level Gate The iteration-level gate compares the current and candidate harnesses through paired execution. Let $\mathcal{H}^{(k)}_0$ denote the current harness and $\mathcal{H}^{(k)}_1$ the candidate:

$$\mathcal{H}^{(k)}_0 = \mathcal{H}^{(k)}, \qquad \mathcal{H}^{(k)}_1 = \widetilde{\mathcal{H}}^{(k+1)}. \tag{20}$$

The two harnesses are executed on the same validation set $\mathcal{V}^{(k)}$ using identical model configurations, environments, and resource limits.

Every validation task remains in the task-score denominator. An execution error or missing run receives a score of zero. For the cost comparison, let $\mathcal{O}^{(k)} \subseteq \mathcal{V}^{(k)}$ contain tasks that finish without infrastructure or harness execution errors under both harnesses. For either harness $\mathcal{H}'$, let $\widehat{\mathcal{U}}_k$ be its mean score over the full validation set and $\widehat{\mathcal{C}}_k$ its mean cost over the common error-free set:

$$\begin{aligned} \widehat{\mathcal{U}}_k(\mathcal{H}') &= \frac{1}{|\mathcal{V}^{(k)}|} \sum_{x \in \mathcal{V}^{(k)}} \mathcal{U}\big(\tau(x; \mathcal{H}'), x\big), \\ \widehat{\mathcal{C}}_k(\mathcal{H}') &= \frac{1}{|\mathcal{O}^{(k)}|} \sum_{x \in \mathcal{O}^{(k)}} \mathcal{C}\big(\tau(x; \mathcal{H}')\big). \end{aligned} \tag{21}$$

The first definition prevents failed runs from being removed from task-performance evaluation. The second excludes apparent savings caused by infrastructure or harness execution errors.

At stage r, let $p_r \in (0, 1]$ denote the minimum fraction of the current harness's task score that the candidate must retain, and let $\gamma_r < 0$ denote the largest allowed relative cost change. A more negative γ_r requires a larger cost reduction. We use increasingly strict stages satisfying

$$p_1 \le p_2 \le \cdots \le p_{R_{\max}}, \qquad \gamma_1 \ge \gamma_2 \ge \cdots \ge \gamma_{R_{\max}}. \tag{22}$$

Only an accepted harness update advances the stage. Let $g_{\text{iter}}^{(\ell)}$ indicate whether cycle ℓ was accepted. The active stage at cycle k is

$$r_k = \min\left\{1 + \sum_{\ell<k} g_{\text{iter}}^{(\ell)}, R_{\max}\right\}. \tag{23}$$

A rejected cycle, including one that ends before rollout validation, has $g_{\text{iter}}^{(\ell)} = 0$ and does not advance the stage.

The merged candidate is accepted only when a valid paired cost comparison is available, the required task performance is retained, and the required cost reduction is achieved:

$$\begin{aligned} g_{\text{iter}}^{(k)} = \mathbf{1}\Bigg[& |\mathcal{O}^{(k)}| > 0 \\ & \wedge \widehat{\mathcal{U}}_k(\mathcal{H}_1^{(k)}) \geq p_{r_k} \widehat{\mathcal{U}}_k(\mathcal{H}_0^{(k)}) \\ & \wedge \frac{\widehat{\mathcal{C}}_k(\mathcal{H}_1^{(k)}) - \widehat{\mathcal{C}}_k(\mathcal{H}_0^{(k)})}{\widehat{\mathcal{C}}_k(\mathcal{H}_0^{(k)})} \leq \gamma_{r_k} \Bigg]. \end{aligned} \tag{24}$$

If $g_{\text{iter}}^{(k)} = 1$, the candidate becomes the current harness for the next cycle. Otherwise, the previous harness is retained. Let $K_{\max}$ be the maximum number of evolution cycles. Evolution ends when an update is accepted at the final stage or the cycle budget $K_{\max}$ is exhausted.

The relative score floor is a local search criterion for the deployment objective in Equation 11. Early stages allow limited score loss to explore cost-saving edits, while later stages tighten retention. Because the reference is the current harness, successive tolerated losses can accumulate. Passing these finite-sample tests does not certify that expected deployment performance exceeds $U_{\min}$. We assess the final cost and performance trade-off on held-out tasks in Section 4.

B Implementation and Experimental Protocols

B.1 Baseline Implementations

Common execution interface. Original uses the unmodified `mini-swe-agent` task-solving loop (SWE-agent Team, n.d.). The runtime baselines retain its task instructions, native tool interface, and benchmark verifier. Their additional controls operate during task execution, whereas GEPA and Meta-Harness revise a harness before held-out evaluation. The descriptions below distinguish the baseline mechanisms from their Harbor adaptations. Primary and auxiliary model calls during task execution contribute to the reported inference cost, while offline experience extraction and harness search are excluded.

