EVOG: A MODULAR FRAMEWORK FOR EVOLUTIONARY OPTIMIZATION OF AGENTIC WORKFLOWS

Anonymous authors
Paper under double-blind review

ABSTRACT

Designing effective agentic workflows for large language model (LLM) applications requires repeated, costly revision and evaluation. Evolutionary optimization can automate this process and existing methods have instantiated specific combinations of candidate selection, revision, example sampling, and evaluation strategies. Making these choices explicit and composable would help users construct and compare configurations suited to their task objectives and budget. Therefore, we introduce EvoG (Evolve on Graph), a modular framework for composing evolution strategies within a shared optimization loop. EvoG separates evolution into four components: a Selector that chooses parent candidates; a Generator that produces revisions; a Sampler that selects evaluation examples; and an Evaluator that assesses candidates. A shared history links candidate lineage, per-example outcomes, and execution traces to inform subsequent iterations. We evaluate EvoG on HotpotQA, MuSiQue-Ans, BigCodeBench-Hard, ScienceWorld, and WebShop. It achieves the highest mean test scores among the evaluated methods on four of five benchmarks, improving over the strongest baseline by 1.10 points on HotpotQA, 3.76 points on MuSiQue-Ans, 7.00 points on ScienceWorld, and 4.89 points on WebShop. On BigCodeBench-Hard, the best EvoG configuration achieves 35.33 points versus AgentSquare's 37.00 points, at approximately one-eighth its search cost and half its inference cost. For reproducibility, we release our code at `https://anonymous.4open.science/r/EvoG-D6C2/`.

1 INTRODUCTION

Large language model (LLM) agents solve tasks by combining reasoning, computation, and interaction with tools or external environments. Agentic workflows specify how these operations are organized and executed within a single agent or across multiple agents. Such workflows can include iterative execution and conditional routing. Applications with repetitive task patterns and relatively stable requirements can benefit from workflows tailored to their needs (Shao et al., 2024; Chen et al., 2026; Li et al., 2026). Such workflows can incorporate task-specific strategies for reflection, voting, intermediate verification, and failure recovery. However, developing and refining these designs can be time-consuming and costly: changes to prompts, control flow, or tool use require implementation and repeated execution to assess their effect on task performance. Evolutionary optimization methods automate this procedure by generating candidate changes, executing the revised workflows, and using evaluation feedback to guide subsequent revisions (Hu et al., 2025; Zhang et al., 2025; 2026).

These methods share common design choices on example sampling, candidate selection, evaluation, and revision (Agrawal et al., 2026; Xiao et al., 2026). These design choices are interdependent. The examples sampled for evaluation supply both metric scores and execution traces for candidate selection and revision. Sample sizes, the number of proposed variants, and the effort spent producing revisions also affect the resources consumed. Existing methods instantiate specific combinations of these design choices (Zhang et al., 2025). Making the choices explicit and composable would help users construct and compare alternative combinations of evolutionary optimization strategies suited to their task objectives and budget, while reusing the same execution and feedback infrastructure.

Therefore, we propose a modular framework with shared interfaces for four components: a *Selector*, a *Generator*, a *Sampler*, and an *Evaluator*. In each iteration, the Selector chooses parent candidates from the candidate pool based on evaluation scores. The Generator proposes revised candidates

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.

based on the selected parents and historical evidence. The Sampler selects examples for evaluating these candidates, and the Evaluator executes the evaluation to collect scores and execution traces. A shared history links candidate lineage, per-example outcomes, and execution traces across iterations. Candidate workflows are implemented as Python programs using LangGraph (LangChain, 2024), which defines execution graphs with nodes performing operations and edges specifying control flow. Editing these programs allows prompts, control flow, and tool implementations to be revised together. We therefore name the resulting evolutionary optimization framework *EvoG (Evolve on Graph)*. We instantiate an evidence-guided Generator that connects execution traces and prior revisions with their evaluation outcomes to guide subsequent workflow edits.

We evaluate EvoG on five benchmarks: HotpotQA and MuSiQue-Ans for multi-hop question answering (Yang et al., 2018; Trivedi et al., 2022), BigCodeBench-Hard for code generation (Zhuo et al., 2025), and ScienceWorld and WebShop for interactive task completion (Wang et al., 2022; Yao et al., 2022). In our runs, EvoG achieves the highest mean test scores among the evaluated methods on four of the five benchmarks, improving over the strongest baseline for each task by 1.10 points on HotpotQA, 3.76 points on MuSiQue-Ans, 7.00 points on ScienceWorld, and 4.89 points on WebShop. On BigCodeBench-Hard, the strongest EvoG configuration achieves 35.33 points compared with AgentSquare's 37.00 points, at approximately one-eighth of its search cost and half its inference cost. In our QA experiments, observed Selector rankings vary across Sampler settings, motivating comparisons of selection and sampling strategies in combination. Evidence ablations further suggest that workflow revision benefits from detailed execution traces and accumulated analysis of earlier edits and their evaluation outcomes.

Our contributions are:

- We introduce EvoG, a modular framework for composing parent selection, candidate generation, example sampling, and evaluation within a common evolution loop, allowing alternative strategies to reuse the same execution and feedback infrastructure.
- We instantiate an evidence-guided Generator for revising agentic workflows. It connects detailed execution traces and prior edits with their evaluation outcomes to guide diagnosis and revision, followed by validation and review.
- We characterize performance–cost trade-offs across five benchmarks, comparing Selector–Sampler combinations on question answering and Selector policies on code-generation and interactive tasks. Our results show that configuration effectiveness varies across tasks. Ablation studies further highlight the value of detailed execution traces and insights accumulated from prior edits and evaluations in guiding subsequent revisions.

2 Related Work

Automated workflow optimization. Automated workflow optimization improves the operations within an agentic workflow and the structure connecting them. GPTSwarm optimizes node prompts and inter-agent connections, while MASS coordinates prompt and topology optimization through successive stages (Zhuge et al., 2024; Zhou et al., 2026). ADAS uses a meta-agent to propose executable agent programs using an archive of prior designs and results, whereas AFlow searches code-defined workflows using operators, execution feedback, and a search procedure inspired by Monte Carlo tree search (Hu et al., 2025; Zhang et al., 2025). Search strategies also address task variation and resource constraints. AdaptFlow combines subtask-specific workflow adaptation with consolidation into a reusable initialization; AgentEvo combines cost-aware search with adaptive exploration, refinement, and crossover (Zhu et al., 2025; Xiao et al., 2026). Another recurring choice is how execution evidence informs revisions. GEPA reflects on execution traces to revise prompts and uses per-example performance to select complementary candidates (Agrawal et al., 2026). ReCreate uses execution experience to propose instance-level revisions and consolidates them into domain-level scaffold updates, while Meta-Harness lets a coding-agent proposer inspect stored programs, scores, and traces when revising an executable harness (Hao et al., 2026; Lee et al., 2026). AHE and TPGO make change histories part of this feedback: AHE compares predicted effects of edits with later outcomes, while TPGO retrieves problem–edit–outcome records to guide proposals (Lin et al., 2026; He et al., 2026). EvoTool attributes failures to tool-use policy modules to target mutations and preserves candidates with complementary instance-level successes (Yang et al., 2026). These works provide precedents for trace-guided revision and historical evidence reuse. Our

proposed EvoG instantiates an evidence-guided Generator within a framework that allows it to be combined with different sampling, selection, and evaluation strategies.

Modular frameworks for agent optimization. Some existing works explore the reuse of agent components and optimization methods. AgentSquare defines standard interfaces for planning, reasoning, tool-use, and memory modules, allowing these components to be revised and recombined during agent search (Shang et al., 2025). DSPy lets users build LM programs and choose or combine optimizers to improve them (Khattab et al., 2024). TextGrad propagates natural-language feedback backward through computation graphs to guide updates to prompts or code (Yuksekgonul et al., 2025). EvoAgentX integrates existing optimizers, including TextGrad, MIPRO, and AFlow, into a common platform for workflow construction, execution, and evaluation (Wang et al., 2025). Tressoir uses a supervising agent to execute tasks and improve agent systems, guided by editable blueprints containing instructions and reusable code (Ngom et al., 2026). EvoG makes the decisions within an evolutionary optimizer separately configurable: users can combine different strategies for sampling evaluation examples, selecting parent candidates, generating revisions, and evaluating candidates within one shared loop. Common interfaces and a shared history of candidate lineage, per-example outcomes, and execution traces support this recombination, allowing users to compare configurations for their task and budget while reusing the same execution and feedback infrastructure.

3 METHOD

EvoG organizes workflow evolution as a stateful search over candidate programs. In each round, a parent candidate is selected and revised. The revised candidate is evaluated on chosen examples, and the resulting observations are added to a shared history. Explicit component interfaces make each decision in this cycle configurable. We first formalize the cycle, then describe the policies used here and the evidence-guided Generator that produces program revisions. Figure 1 summarizes the framework and its information flow.

3.1 EVOLUTION LOOP

The framework can be generalized to one-to-many proposals that explore different improvement directions from a parent candidate, or many-to-one proposals that combine the strengths of multiple parents into a new candidate. We describe the one-to-one setting used in all our experiments.

Let $\mathcal{D}$ be a training pool, c_0 an initial workflow, and $m(c, x)$ the task metric observed for candidate c on example x. In round t, b_t denotes the selected parent candidate, and c_t denotes the revised candidate generated from b_t. At initialization, the Sampler chooses a nonempty *select set* $S \subseteq \mathcal{D}$ and keeps it fixed as a common basis for comparing candidates. In round t, it may add a disjoint *propose set* $F_t \subseteq \mathcal{D} \setminus S$. The new candidate c_t is evaluated on both sets, and per-example results from both provide feedback for later revisions.