AgentDiet. AgentDiet compresses selected trajectory steps using their surrounding context (Xiao et al., 2026). The compressor receives an XML representation and a target step identifier. It removes irrelevant or duplicated text while preserving useful details and structural markers. The Harbor adapter rejects empty or length-truncated compression outputs and retains the usage records of both compressor calls and replaced messages.

ZipAct. ZipAct separates state initialization, action generation, and state updating (Pan et al., 2026). The actor receives the immutable task, a compact Goal-World-Constraint state, and the latest observation instead of the full interaction history. The updater retains durable facts, active objectives, and negative constraints. The Harbor adaptation bounds the state representation and adds action-budget-dependent guidance for editing, verification, and submission.

Compact state introduces an information-retention trade-off in coding tasks. ZipAct updates the previous state using the latest action and observation, so earlier evidence remains available to the actor only if it is retained in the state or retrieved again. The updater must therefore select information before its relevance to later debugging is fully known. For an illustrative example, a test may report that a parser returns an empty list where `None` is expected. A state that retains only that the parser

test fails loses the distinction between an absent value and an explicitly empty collection. If a later edit depends on that distinction, the actor must inspect the test or rerun it to recover the evidence, or risk applying an incorrect fix. Such repeated inspection can offset savings from shorter model inputs. This causes the poor performance as illustrated in Table 2.

EET. EET augments the task prompt with retrieved experience and requests confidence assessments at execution checkpoints (Guo et al., 2026). The adapter uses the upstream experience store and TF-IDF retrieval. A confidence response supplies a score from 0 to 100. The subsequent completion prompt requests submission only if the artifact is complete and verified, and otherwise permits continued execution. The standard task-solving and submission interface remains unchanged.

Self-evolution search space and objective. We preserve the internal search mechanisms of each evolution method rather than imposing a shared acceptance rule. GEPA retains Pareto-based parent selection, Meta-Harness retains proposer-driven archive search, and EvoCost uses its proposed Hybrid Gating. These mechanisms constitute part of the methods being compared. Fairness is controlled externally: all methods start from the same harness, optimize the same editable artifacts, use the same development tasks and model configuration, and operate under identical cycle, proposal, and rollout budget limits. Importantly, the lower inference cost of EvoCost is not obtained by systematically accepting lower-performing harnesses. With GPT-5.5, EvoCost achieves a higher SWE-bench score than both evolution baselines while using lower inference cost, and on DeepSWE it matches Meta-Harness in score while reducing cost.

GEPA. We use the upstream optimizer's reflective mutation, instance-level Pareto parent selection, and minibatch comparison (Agrawal et al., 2026). The three harness files are serialized as one JSON component so that a mutation can coordinate their interfaces. Reflection receives four development cases with reward, cost, and ordered execution feedback. Trace fields are bounded for the reflection context, while complete traces remain archived. The development tasks serve both search-internal training and validation, without exposing held-out tasks.

Meta-Harness. The proposer queries a read-only archive of candidate code, scores, and raw traces through filesystem tools (Lee et al., 2026). It chooses which archived candidate to build on, writes one complete candidate, checks its interface, and submits a parent identifier and hypothesis. The outer controller evaluates and archives the candidate. This adaptation restricts the original code-search interface to the three harness artifacts. Neither self-evolution baseline receives EvoCost's dependency patterns, parallel patch merger, or Hybrid Gating.

Search budgets. Within each benchmark and backbone setting, GEPA, Meta Harness, and EvoCost use the same initial harness, the same 16 reserved tasks, the same editable artifacts, and the same limits on cycles, proposals, and rollout evaluations. These limits control the external search resources. Each method retains its own candidate generation, parent selection, and acceptance procedure because these are part of its optimization algorithm. Actual search expenditure may therefore differ across methods.

B.2 Prompt Structures

Figures 7 and 8 summarize the role, input, constraints, and output of each prompt interface. The Markdown templates in `prompt/` follow the same structure. These are abbreviated descriptions of the implemented prompts, not verbatim execution strings.

B.3 API Pricing and Accounting

Table 4 records the accounting configuration inspected on August 5, 2026. Rates are expressed in dollars per million tokens and retain the manuscript's existing currency normalization.

If reported input usage includes cache hits, we first compute

$$n_{\text{uncached}} = n_{\text{input}} - n_{\text{cached}}, \qquad c = \frac{p_u n_{\text{uncached}} + p_c n_{\text{cached}} + p_o n_{\text{output}}}{10^6}. \tag{25}$$

Here p_u, p_c, and p_o are the three rates in Table 4. Cached tokens are not charged again as uncached input.

Dependency annotation
Input: task, trajectory prefix, current step.
Instruction: identify supporting earlier steps and classify the operation.
Output: dependency indices and operation type as JSON.

Parallel Patch Generation
Input: shared harness, pattern evidence, rejection memory.
Instruction: make one reusable cost-saving change across allowed artifacts.
Output: coherent patch or no change.

Patch screening
Input: staged diff, harness, source evidence.
Instruction: assess reusability, savings, correctness, consistency, and evidence.
Output: criterion scores and blocking issues.

Patch merging
Input: current harness, retained patches and evidence.
Instruction: reconcile overlaps and check interfaces without adding unrelated edits.
Output: merged artifacts or unchanged harness.

Installed-harness validation
Input: installed candidate, diff, disposable codebase.
Instruction: execute bounded checks without modifying the harness.
Output: passed and failed checks as JSON.

Shared evolution constraints
Editable: tool schemas, executor, instructions.
Preserve: native bash, bounded outputs, visible errors, and interface consistency.
Exclude: hidden answers, held-out evidence, and hard-coded task solutions.

Figure 7: Structural prompt templates for EvoCost. The shared constraints apply to harness evolution. Paired score and cost acceptance is computed separately by the controller.

Original
Input: task and interaction history.
Instruction: follow the standard task, tool, and submission rules.
Output: native action or submission.

AgentDiet
Input: XML context and target step.
Instruction: remove irrelevant or duplicated content while retaining useful details.
Output: compressed target step only.

ZipAct
Initialize: task → compact state.
Act: state and latest observation → action.
Update: prior state, action, observation → goals, facts, constraints.

EET
Retrieve: task → relevant experience.
Assess: execution evidence → confidence score.
Complete: submit if complete and verified, otherwise continue.

GEPA
Input: harness JSON and development feedback.
Instruction: reflect and revise under the shared interface and cost objective.
Output: complete three-artifact JSON.

Meta-Harness
Input: searchable development archive.
Instruction: inspect history, choose a parent, write and validate a candidate.
Output: candidate files and hypothesis.

Figure 8: Structural prompt templates for Original and the five adapted baselines. Runtime methods change execution context or stopping behavior. GEPA and Meta-Harness propose harness revisions before evaluation.

Table 4: Configured accounting rates ($/million tokens).

Model / accounting basis	Uncached input	Cached input	Output
DeepSeek-V4-Flash	0.14	0.0028	0.28
GPT-5.6-Terra	2.00	0.20	12.00
GPT-5.5	5.00	0.50	30.00

C Cost and Performance Analysis

C.1 Inference Cost and Performance Trade-offs

Figure 9 plots the cost and score of all seven methods in Table 2 for each benchmark and backbone. Positions toward the upper left indicate lower cost and higher performance.

EvoCost reduces cost relative to Original in all settings, but the SWE-bench result with GPT-5.6-Terra and the DAB result with DeepSeek-V4-Flash trade lower cost for lower scores. ZipAct on DeepSWE with GPT-5.5 likewise illustrates that a cheaper harness can have substantially lower performance. These trade-offs depend on the performance loss acceptable for deployment. The arrows summarize aggregate differences under the table's evaluation and billing conventions.

C.2 Evolution Cost and Break-even Analysis

For a harness-evolution method m, let E_m denote the total API expenditure from the initial development rollouts through final harness selection. For EvoCost, this includes dependency annotation, patch generation, semantic judging, merging, installed-harness checks, and development-set validation. Rejected proposals and unsuccessful validation attempts remain included. For GEPA and Meta-Harness, the corresponding total includes reflection or proposer calls and all candidate evaluations. Each call is counted once, and final held-out evaluation is excluded from E_m.

Search expenditure. The methods share the same search budget limits, while their actual expenditure may differ because their search procedures differ. Table 5 reports total evolution API expenditure with GPT-5.6-Terra across four benchmarks. EvoCost incurs the lowest search cost on DeepSWE, but not on the other three benchmarks. Compared with Meta-Harness, its search cost is lower on three benchmarks and higher on SWE-bench. These results show that the inference cost savings do not require uniformly lower search expenditure. Their economic benefit instead depends on how many subsequent task queries share the evolved harness.

Table 5: Evolution cost and recovery workload with GPT-5.6-Terra.