The history H_t stores the initial evaluation and the first t iteration records. Each record links the selected parent b_t, the generated candidate c_t, the sampled sets (S, F_t), and evaluation E_t. The history retains candidate lineage, per-example results and traces, and resource usage. The Selector chooses b_t from eligible candidates in H_{t-1}, and the Generator produces a revised candidate c_t from b_t and prior evidence. The Sampler then chooses F_t, and the Evaluator runs c_t on $S \cup F_t$ before the complete iteration is appended to the history (Algorithm 1).

We summarize the shared select-set scores by

$$\widehat{J}_S(c) = \frac{1}{|S|} \sum_{x \in S} m(c, x), \tag{1}$$

computed from stored evaluations. Selector policies use outcomes on S, with some also considering lineage. After the final iteration, we choose the eligible candidate with the highest stored select-set mean.

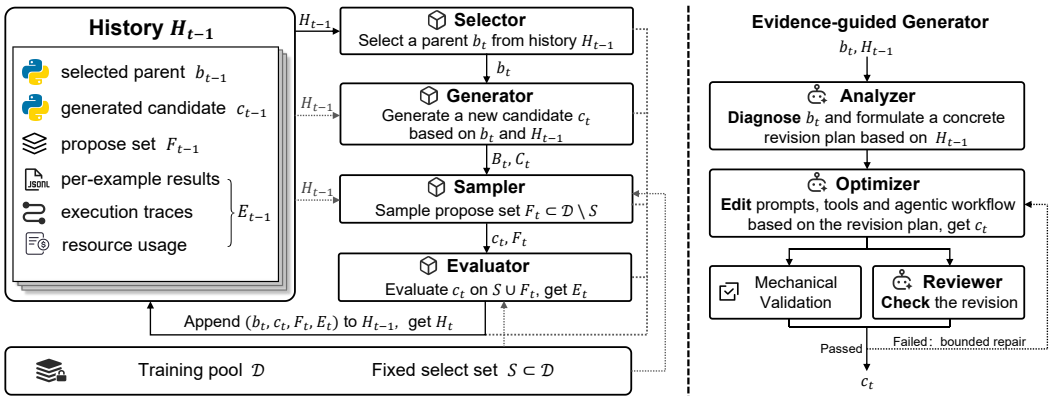


Figure 1: Overview of EvoG. Left: the Selector chooses a parent workflow b_t from history H_{t-1}, and the Generator produces a revised candidate c_t using historical evidence. The Sampler chooses an optional propose set $F_t \subseteq \mathcal{D} \setminus S$, and the Evaluator runs c_t on $S \cup F_t$, where the fixed select set S supports candidate comparison. The iteration record, including candidate lineage, sampled set, per-example results, execution traces, and resource usage, is appended to history for subsequent rounds. Right: the evidence-guided Generator uses an Analyzer to diagnose the selected parent and plan a revision, an Optimizer to edit prompts, tools, and workflow code, and mechanical validation and a Reviewer to check the revision, with bounded repair by the Optimizer when checks fail.

Algorithm 1 EvoG evolution loop

Require: Training pool $\mathcal{D}$, seed workflow c_0, rounds T

$S \leftarrow \mathsf{ChooseSelectSet}(\mathcal{D})$ ▷ fixed for all rounds
$F_0 \leftarrow \mathsf{InitializeProposeSet}(\mathcal{D} \setminus S)$
$E_0 \leftarrow \mathsf{Evaluate}(c_0, S \cup F_0)$
$H_0 \leftarrow \mathsf{Initialize}(c_0, F_0, E_0)$
for $t = 1, \ldots, T$ **do**
  $b_t \leftarrow \mathsf{Select}(H_{t-1})$
  $c_t \leftarrow \mathsf{Generate}(b_t, H_{t-1})$
  $F_t \leftarrow \mathsf{SampleProposeSet}(H_{t-1}, b_t, c_t, \mathcal{D} \setminus S)$
  $E_t \leftarrow \mathsf{Evaluate}(c_t, S \cup F_t)$
  $H_t \leftarrow H_{t-1} \oplus (b_t, c_t, F_t, E_t)$
end for
return H_T

3.2 Workflow Representation and Loop Policies

The loop uses one candidate representation and interchangeable policies for parent selection, example sampling, and evaluation. We describe the Generator's revision process in Section 3.3.

Workflow representation. A candidate is a directory of Python code that builds an executable agent graph. Every candidate receives the same read-only task interface, which exposes available models and task tools and specifies the required output format. Candidate revisions can change prompts, control flow, and candidate-owned tools while preserving this interface. We retain each revision and its lineage. ADAS and AFlow also represent agents or workflows as executable code (Hu et al., 2025; Zhang et al., 2025).

Selector. We implement four Selector policies, each choosing a parent candidate b_t in each round based on the search history H_{t-1}. *Best-observed* chooses the highest-scoring candidate. *Soft-mixed* combines score-weighted and uniform sampling over strong candidates and the seed, following AFlow's parent-selection idea (Zhang et al., 2025). *UCT-tree* combines the scores of a candidate and its direct revisions with an exploration bonus, and may restart from the seed (Kocsis & Szepesvári, 2006). *Pareto-coverage*, inspired by GEPA's instance-wise winner sampling, weights candidates by

how often they attain the best score on individual select examples, sharing credit across ties (Agrawal et al., 2026). Appendix A gives the exact selection rules; Appendix B.3 reports their experimental parameter values.

Sampler. We study four sampling settings. *Full-pool* uses $S = \mathcal{D}$ and $F_t = \emptyset$. *Fixed-subset* samples S once and uses $F_t = \emptyset$. *Random-feedback* also fixes S, but draws a fresh F_t uniformly from $\mathcal{D} \setminus S$ in each round, including initialization. *Low-score-feedback* uses the same random initialization; in later rounds, it prioritizes low-scoring examples from the selected parent's most recent propose evaluation and randomly fills the remaining slots from $\mathcal{D} \setminus S$. Examples may reappear in different rounds. The evaluation scores used for candidate selection come only from S; observations on both sets can inform subsequent revisions.

Evaluator. The Evaluator assesses candidate performance on the examples chosen by the Sampler and returns evaluation feedback. Some policies reduce evaluation cost through score prediction or early stopping. For example, AgentSquare predicts performance for some module combinations (Shang et al., 2025); PACE can accept a challenger before evaluating every example when a sequential test on paired win–loss outcomes reaches its threshold (Shawn, 2026). To compare candidates on the shared select set, we execute each new candidate on all examples in $S \cup F_t$ and retain the per-example results as feedback for later revisions.

3.3 Evidence-Guided Generation

We implement an evidence-guided Generator that uses per-example feedback, execution traces, and records linking past revisions to their evaluation outcomes to guide workflow diagnosis and code editing. It has three agents: an *Analyzer* that diagnoses the parent's behavior and plans a revision, an *Optimizer* that edits a copy of the parent workflow, and a *Reviewer* that checks the revised program. It combines trace-guided reflection, as explored in GEPA and ReCreate (Agrawal et al., 2026; Hao et al., 2026), with history-guided code revision, as explored in Meta-Harness (Lee et al., 2026).

At round t, the Analyzer can inspect evidence in H_{t-1}: the selected parent's per-example scores, outputs, and errors on S and earlier propose sets, together with complete traces selected from higher- and lower-scoring cases. Its continuing context links prior workflow versions and edits to their comparable select-set results. These views provide broad behavioral coverage, execution detail, and a record of previous revisions for forming a concrete revision plan.

The Optimizer uses that plan to produce c_t by revising prompts, control flow, and candidate-owned tools in the copied workflow code. Mechanical validation and the read-only Reviewer then check the revision for Python compilation, preservation of the task interface, successful workflow construction, and concrete code or execution-contract defects. Identified issues can trigger bounded repair by the Optimizer, followed by the same checks. The candidate and its generation status enter H_t; only candidates marked ready are eligible as later parents or final outputs.

4 Experiments

All search and task-evaluation model calls use DeepSeek-V4.1-Flash (DeepSeek-AI, 2026) for EvoG and the baselines. We run each applicable search once per task and configuration, evaluate its selected candidate on the test set three times, and report the mean; fixed solvers are tested in the same way. Appendix B.3 specifies the shared budgets and execution limits.

4.1 Question Answering

Benchmarks and metrics. We evaluate on HotpotQA and MuSiQue-Ans, with 200 search and 800 test examples each (Yang et al., 2018; Trivedi et al., 2022). For HotpotQA, we use the same data split as AFlow (Zhang et al., 2025). MuSiQue-Ans search examples come from the official training split and test examples from the development split. We report normalized answer-token F1 on a 0–100 scale. Appendix B.1 gives the split and scoring details.

Baselines. We compare with direct input–output prompting (IO), zero-shot chain-of-thought prompting (CoT) (Kojima et al., 2022), ADAS (Hu et al., 2025), AFlow (Zhang et al., 2025), and

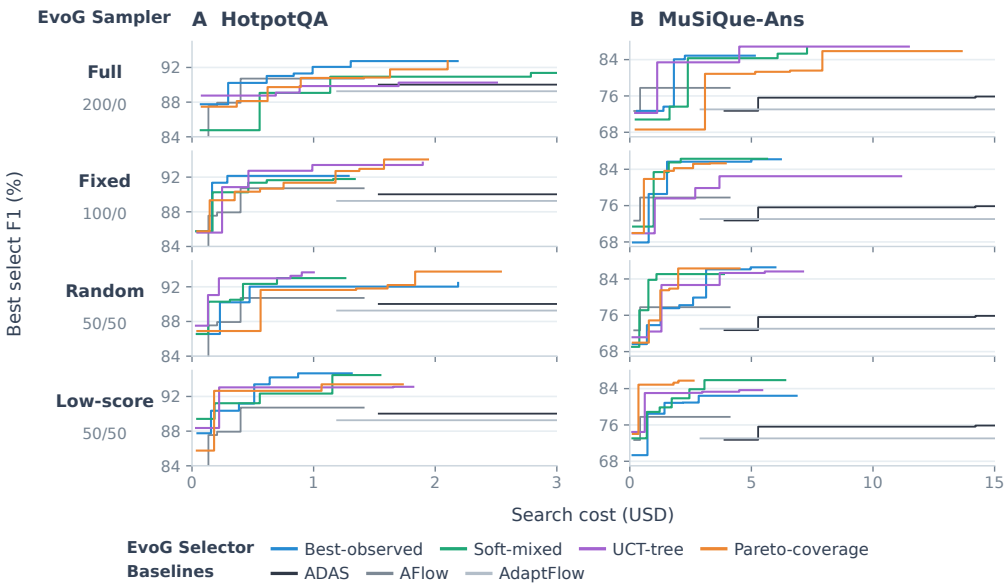


Figure 2: QA search trajectories: columns show (A) HotpotQA and (B) MuSiQue-Ans; rows show the four EvoG Samplers from Table 1. Each step curve shows best-so-far select-set F1 from one search run. Each panel compares four EvoG Selectors by color, with three gray baselines repeated for reference; two-row legend below the panels groups EvoG Selectors and baselines. Rows share axis limits within each benchmark. Scores use the common 50-question select set, while costs include all optimization and search evaluation in the original runs. Curves are clipped at the displayed limits.

AdaptFlow (Zhu et al., 2025). We report AdaptFlow with test-time adaptation (TTA) separately. Their search mechanisms and task adaptations are detailed in Appendix B.2.

EvoG configurations. We cross the four Selectors in Section 3.2 with four Sampler settings: full-pool ($|S|/|F_t| = 200/0$), fixed-subset ($100/0$), random-feedback ($50/50$), and low-score-feedback ($50/50$). Each run starts from a task-specific, single-agent seed and generates one candidate per round for 10 rounds. Final selection maximizes the stored select-set mean among eligible workflows. Appendix B.3 gives implementation details.

Results. Table 1 shows that EvoG exceeds the strongest baselines by 1.10 points on HotpotQA and 3.76 points on MuSiQue-Ans. In our runs, the relative performance of Selectors varies across Sampler settings. For example, on MuSiQue-Ans, soft-mixed scores higher than UCT-tree with full-pool sampling (84.48 vs. 83.62), but lower with random-feedback sampling (82.49 vs. 84.65). This ranking reversal supports evaluating selection and sampling strategies in combination. Random-feedback also outperforms low-score-feedback in six of eight matched comparisons, indicating that repeatedly prioritizing low-scoring examples is not uniformly beneficial. Their differing feedback coverage and repetition (Appendix Figure 8) change revision evidence at the same feedback batch size.

Most EvoG configurations exceed ADAS's and AdaptFlow's first reported select-set scores within the corresponding search cost (Figure 2). Appendix Figure 6 separates optimization and candidate-evaluation costs. At test time, cheaper baselines and higher-scoring EvoG configurations occupy different parts of the Pareto frontier (Figure 3), offering different score–cost trade-offs.

4.2 Code Generation and Interactive Tasks

Benchmarks and metrics. We use fixed 20-search/100-test subsets of BigCodeBench-Hard, ScienceWorld, and WebShop (Zhuo et al., 2025; Wang et al., 2022; Yao et al., 2022). We report the percentage of BigCodeBench-Hard tasks passing all unit tests, the native ScienceWorld task score,

Table 1: QA test answer-token F1 (0–100; 800 examples per benchmark). EvoG configurations are grouped by parent Selector, each evaluated with four Samplers. Sampler entries give $|S|/|F_t|$: the sizes of the fixed select and per-round propose sets within the 200-example search pool. Baselines follow their own procedures; – in either configuration column indicates that the EvoG component does not apply. Scores average three test executions after one search where applicable. Bold, single underline, and double underline mark the first, second, and third scores in each benchmark column.

Method	EvoG configuration		HotpotQA	MuSiQue-Ans
	Selector	Sampler		
IO	–	–	63.37	80.14
CoT	–	–	80.65	79.35
ADAS	–	–	80.71	80.89
AFlow	–	–	79.86	79.34
AdaptFlow	–	–	82.60	80.19
AdaptFlow + TTA	–	–	81.98	79.22
EvoG	Best-observed	Full (200/0)	82.70	83.75
		Fixed (100/0)	<u>83.38</u>	83.05
		Random (50/50)	82.52	82.80
		Low-score (50/50)	83.09	77.85
	Soft-mixed	Full (200/0)	**83.70**	<u>84.48</u>
		Fixed (100/0)	82.98	82.00
		Random (50/50)	<u>83.51</u>	82.49
		Low-score (50/50)	82.18	82.93
	UCT-tree	Full (200/0)	83.05	83.62
		Fixed (100/0)	82.73	83.12
		Random (50/50)	83.10	**84.65**
		Low-score (50/50)	81.99	82.14
	Pareto-coverage	Full (200/0)	82.65	<u>84.03</u>
		Fixed (100/0)	82.55	82.09
		Random (50/50)	82.78	83.61
		Low-score (50/50)	82.20	83.38

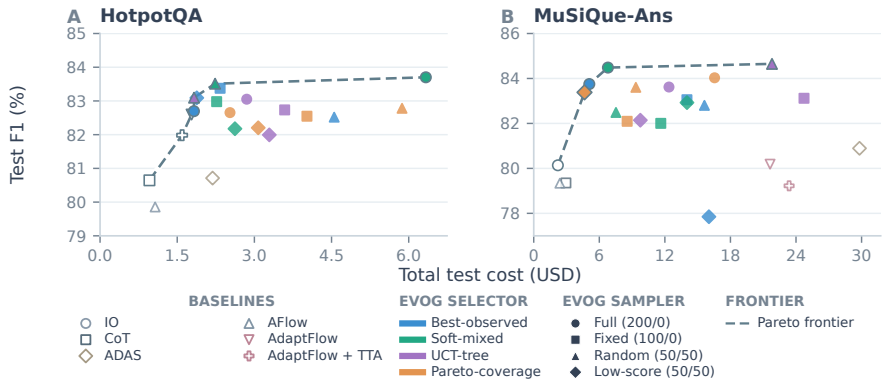


Figure 3: QA test answer-token F1 against total cost over three test executions: (A) HotpotQA and (B) MuSiQue-Ans. Colors identify EvoG Selectors and symbols identify Sampler settings; hollow symbols denote baselines. Pareto optimality is computed over all evaluated configurations; the dashed line joins the visible frontier points. IO (63.37 F1, 0.56 USD) is clipped at the displayed limits in (A), but it also lies on the Pareto frontier.

and the WebShop purchase reward, all on a 0–100 scale. Appendix B.1 describes the subsets and scoring.

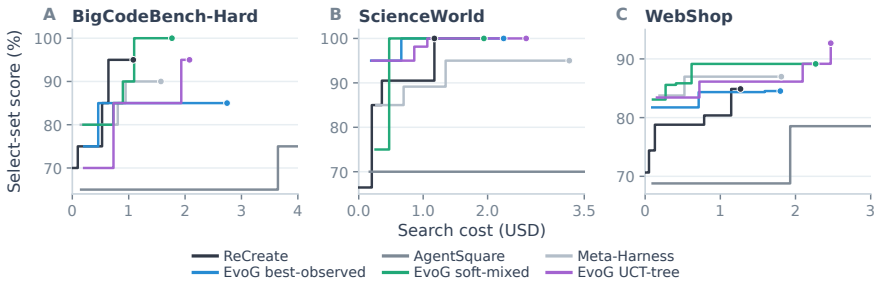


Figure 4: Search scores on code generation and interactive tasks against cumulative search cost in USD. Panels (A)–(C) show BigCodeBench-Hard, ScienceWorld, and WebShop; colors identify the three EvoG selectors, and gray shades distinguish the three baselines in the legend below. Scores use the fixed 20-task select set for each benchmark. Curves are clipped at the displayed limits.

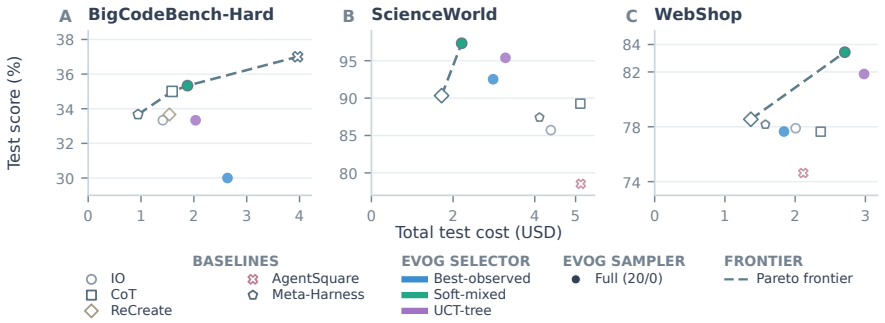


Figure 5: Test scores on code generation and interactive tasks against total cost over three test executions. Panels (A)–(C) show BigCodeBench-Hard pass rate, ScienceWorld task score, and WebShop purchase reward. The legend uses hollow symbols for baselines, color for EvoG Selectors, and circles for the Full $(20/0)$ Sampler; the dashed line joins the observed Pareto-optimal configurations within each benchmark.