Method	SWE-bench		DeepSWE		DAB		TB-2.1	
	Cost	# Recover	Cost	# Recover	Cost	# Recover	Cost	# Recover
GEPA	20.8481	152	99.4698	371	11.3786	328	25.2248	∞
Meta-Harness	22.2606	137	106.2578	3,496	23.1459	1,666	55.1580	∞
EvoCost	34.8154	228	79.9158	192	16.6607	305	27.0458	411

Cost denotes total evolution API expenditure in $. # Recover denotes the number of task queries required to offset this expenditure relative to Original, computed from the mean inference costs in Table 2 using Equation 27. ∞ denotes no finite recovery workload because inference cost does not decrease.

Amortization criterion. Let E be the complete search API cost and let C_0 and C_1 be mean inference costs measured on the same evaluation tasks before and after evolution. For a deployment workload of N comparable tasks, the API-cost difference is

$$\Delta C_{\text{total}}(N) = E + N(C_1 - C_0). \tag{26}$$

If $C_1 < C_0$, the smallest workload with non-positive incremental API cost is

$$N_{\text{break-even}} = \left\lceil \frac{E}{C_0 - C_1} \right\rceil. \tag{27}$$

There is no finite break-even workload when inference cost does not decrease and search incurs positive cost. This calculation excludes hardware and labor costs and assumes the measured task distribution remains representative. It must be accompanied by the score difference, since a cheaper but less capable harness need not be preferable. Using the GPT-5.6-Terra inference costs in Table 2, EvoCost saves $0.1529 per task on SWE-bench, $0.4168 on DeepSWE, $0.0548 on DAB, and $0.0659 on TB-2.1. Equation 27 therefore gives recovery workloads of 228, 192, 305, and 411 task queries, respectively. Each query denotes a complete benchmark task request rather than an individual

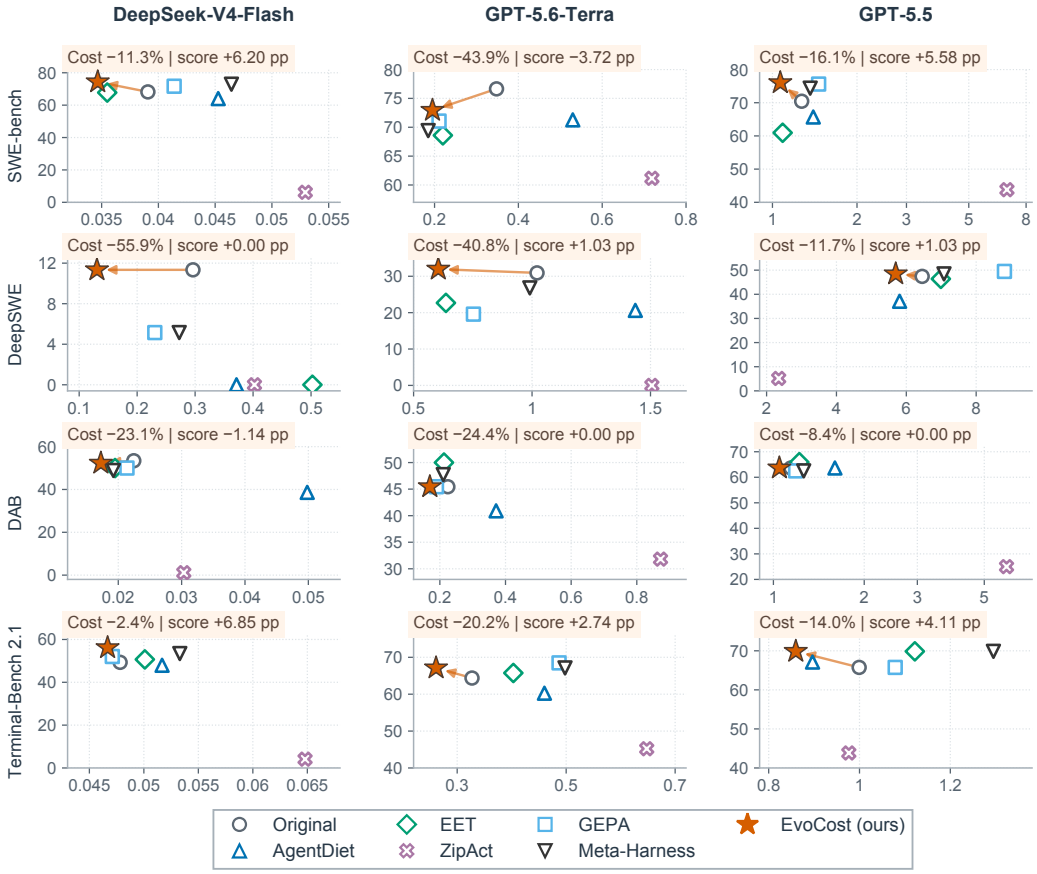


Figure 9: Inference cost and performance trade-offs across the four main benchmarks (rows) and three backbones (columns). Horizontal axes show inference cost ($/task), and vertical axes show score (%). Cost axes are logarithmic for GPT-5.5 on SWE-bench and DAB. Arrows connect Original to EvoCost, with annotations showing relative cost changes and score differences in percentage points.

model call. These estimates use the reported mean inference costs and assume comparable future tasks. Cost recovery does not imply unchanged task performance. Relative to Original, the reported EvoCost score decreases by 3.72 percentage points on SWE-bench and is preserved or improved on the other three benchmarks. GEPA and Meta-Harness recover their evolution expenditure on SWE-bench, DeepSWE, and DAB, but not on TB-2.1, where their inference costs exceed Original. EvoCost reaches recovery sooner than either baseline on DeepSWE and DAB, whereas both baselines recover sooner on SWE-bench.

Comparing evolution methods. For two evolved harnesses a and b, total API expenditure at workload N is $E_a + NC_a$ and $E_b + NC_b$. If $C_a < C_b$, the smallest nonnegative workload at which a is no more expensive is

$$N_{a \preceq b} = \max\left(0, \left\lceil \frac{E_a - E_b}{C_b - C_a} \right\rceil\right). \tag{28}$$

Tables 5 and 2 provide the search and inference costs for this comparison with GPT-5.6-Terra. EvoCost has lower inference cost than both evolution baselines on DeepSWE, DAB, and TB-2.1, so any initial search cost disadvantage decreases as deployment volume grows. On SWE-bench, Meta-Harness incurs lower search and inference costs than EvoCost, whereas EvoCost achieves a higher task score.

C.3 Inference Cost Decomposition

Figure 10 compares the Original and EvoCost harnesses with GPT-5.6-Terra, decomposing mean inference cost into uncached-input, cached-input, and output charges. Token-charge shares and model-call counts come from the recorded Terra runs. The accompanying model-call counts distinguish changes in interaction frequency from changes in token expenditure.

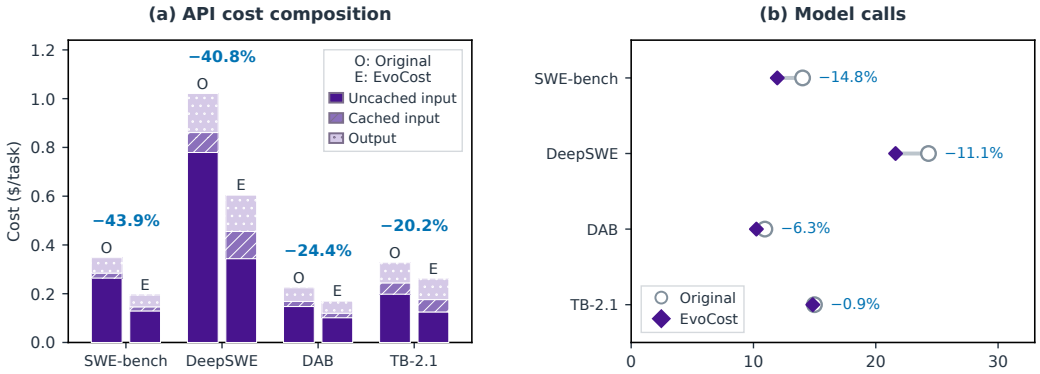


Figure 10: Inference cost decomposition from recorded GPT-5.6-Terra runs. (a) Mean token charges per task. (b) Mean model calls per task. Blue annotations show relative changes from Original to EvoCost.

The cost reduction exceeds the reduction in call count on every benchmark. On SWE-bench, mean cost falls from \$0.3479 to \$0.1950 (43.9%), while calls decrease by 14.8%. The corresponding reductions are 40.8% versus 11.1% on DeepSWE, 24.4% versus 6.3% on DAB, and 20.2% versus 0.9% on TB-2.1. Thus, fewer calls alone do not account for the savings: average expenditure per call also decreases. Uncached-input charges contribute the largest absolute reduction on all four benchmarks. On TB-2.1, these savings offset increases in cached-input and output charges despite an almost unchanged call count. This decomposition is consistent with cost-effective harness self-evolution changing the token expenditure of execution, rather than merely shortening it.

D Mechanism Analysis and Case Studies

D.1 Harness Modification Distribution

Figure 11 summarizes 1,346 nonempty proposals and 1,170 proposals that pass the local patch gate. Instructions account for the largest share at both stages, while tool schemas and executors together account for the majority of modification events. This distribution shows that the search extends beyond instruction editing. These frequencies describe the scope of proposed and screened edits rather than the independent cost contribution of each component.