Baselines. We compare with IO, CoT, ReCreate (Hao et al., 2026), AgentSquare (Shang et al., 2025), and Meta-Harness (Lee et al., 2026). The QA implementations of ADAS, AFlow, and AdaptFlow used here produce answers without the benchmark action and submission interfaces required by these benchmarks, so we do not carry them over. We adapt each baseline in this group to each benchmark's task and tool interfaces; Appendix B.2 describes the search mechanisms and adaptations.

EvoG configurations. Every candidate is evaluated on the full, fixed set of 20 search tasks. We compare best-observed, soft-mixed, and UCT-tree selection. With only 20 search tasks, subset sampling and per-example Pareto-coverage comparisons could be sensitive to individual tasks; we therefore do not vary the Sampler or run Pareto-coverage here. As in QA, each configuration uses a task-specific seed, 10 revision rounds, and final selection by stored select-set mean.

Results. With full-pool sampling fixed, soft-mixed outperforms best-observed by 4.81–5.79 points across the three benchmarks (Table 2). Parent selection thus remains an important configuration choice even when every candidate is evaluated on all search examples. Soft-mixed achieves the highest ScienceWorld and WebShop scores, exceeding the strongest baselines by 7.00 and 4.89 points, respectively.

On BigCodeBench-Hard, soft-mixed scores 1.67 points below AgentSquare while using approximately one-eighth of its search cost and half of its inference cost (Figures 4 and 5). Soft-mixed lies on the test score–cost Pareto frontier for all three benchmarks; ReCreate offers lower-cost alternatives on ScienceWorld and WebShop. The preferred operating point therefore depends on the

Table 2: Scores on code generation and interactive tasks (0–100; 100 test tasks per benchmark). BigCodeBench-Hard is the percentage passing all unit tests; ScienceWorld uses its native task score, and WebShop its native purchase reward. All searchable methods use the same fixed 20-task search set; scores average three test executions after one search where applicable. Bold, single underline, and double underline mark the first, second, and third scores in each benchmark column.

Method	BigCodeBench-Hard	ScienceWorld	WebShop
IO	33.33	85.72	77.89
CoT	<u>35.00</u>	89.26	77.64
ReCreate	33.67	90.33	<u>78.55</u>
AgentSquare	**37.00**	78.54	74.63
Meta-Harness	33.67	87.41	78.16
EvoG best-observed	30.00	<u>92.52</u>	77.65
EvoG soft-mixed	<u>35.33</u>	**97.33**	**83.44**
EvoG UCT-tree	33.33	<u>95.38</u>	<u>81.85</u>

Table 3: Evidence ablations under soft-mixed selection and full-pool sampling. Scores are on each benchmark's 0–100 scale and average three test executions after one search run. Bold marks the standard configuration.

Task	Full evidence	No traces to Analyzer	Reset Analyzer
HotpotQA	**83.70**	82.43	82.73
MuSiQue-Ans	**84.48**	83.65	82.71
BigCodeBench-Hard	**35.33**	31.67	33.33
ScienceWorld	**97.33**	95.26	95.93
WebShop	**83.44**	77.30	78.20

desired score and resource budget. Appendix Figure 7 attributes most of the BigCodeBench-Hard search-cost gap to candidate evaluation.

4.3 Ablations

We use soft-mixed selection and full-pool sampling on all five tasks. One variant withholds detailed traces from the Analyzer while retaining evaluation summaries; the other resets its context between rounds. All other settings match the standard configuration.

Both ablations reduce performance on all five benchmarks (Table 3), with the largest decreases on WebShop: 6.14 points without detailed traces and 5.24 points after resetting the Analyzer. Traces expose the intermediate decisions behind an outcome, while persistent analysis links earlier edits to subsequent results. Both inform what to change and which previous decisions to revisit. Appendix C illustrates their use in revising WebShop candidate comparison and search control.

5 Conclusion

We introduced EvoG, a modular framework for composing parent selection, candidate generation, example sampling, and evaluation within a shared workflow-evolution loop. Across five benchmarks, EvoG configurations achieve the highest mean test scores on four, while revealing configuration-dependent performance and trade-offs between search cost, inference cost, and task scores. Selector rankings change with the Sampler, supporting the joint comparison of evolution strategies. Evidence ablations further support using detailed execution traces and persistent analysis to guide program revision. EvoG provides a common execution and feedback infrastructure for exploring these choices and composing strategies suited to task objectives and resource budgets.

AI USE STATEMENT

We used generative AI tools to assist with code implementation and to correct grammar and improve the clarity and readability of the manuscript. This assistance did not extend to formulating the research ideas, designing or conducting experiments, or analyzing or interpreting the experimental results. The authors reviewed the AI-assisted contributions and take full responsibility for the final content of this work, including text, claims, code, and other artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work adheres to the ICLR Code of Ethics. To the best of our knowledge, the methods and experiments presented in this paper raise no specific ethical concerns requiring additional discussion.

REPRODUCIBILITY STATEMENT

To support reproducibility, we describe the EvoG framework and its optimization procedure in Section 3, and the experimental protocols and evaluation settings in Section 4. Appendix A specifies the Selector rules, and Appendix B documents dataset construction, scoring, baseline implementations, and configuration and execution settings. The code is available at: `https://anonymous.4open.science/r/EvoG-D6C2/`.

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, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. In *International Conference on Learning Representations*, 2026. URL `https://proceedings.iclr.cc/paper_files/paper/2026/hash/0e9e708b6f48e14fd0ac29e167413f76-Abstract-Conference.html`.

Zheyu Chen, Huiteng Zhuang, Zhuohuan Li, and Chuanhao Li. Specification-driven generation and evaluation of discrete-event world models via the DEVS formalism, 2026. URL `https://arxiv.org/abs/2603.03784`.

DeepSeek-AI. DeepSeek-V4.1-Flash: Pushing the Limits of KV Cache Compression. arXiv preprint arXiv:2609.19969, 2026. URL `https://arxiv.org/abs/2609.19969`.

Zhezheng Hao, Hong Wang, Jian Luo, Jianqing Zhang, Yuyan Zhou, Qiang Lin, Can Wang, Hande Dong, and Jiawei Chen. ReCreate: Reasoning and Creating Domain Agents Driven by Experience. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 31018–31046, 2026. doi: 10.18653/v1/2026.acl-long.1432. URL `https://aclanthology.org/2026.acl-long.1432/`.

Shan He, Runze Wang, Zhuoyun Du, Huiyu Bai, Zouying Cao, Yu Cheng, and Bo Zheng. Learning to Evolve: A Self-Improving Framework for Multi-Agent Systems via Textual Parameter Graph Optimization. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 30719–30732, 2026. doi: 10.18653/v1/2026.findings-acl.1534. URL `https://aclanthology.org/2026.findings-acl.1534/`.

Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems. In *International Conference on Learning Representations*, 2025. URL `https://proceedings.iclr.cc/paper_files/paper/2025/hash/36b7acf6f6010652b3f2a433774a66fe-Abstract-Conference.html`.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.),

International Conference on Learning Representations, volume 2024, pp. 54928–54958, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/f1cf02ce09757f57c3b93c0db83181e0-Paper-Conference.pdf.

Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pp. 282–293, 2006. doi: 10.1007/11871842_29. URL https://doi.org/10.1007/11871842_29.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large Language Models are Zero-Shot Reasoners. In *Advances in Neural Information Processing Systems*, volume 35, pp. 22199–22213, 2022. URL https://proceedings.neurips.cc/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html.

LangChain. LangGraph. https://github.com/langchain-ai/langgraph, 2024. Software repository. Accessed: 2026-09-25.

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

Chuanhao Li, Xiaoan Xu, Dirk Bergemann, Ethan X. Fang, Yehua Wei, and Zhuoran Yang. COOPA: A modular LLM agent architecture for operations research problems, 2026. URL https://arxiv.org/abs/2606.27611.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic Harness Engineering: Observability-Driven Automatic Evolution of Coding-Agent Harnesses. arXiv preprint arXiv:2604.25850, 2026. URL https://arxiv.org/abs/2604.25850.

Amadou Latyr Ngom, Ziniu Wu, Jason Mohoney, James Moore, Alex Zhang, Samuel Madden, and Tim Kraska. Tressoir: Unifying Online, Offline, and HIL Design and Evolution of Multi-Agent Systems via Interpretable Blueprints. In *Proceedings of the ACM Conference on AI and Agentic Systems*, pp. 46–60, 2026. doi: 10.1145/3786335.3813143. URL https://doi.org/10.1145/3786335.3813143.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. AgentSquare: Automatic LLM Agent Search in Modular Design Space. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=mPdmDYIQ7f.

Yijia Shao, Yucheng Jiang, Theodore Kanell, Peter Xu, Omar Khattab, and Monica Lam. Assisting in writing Wikipedia-like articles from scratch with large language models. In Kevin Duh, Helena Gomez, and Steven Bethard (eds.), *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 6252–6278, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.naacl-long.347. URL https://aclanthology.org/2024.naacl-long.347/.