The similar distributions are consistent with screening proposal quality rather than component type. The gate assesses validity, reusability, evidence support, and potential savings without explicitly favoring a component. Proposals can also modify several components, whose counts change together when a proposal is retained or rejected. Among proposals modifying instructions, tool schemas, and executors, the local pass rates are 93.00%, 87.17%, and 86.48%, respectively. Each rate divides locally passed proposals by nonempty proposals modifying that component.

D.2 Evolution Cases and Failure Analysis

The following cases come from the four-cycle Terra runs. They illustrate accepted mechanisms, integration failures, and local score and cost trade-offs.

Coordinated batch editing on SWE-bench. In the second cycle, Parallel Patch Generation yields complementary instruction and tool changes from separate evidence batches. For example, dependency analysis flags three operations outside the final action's dependency closure in a Sphinx

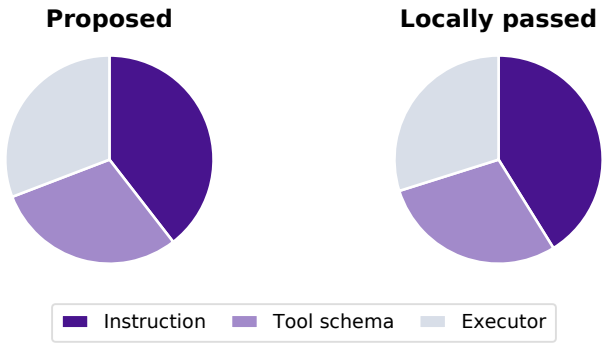


Figure 11: Distribution of harness file modifications before and after patch screening. Each modified file contributes one event, so proposals affecting multiple components contribute to multiple categories.

trajectory. The corresponding accepted batch changes instructions only. A separate accepted batch uses two Pylint trajectories containing recurring Python operations and failure-to-pivot segments to propose a batch-edit interface. One cost hotspot comprises five Python operations with 13,587 observation characters. The merged harness combines behavioral instructions with the new `apply_edits` tool. Its schema and executor accept caller-supplied exact replacements, preflight the batch, and return a bounded unified diff. This makes one coordinated edit request possible without embedding repository-specific answers.

Static checks were insufficient to establish that the tool worked. The first installed-harness validation rejected a schema-valid two-edit request because the executor reported unexpected parameters. After merger feedback and repair, validation successfully edited two tracked files, checked rollback on an invalid batch, rejected path traversal, and verified bounded diff output. The repaired cycle retained the development score at 62.50% and reduced mean inference cost from $0.3248 to $0.2580, a 20.59% reduction. All 16 development tasks were in the error-free cost comparison. This case illustrates why interface consistency must be tested after integration, even when the source files parse correctly.

Instruction-only consolidation on DeepSWE. The evidence contains successful sibling operations with identical direct dependencies and failed attempts followed by a successful pivot. The accepted harness retains an empty additional-tool registry and changes behavioral instructions. It directs the agent to batch known independent probes, avoid singleton search/read loops, pivot when repeated failures add no evidence, and use targeted verification. The cycle retains a 25.00% development score while reducing mean cost from $0.6483 to $0.5187, a 20.00% reduction over the same 16 error-free tasks. A new tool is therefore not necessary for every accepted optimization.

Rejections and tolerated losses. Patch screening rejects a SWE-bench context tool whose schema permits empty query strings while its executor rejects them. A separate proposal is rejected for allowing symlink reads outside the repository and exceeding its advertised output bound. At the iteration level, the first SWE-bench cycle preserves score but produces no cost reduction, so the candidate is rolled back.

Not all accepted updates preserve development score exactly. In DAB's fourth cycle, cost falls from $0.1132 to $0.0887, a 21.63% reduction, while the score falls from 56.25% to 50.00%. This example makes the local trade-off explicit: passing Hybrid Gating is not a guarantee of unchanged performance, even before held-out evaluation.

D.3 Useful Evidence Within a Broad Search

This case illustrates why cost information and structured trajectory annotations alone can be insufficient for safe harness optimization. A broad interaction may contain substantial irrelevant output while still producing evidence that is necessary for downstream decisions. Without dependency information, a cost-directed optimizer can identify such an interaction as an expensive hotspot, but

Table 6: Within-cycle development comparisons for the recorded Terra cases. Arrows show reference to candidate. Costs are in $/task.