Zayx Shawn. PACE: Anytime-Valid Acceptance Tests for Self-Evolving Agents, 2026. URL https://arxiv.org/abs/2606.08106.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. MuSiQue: Multihop Questions via Single-hop Question Composition. *Transactions of the Association for Computational Linguistics*, 10:539–554, 2022. doi: 10.1162/tacl_a_00475. URL https://aclanthology.org/2022.tacl-1.31/.

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. ScienceWorld: Is your Agent Smarter than a 5th Grader? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 11279–11298, 2022. doi: 10.18653/v1/2022.emnlp-main.775. URL https://aclanthology.org/2022.emnlp-main.775/.

Yingxu Wang, Siwei Liu, Jinyuan Fang, and Zaiqiao Meng. EvoAgentX: An Automated Framework for Evolving Agentic Workflows. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 643–655, 2025. doi: 10.18653/v1/2025.emnlp-demos.47. URL https://aclanthology.org/2025.emnlp-demos.47/.

Bo Xiao, Haiyang Liu, Yitong Wang, Yanning Hou, Ke Xu, and Xingyi Zhang. AgentEvo: cost-aware agentic workflow generation via adaptive multi-stage evolution. *Complex & Intelligent Systems*, 12:208, 2026. doi: 10.1007/s40747-026-02325-0. URL https://doi.org/10.1007/s40747-026-02325-0.

Shuo Yang, Soyeon Caren Han, Xueqi Ma, Yan Li, Mohammad Reza Ghasemi Madani, and Eduard Hovy. EVOTOOL: Self-Evolving Tool-Use Policy Optimization in LLM Agents via Blame-Aware Mutation and Diversity-Aware Selection. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 43553–43572, 2026. doi: 10.18653/v1/2026.acl-long.2016. URL https://aclanthology.org/2026.acl-long.2016/.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, 2018. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. WebShop: Towards Scalable Real-World Web Interaction with Grounded Language Agents. In *Advances in Neural Information Processing Systems*, volume 35, pp. 20744–20757, 2022. URL https://papers.nips.cc/paper_files/paper/2022/hash/82ad13ec01f9fe44c01cb91814fd7b8c-Abstract-Conference.html.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639:609–616, 2025. doi: 10.1038/s41586-025-08661-4. URL https://www.nature.com/articles/s41586-025-08661-4.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/aa5f5e6eb6f613ec412f1d948dfa21a5-Abstract-Conference.html.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, XiongHui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating Agentic Workflow Generation. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/5492ecbce4439401798dcd2c90be94cd-Abstract-Conference.html.

Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö. Arık. Multi-Agent Design: Optimizing Agents with Better Prompts and Topologies. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2502.02533.

Runchuan Zhu, Bowen Jiang, Lingrui Mei, Fangkai Yang, Lu Wang, Haoxiang Gao, Fengshuo Bai, Pu Zhao, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. AdaptFlow: Adaptive Workflow Optimization via Meta-Learning. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 3287–3302, 2025. doi: 10.18653/v1/2025.findings-emnlp.175. URL https://aclanthology.org/2025.findings-emnlp.175/.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language Agents as Optimizable Graphs. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62743–62767, 2024. URL https://proceedings.mlr.press/v235/zhuge24a.html.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, James Hoang, Armel Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro von Werra. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a6a90bcc2aa470c3871b2d39a67d26e8-Abstract-Conference.html.

A Selector Policy Definitions

The four Selector policies implemented here each select an evaluated candidate for revision. At round t, let $\mathcal{V}_t$ be the eligible, evaluated candidates in H_{t-1}, c_0 the initial workflow, and b_t the selected parent in Algorithm 1. All candidates are evaluated on the same fixed example set S; $m(c, x)$ is the score of candidate c on example x, and $q_c = \widehat{J}_S(c)$ is its mean score (Equation 1). For randomized policies, write $P_t(c) = \Pr(b_t = c)$. Experimental parameter values and tie-breaking details are given in Appendix B.3.

Best-observed. This policy selects the candidate with the highest mean score:

$$b_t \in \arg\max_{c \in \mathcal{V}_t} q_c. \tag{2}$$

Soft-mixed. This policy mixes a preference for higher scores with uniform sampling among the top candidates and the initial workflow. Let $\mathrm{Top}_k(\mathcal{V}_t)$ contain up to k candidates with the highest q_c, and define $A_t = \mathrm{Top}_k(\mathcal{V}_t) \cup \{c_0\}$. For $c \in A_t$,

$$P_t(c) = (1-\lambda)\frac{\exp(q_c/\tau)}{\sum_{u \in A_t} \exp(q_u/\tau)} + \frac{\lambda}{|A_t|}, \tag{3}$$

where smaller $\tau > 0$ gives a stronger preference for higher scores and $\lambda \in [0, 1]$ is the uniform mixture weight. Candidates outside A_t have zero selection probability.

UCT-tree. This policy favors candidates that perform well together with their direct revisions, while giving an exploration bonus to candidates with fewer such revisions. If workflow u is generated by directly revising workflow c, we call c its *parent* and u its *child*. Let $D_t(c) \subseteq \mathcal{V}_t$ contain the evaluated children of c, and define $n_t(c) = |D_t(c)|$ and $N_t = \sum_{u \in \mathcal{V}_t} n_t(u)$. For each $c \in \mathcal{V}_t$,

$$U_t(c) = \frac{q_c + \sum_{u \in D_t(c)} q_u}{1 + n_t(c)} + \beta\sqrt{\frac{\log(N_t + 1)}{1 + n_t(c)}}, \tag{4}$$

where $\beta \geq 0$ controls exploration. Let $M_t = \arg\max_{c \in \mathcal{V}_t} U_t(c)$. With probability $\rho \in [0, 1]$, the policy selects the initial workflow c_0; otherwise, it samples uniformly from M_t:

$$P_t(c) = \rho\,\mathbf{1}[c = c_0] + (1-\rho)\frac{\mathbf{1}[c \in M_t]}{|M_t|}. \tag{5}$$

Pareto-coverage. This policy samples candidates according to how often they achieve the best score on individual examples, sharing each example's credit equally among ties. For each $x \in S$, define the candidates whose scores are within $\epsilon \geq 0$ of the best score:

$$W_t(x) = \left\{ c \in \mathcal{V}_t : m(c, x) \geq \max_{u \in \mathcal{V}_t} m(u, x) - \epsilon \right\}. \tag{6}$$

Each candidate in $W_t(x)$ receives credit $1/|W_t(x)|$ for example x. Its total credit and selection probability are

$$a_t(c) = \sum_{x \in S} \frac{\mathbf{1}[c \in W_t(x)]}{|W_t(x)|}, \qquad P_t(c) = \frac{a_t(c)}{\sum_{u \in \mathcal{V}_t} a_t(u)}. \tag{7}$$

B Experimental Details

B.1 Dataset Construction and Scoring

We fix non-overlapping search and test subsets before optimization and use the same subsets for all methods on a task. The search set supplies workflow feedback and selection. AdaptFlow + TTA separately uses unlabeled test inputs for adaptation, as described in Appendix B.2.

HotpotQA. We use AFlow's split of `hard` questions, with 200 examples for search and 800 for testing (Zhang et al., 2025). Each task includes the question and its context. We score the final answer by normalized token-level F1.

MuSiQue-Ans. We sample 200 questions from MuSiQue-Ans v1.0's official training split and 800 from its development split, using seed 42 and a 50/30/20 mix of two-, three-, and four-hop questions. The agent receives the supplied context paragraphs. We score its final answer by the maximum normalized token-level F1 over the canonical answer and its aliases. The reported test subset therefore comes from official development data.

ScienceWorld. We select 20 search and 100 test variations from the benchmark's respective splits, covering six task types: conductivity, melting point, power supply, renewable versus nonrenewable power supply, growing fruit, and mixing chemicals. All methods use the same three built-in simplifications: direct room travel (`teleportAction`), initially open doors (`openDoors`), and self-watering flower pots (`selfWateringFlowerPots`). This common, simplified protocol reduces navigation and maintenance steps; we score the final state with the native ScienceWorld task score, flooring negative scores at zero.

WebShop. We use the full product catalog and a fixed 20/100 split drawn from the benchmark's training and test goals. Beauty, fashion, garden, grocery, and electronics each contribute four search and 20 test goals. The search and test target products are disjoint. We use the benchmark's native purchase reward.

BigCodeBench-Hard. From BigCodeBench-Hard v0.1.4, we exclude tasks with evident inconsistencies in their instructions or evaluation, publicly reported issues, or incompatibilities with our execution environment. We then construct a fixed 20/100 subset spanning diverse code domains. Search and test task IDs are disjoint. A task scores one only if all benchmark unit tests pass, and zero otherwise.

B.2 Baseline Implementations

B.2.1 Question Answering

For HotpotQA, AFlow and IO flatten context sentences, whereas the other methods retain the structured context; MuSiQue-Ans uses the same formatted question for all methods. IO executes the initial Custom workflow supplied with AFlow, while CoT uses AdaptFlow's static chain-of-thought seed. ADAS, AFlow, and AdaptFlow retain their respective proposal and search mechanisms with task-specific answer contracts. ADAS runs 10 generations, and AFlow runs 10 search rounds. Following AdaptFlow's original schedule (Zhu et al., 2025), we run three outer iterations, with up to six inner-loop updates per task cluster in each outer iteration; the inner loop may stop early. Its TTA variant is evaluated separately: it adapts the selected workflow for each test cluster from two sampled, unlabeled test inputs before testing that cluster.