Benchmark	Cycle	Score (%)	Cost	Decision
SWE-bench	1	$62.50 \rightarrow 62.50$	$0.2916 \rightarrow 0.2916$	Roll back
SWE-bench	2	$62.50 \rightarrow 62.50$	$0.3248 \rightarrow 0.2580$	Accept
DeepSWE	1	$25.00 \rightarrow 25.00$	$0.6483 \rightarrow 0.5187$	Accept
DAB	4	$56.25 \rightarrow 50.00$	$0.1132 \rightarrow 0.0887$	Accept

cannot distinguish the useful evidence from the dispensable exploration. This creates a risk of overly aggressive harness edits that suppress reasonable exploration rather than narrowing only the unnecessary part. Dependency-aware evidence addresses this ambiguity by linking the information produced during exploration to its downstream use. The case comes from the DeepSWE task `datacurve/prometheus-typed-label-sorting` in `prometheus/prometheus`.

Task and localization. The task requires a stable total order for labels containing numbers, durations, byte sizes, semantic versions, IP addresses, CIDR ranges, timestamps, and ordinary strings. It also requires precise comparison of large values, ties broken by the original strings, and separate treatment of leading whitespace. The existing `sort_by_label` and `sort_by_label_desc` functions rely on `natsort.Compare`, which does not implement these requirements. The task supplies neither an error file nor a stack trace. The agent discovers the relevant functions in `promql/functions.go` during execution. No upstream issue identifier is available in the supplied evidence.

Observed interaction sequence. Figure 12 summarizes the run. Step 5 searches repository-wide for natural ordering, labels, semantic versions, durations, and CIDR, and prints `go.mod`. Its output is truncated. Step 6 searches again for comparators and returns 86 Go code matches across 43 files. These are matching code snippets, not complete reads of 43 files. The following search is one subcommand of that interaction, which also lists directories and searches dependency names.

```
grep -RInE 'Natural|natural|Compare.*[Ll]abel|
[Ll]abel.*Compare|SortFunc'
--include='*.go' .  2>/dev/null | head -300
```

Step 7 explores `web/ui`, returning 33 matches across 11 TS and TSX files. Although this explores the wrong implementation module, function documentation provides a useful clue leading back to PromQL. Step 8 reads lines 600–720 of `promql/functions.go` and finds a usage example at `promql/bench_test.go:792`. Steps 9–11 inspect `natsort`, dependencies, and existing PromQL tests. Step 13 introduces `promql/label_sort.go` and changes both sorting call sites. Steps 15–20 add tests and resolve a compilation timeout and a test-expectation correction before targeted testing succeeds. Steps 21–22 run the full PromQL tests successfully, and step 24 commits the three changed files. All step numbers are the saved trajectory's original `step_id` values.

Mixed evidence in one interaction. The step-6 output includes the following abbreviated matches. Paths and line numbers are retained, while the ellipses omit details within each line.

```
./storage/remote/read_handler.go:97:
slices.SortFunc(sortedExternalLabels, ...)
./tsdb/wlog/checkpoint.go:441:  slices.SortFunc(refs, ...)
./tsdb/db.go:752:  slices.SortFunc(loadable, ...)
./util/stats/timer.go:87:  slices.SortFunc(timers, ...)
./rules/manager.go:425:  slices.SortFunc(rgs, ...)
./promql/functions.go:645:  slices.SortFunc(vectorVals[0],
...)
./promql/functions.go:671:  slices.SortFunc(vectorVals[0],
...)
```

The final two matches point to the sorting functions subsequently inspected and modified. Lines 642–688 contain the relevant implementation, and the example at `promql/bench_test.go:792`