B.2.2 Code Generation and Interactive Tasks

We adapt each baseline to the task format, tools, and submission interface of each benchmark while retaining its core method.

IO and CoT. Both use the same fixed tool loop; CoT adds an explicit step-by-step reasoning instruction.

ReCreate. We follow its native scaffold and experience-memory update procedure: two outer iterations over the 20 search tasks, each with five batches of four tasks. Each batch evaluates the current scaffold on its four tasks, performs local revisions, and synthesizes one global update.

AgentSquare. After evaluating an initial seed, AgentSquare runs 10 search iterations. Each iteration evaluates up to four single-module, four evolved-module, and one recombined candidate, each on all 20 search tasks. To accelerate evaluation, our adaptation processes tasks in batches of up to five. For a memory-enabled candidate at test time, traces from earlier test tasks may thus influence later ones, creating a possible within-test information leakage channel.

Meta-Harness. We run 10 solver-revision iterations, evaluating one new executable solver on all 20 search tasks per iteration after an initial seed evaluation.

B.3 Configuration and Execution Settings

Model settings. All runs use DeepSeek-V4.1-Flash (DeepSeek-AI, 2026) through DeepSeek's official API, with thinking enabled and low reasoning effort.

Search and revision. EvoG starts by evaluating a task-specific seed workflow and then generates one candidate in each of 10 iterations. The Analyzer keeps context across iterations. After the initial edit, the Generator allows at most two repair attempts per candidate when the validation fails.

Selector parameters and ties. For parent selection, Soft-mixed uses a top-k shortlist with $k = 4$ plus the evaluated seed when it falls outside that shortlist, temperature $\tau = 0.02$, and uniform-mixture weight $\lambda = 0.1$. UCT-tree uses exploration coefficient $\beta = 0.02$ and restart probability $\rho = 0.05$, a restart selects the initial workflow. Pareto-coverage uses tie tolerance $\epsilon = 10^{-9}$. All stochastic Selectors use random seed 42; Best-observed has no tunable selector parameter. Their exact rules are defined in Appendix A.

For ties, Best-observed prefers fewer evaluation errors on S, then lower estimated evaluation cost on S, then earlier candidate creation. Soft-mixed retains evaluation order when scores tie in its top-k ranking. UCT-tree treats UCT scores $U_t(c)$ within 10^{-12} of the maximum as tied and samples uniformly among them.

Evaluation example sampling. The fixed select set ranks candidates; the propose set supplies additional revision feedback but does not enter the final ranking. In QA's two $50/50$ settings, the bootstrap propose set is sampled uniformly from the 150 search examples outside the fixed select set. *Random-feedback* redraws all 50 propose examples uniformly in each later round. *Low-score-feedback* instead prioritizes examples from the selected parent's most recent evaluated propose batch whose F1 is below 0.8 on the 0–1 scale, ordering them by score and retaining up to 50; it fills the remaining slots uniformly from examples outside the select set that were not retained. Sampling is without replacement within a round, but examples may recur across rounds.

Execution trace selection. For each selected parent, the Analyzer receives an evaluation summary and a small set of complete execution traces showing its model and tool interactions on individual examples. We rank examples by their benchmark score and choose traces from both ends, so the Analyzer can inspect relatively strong and weak outcomes. These are relative score groups, not necessarily successes and failures. The target is 10 traces from each end for QA's $200/0$ setting, five from each end for QA's $100/0$ and $50/50$ settings, and four from each end for code generation and interactive tasks. If one end has too few eligible examples, we fill its remaining slots from the other end without duplicating examples.

Execution limits. Across EvoG and the baselines, QA solvers have a 3,600-second limit per question, and final evaluations on code generation and interactive tasks have a 600-second limit per task. Timed-out questions and tasks receive zero. Task-solving model requests specify at most 32,768 output tokens with a 120-second timeout. For QA, all methods allow at most 64 model calls and 256 tool calls per question. For code generation and interactive tasks, all methods allow at most 128 model calls and 512 tool calls per task. EvoG also sets the LangGraph recursion limit to 256.

C Workflow Evolution Case Study

We illustrate how execution evidence and prior outcomes inform program revision using a WebShop run with soft-mixed selection and full-pool sampling. The workflow separates requirement extraction, product research, and purchase. Table 4 follows three consecutive candidates evaluated on the same 20-example select set.

Execution evidence. WebShop task `4099` requests purple massage-table sheets with a size of 70×185 cm. Candidate 2 extracts both requirements, but its research result treats a size mentioned in the product title as satisfied even though the page exposes only a color selector. The structured result lists no unmet requirements after five of the six allowed searches. The purchase stage then follows

the supplied plan, selecting only purple. The trace therefore locates the missing size selection in product research and its stopping decision, before purchase.

Historical comparison and diagnosis. The Analyzer compares this outcome with Candidate 1, which had selected both size and color on the same task and received full reward. This earlier result establishes that a product offering both options had been reachable. Although Candidate 2 improves the aggregate select score, it loses this previous success. The resulting diagnosis targets premature acceptance of partially matching products while retaining the existing purchase procedure.

Program revision. Candidate 3 adds a structured table of inspected products to the research output, recording option coverage, literal attribute matches, price, and unmet requirements. Its research prompt ranks products by these fields and requires further queries based on missing options or attributes before returning a partial match. The edit also raises the search cap from six to eight and the research model-call limit from 30 to 40. These changes modify the intermediate schema, decision instructions, and execution limits within the same program.

Table 4: Observed select-set outcomes in the WebShop case study. Rewards are scaled to 0–100; full success denotes reward 100. Each candidate is evaluated on the same 20 examples.

Candidate	Mean reward	Full success (%)	Task `4099` reward
1	85.58	55	100.00
2	85.85	60	62.50
3	89.15	75	100.00

Observed outcome. Candidate 3 purchases a product with both the requested size and color on task `4099`, recovering full reward. Across the select set, mean reward rises from 85.85 to 89.15 and fully successful purchases increase from 12 to 15. This sequence makes the feedback loop concrete: traces locate a failing intermediate decision, history supplies a prior successful outcome for comparison, and the revision turns that diagnosis into explicit product comparisons and continued search.

D Supplementary Results

Search cost by phase. Figures 6 and 7 separate the estimated model-call cost of optimization from candidate evaluation. Optimization covers calls used to propose and revise workflows, including analysis and review; evaluation covers calls made when executing candidates on search examples. We use the same cost estimates as the main-text search trajectories. In the QA figures, Best, Soft, UCT, and Pareto denote best-observed, soft-mixed, UCT-tree, and Pareto-coverage selection; Full, 100, Rand, and Low denote full-pool $(200/0)$, fixed-subset $(100/0)$, random-feedback $(50/50)$, and low-score-feedback $(50/50)$ sampling, respectively. On BigCodeBench-Hard, AgentSquare spends 0.224 USD on optimization and 14.106 USD on evaluation, compared with 0.271 USD and 1.496 USD for EvoG soft-mixed. Candidate evaluation therefore accounts for most of this search-cost gap.

Feedback coverage and repetition. Figure 8 compares random-feedback and low-score-feedback under each Selector. Both evaluate 50 feedback examples per round from the 150 examples outside the fixed select set. Over the ten revision rounds, excluding initialization, we count the distinct feedback examples $|\bigcup_{t=1}^{10} F_t|$ and compute the mean adjacent-round overlap $\frac{1}{9}\sum_{t=2}^{10} |F_t \cap F_{t-1}|/50$. Low-score-feedback covers fewer distinct examples and repeats more examples between consecutive rounds in all eight matched comparisons. The Sampler thus changes the breadth and recurrence of revision feedback at the same per-round evaluation count.

Token use and execution time. Figures 9–14 report model token use and test execution time for the evaluated configurations. Token stacks separate input, cached input, and output. The three labels above each bar give its input, cached-input, and output counts, from bottom to top, in K/M units. The search figures include only methods with a search run; search totals combine model calls for

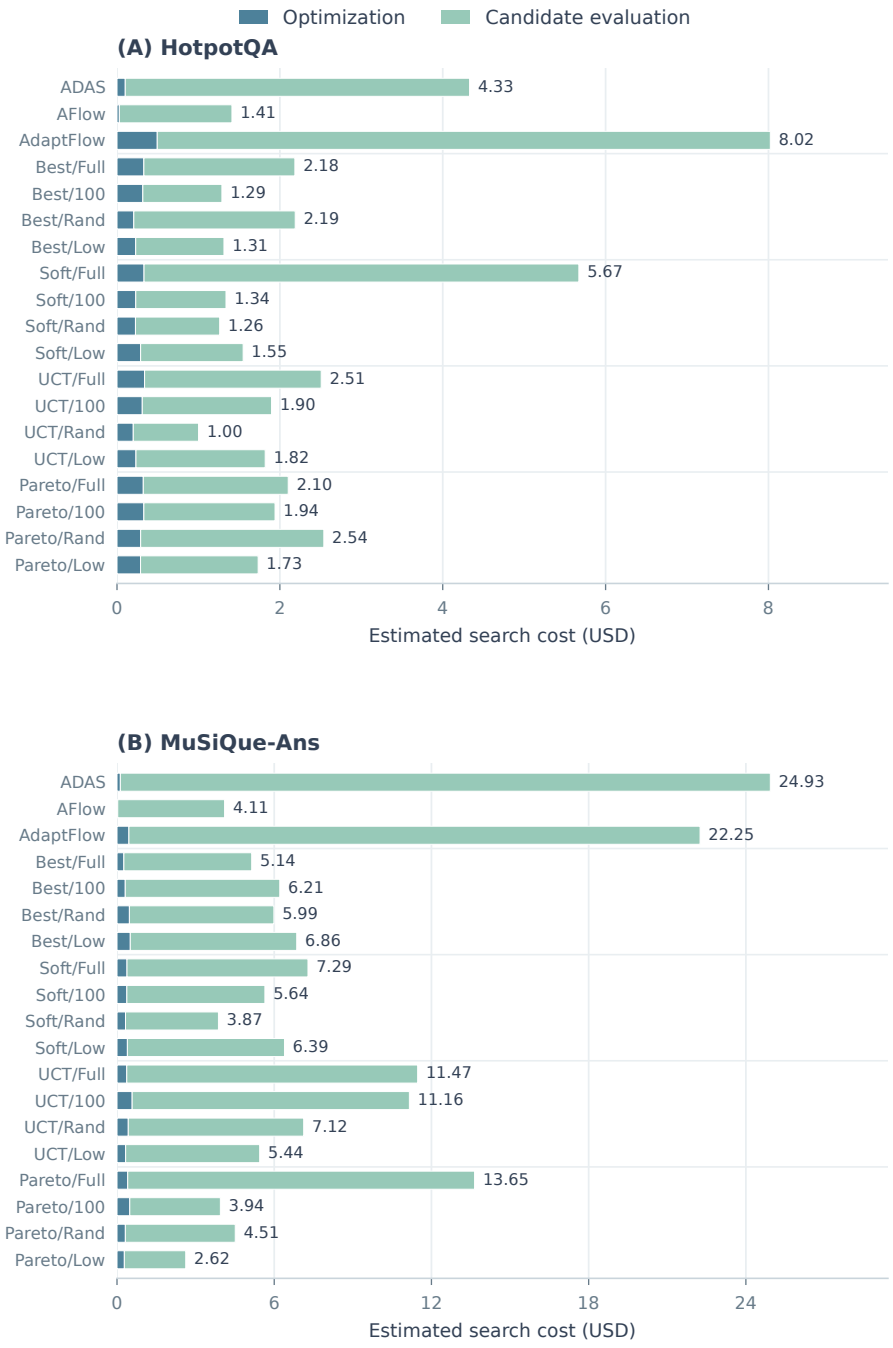


Figure 6: Estimated model-call cost of one QA search run, split into optimization and candidate evaluation: (A) HotpotQA and (B) MuSiQue-Ans. Each bar includes both phases; labels show total cost in USD.

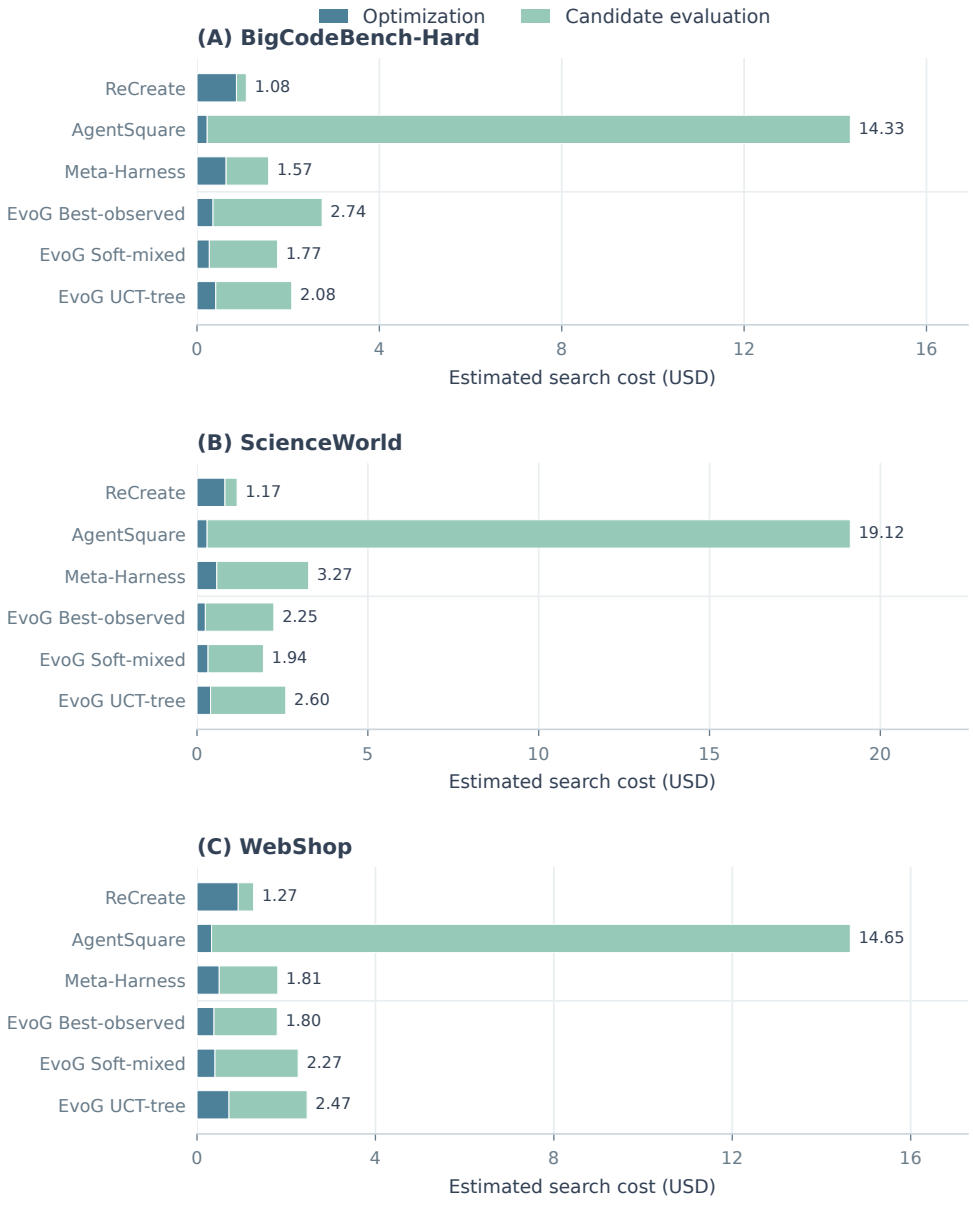


Figure 7: Estimated model-call cost of one search run on code generation and interactive tasks, split into optimization and candidate evaluation: (A) BigCodeBench-Hard, (B) ScienceWorld, and (C) WebShop. Labels show total cost in USD. Phase totals use the same accounting as the main-text search curves.

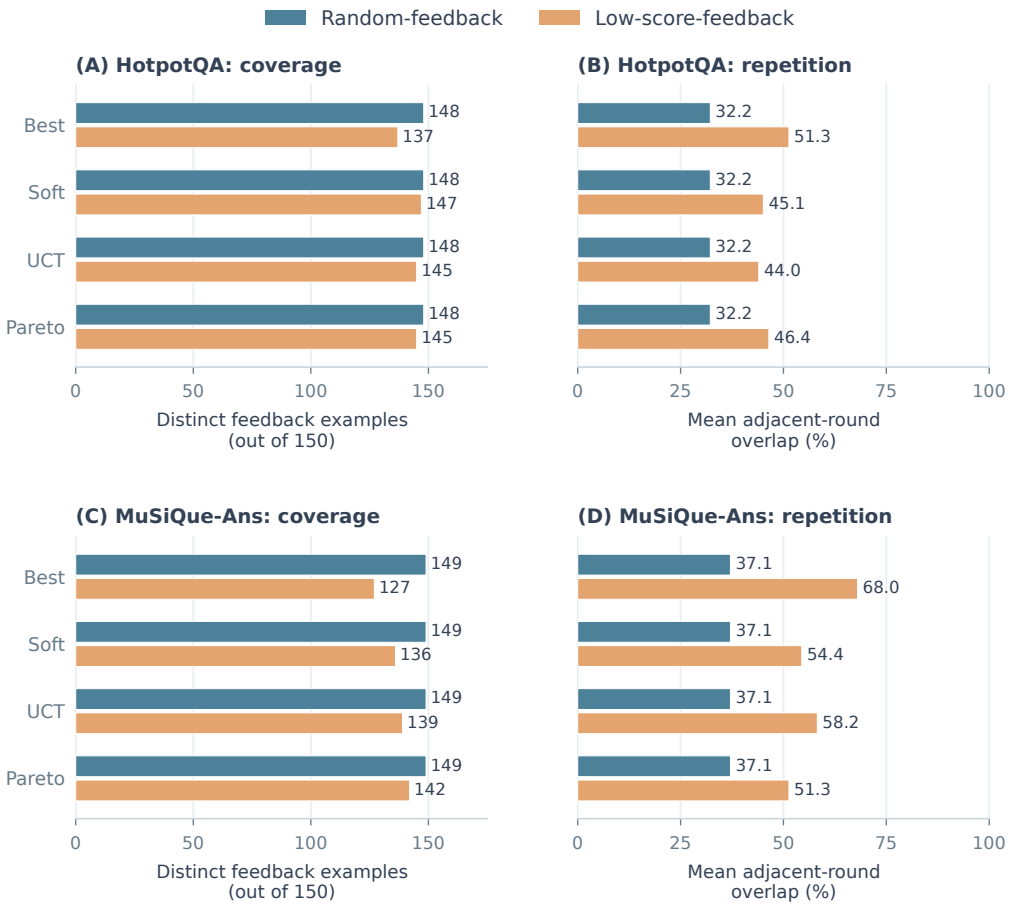


Figure 8: Feedback coverage and adjacent-round repetition over ten revision rounds, excluding initialization. The left panels count distinct feedback examples out of 150; the right panels show the mean percentage of a round's 50 feedback examples also evaluated in the preceding round. Both Samplers use $|S|/|F_t| = 50/50$. Random-feedback uses the same seeded feedback sequence across Selectors within each benchmark.

optimization and search evaluation in one run. Test totals cover three executions of the test set; the AdaptFlow + TTA bar includes both adaptation and test execution. Wall time measures the elapsed time from the start to the end of each recorded test-item execution, averaged across the three repetitions; adaptation before execution is excluded. It includes model-response and tool latency, so the reported values depend on service responsiveness and serve as a reference for these runs.