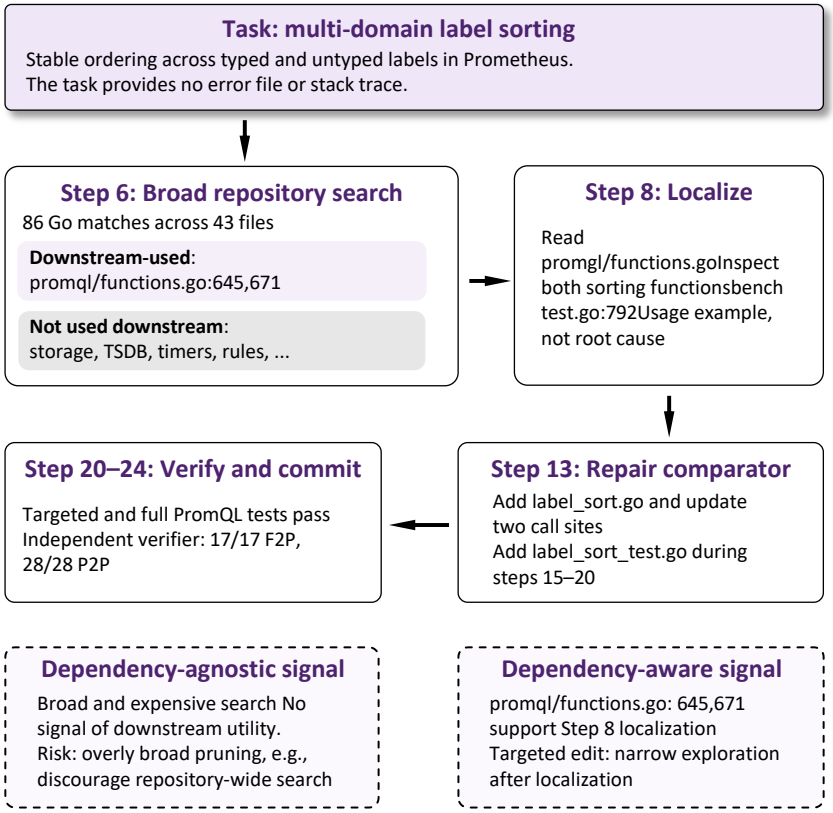


Figure 12: Why cost hotspots alone can lead to over-pruning.

shows `sort_by_label(metric, "foo", "bar")`. This example supplies usage context rather than the root cause.

The other displayed matches concern external-label ordering in remote reads, checkpoint references, database blocks, timer creation order, and rule groups. Additional unrelated matches concern WAL segments at `tsdb/wlog/wlog.go:906`, database blocks at `tsdb/db.go:1864,1946`, compaction inputs at `tsdb/compact.go:284,474`, and rule ordering at `rules/group.go:331`. Azure target sorting at `discovery/azure/azure_test.go:862` and exemplar sorting at `scrape/scrape.go:1803` and `scrape/scrape_append_v2.go:285` likewise do not implement the label comparator being repaired.

The step-7 match at `web/ui/react-app/src/pages/flags/Flags.tsx:30` sorts UI flag strings. However, `web/ui/mantine-ui/src/promql/functionDocs.tsx` provides useful cross-directory documentation, explicitly acknowledged in the agent's step-8 reasoning. The `go.mod` file, `natsort` package, and PromQL tests also provide dependency and verification context. These distinctions follow the matched code's role and its subsequent use, not simply whether the final patch changes that file. This distinction is precisely what structured annotations without dependency information cannot represent: they can expose that the interaction is broad and expensive, but not which parts of its output later support localization, repair, or verification. Consequently, treating the interaction only as a cost hotspot can encourage an overly broad pruning rule, even though part of the exploration is useful. Dependency analysis instead provides evidence for a more selective intervention: preserve information that supports downstream decisions while reducing unrelated or repeated exploration. We do not claim that every StrucAno run would remove this interaction, since no counterfactual rerun removes each match to test its necessity.

Repair and verification. The patch modifies `promql/functions.go` and adds `promql/label_sort.go` and `promql/label_sort_test.go`. The new comparator classifies values, compares within each type, and breaks ties using the original strings. It uses `math/big.Rat` for numbers, durations, and byte sizes. The ascending call

replaces the previous conditional on `natsort.Compare(lv1, lv2)` with `return compareLabelValues(lv1, lv2)`. The descending call uses the negated comparator.

At step 20, `go test ./promql -run '^TestCompareLabelValues' -count=1` passes in 0.062 seconds. The full `go test ./promql` run at steps 21–22 returns exit code zero in 21.571 seconds. Independent verification reports reward one, with 17 of 17 fail-to-pass tests and 28 of 28 pass-to-pass tests passing. The merged test report contains 45 tests, all passed. The fail-to-pass and pass-to-pass counts follow `reward.json`, rather than equating the two verifier execution batches with those categories.

Implication for harness edits. This case exposes a key limitation of dependency-agnostic cost optimization. A StruAno-style representation can identify step 6 as an expensive and broad search through its operation type, resources, outcome summary, and cost, but it has no structural signal indicating that some of the returned PromQL locations are consumed by later localization and repair. An optimizer acting on this evidence alone can therefore overgeneralize the cost hotspot into an aggressive rule that discourages repository-wide exploration.

The dependency-aware representation supports a different intervention. Rather than suppressing the search itself, it distinguishes exploration whose outputs support downstream decisions from exploration that does not. For this trajectory, the appropriate optimization is therefore to retain the initial localization behavior while narrowing subsequent inspection once the relevant implementation has been identified. This motivates the following candidate instruction.

> After locating the relevant implementation, focus further reading on that file and directly related code and tests. Follow dependencies outside the directory when needed, make the minimal repair, and run targeted tests.

The resulting distinction is between *avoiding exploration* and *avoiding unnecessary exploration*. Cost and structured annotations alone reveal that an interaction is expensive; dependency information provides evidence about which information produced by that interaction must be preserved. This case therefore illustrates why dependency-aware evidence can support more targeted harness edits than cost-aware structured annotations alone.