Figure 9: Model tokens used by one QA search run: (A) HotpotQA and (B) MuSiQue-Ans. Each stacked bar combines optimization and search evaluation. Labels above each bar show the three token components separately in K/M units.



Figure 10: Model tokens used by one search run on code generation and interactive tasks: (A) BigCodeBench-Hard, (B) ScienceWorld, and (C) WebShop. Each stacked bar combines optimization and search evaluation; labels above each bar give the three token components separately in K/M units.

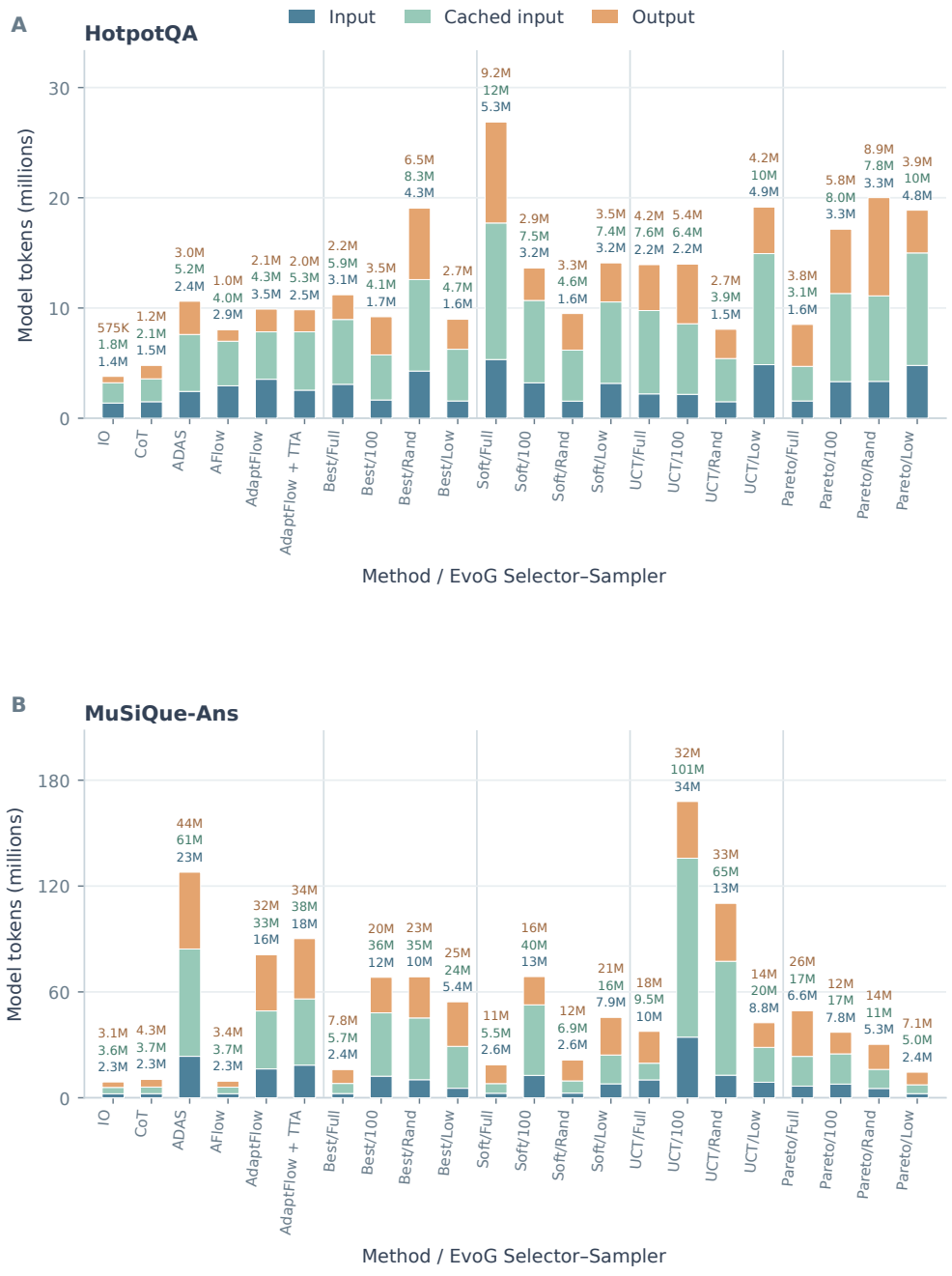


Figure 11: Model tokens over three QA test executions. The AdaptFlow + TTA bar includes its test-time adaptation and execution in one stack. Labels above each bar give the three token components in K/M units.



Figure 12: Model tokens over three test executions on the code generation and interactive task benchmarks. Stack segments show input, cached input, and output; labels above each bar give their counts in K/M units.

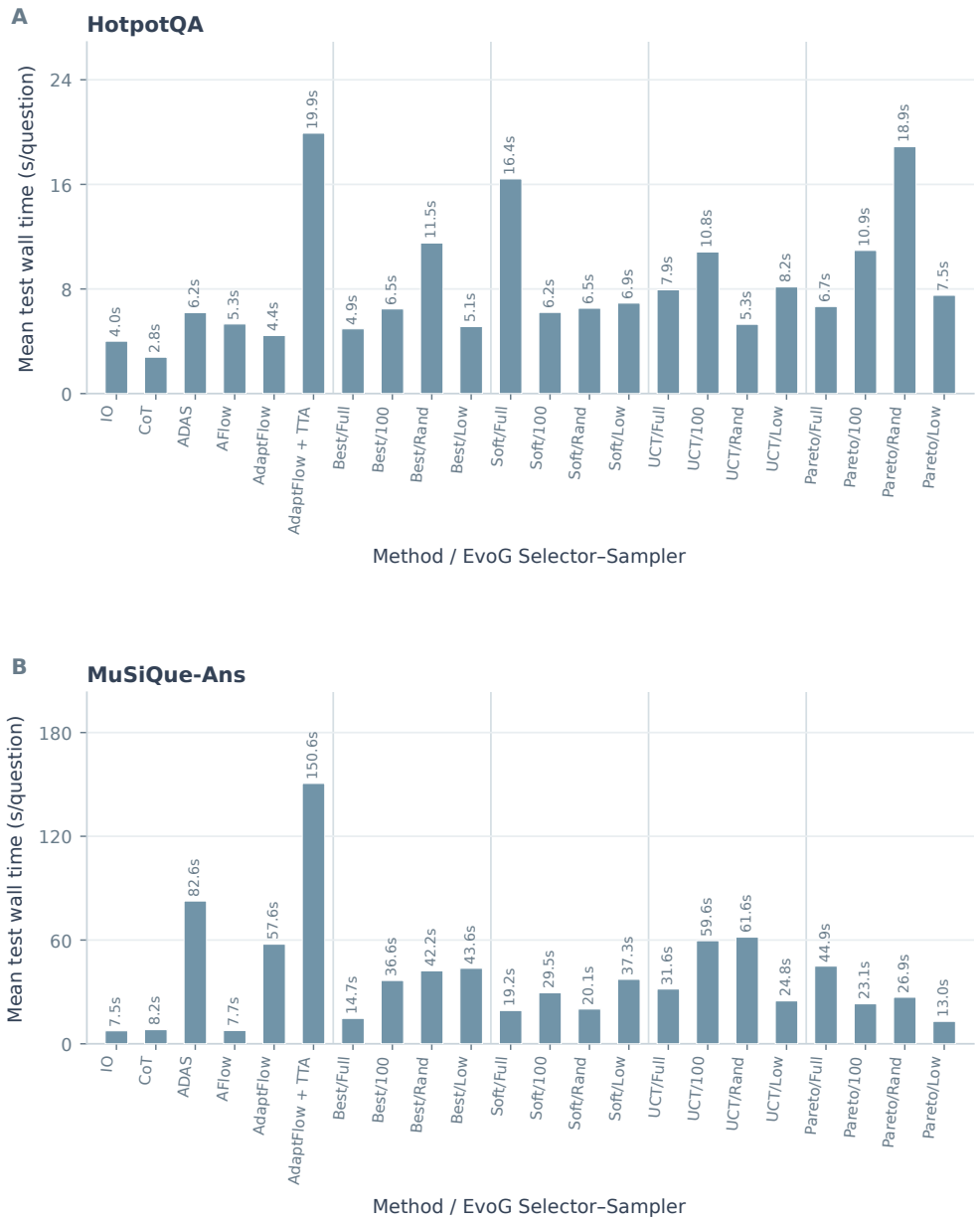


Figure 13: Mean test execution wall time per question for each QA configuration, averaged over the three test repetitions. Vertical labels give seconds per question.

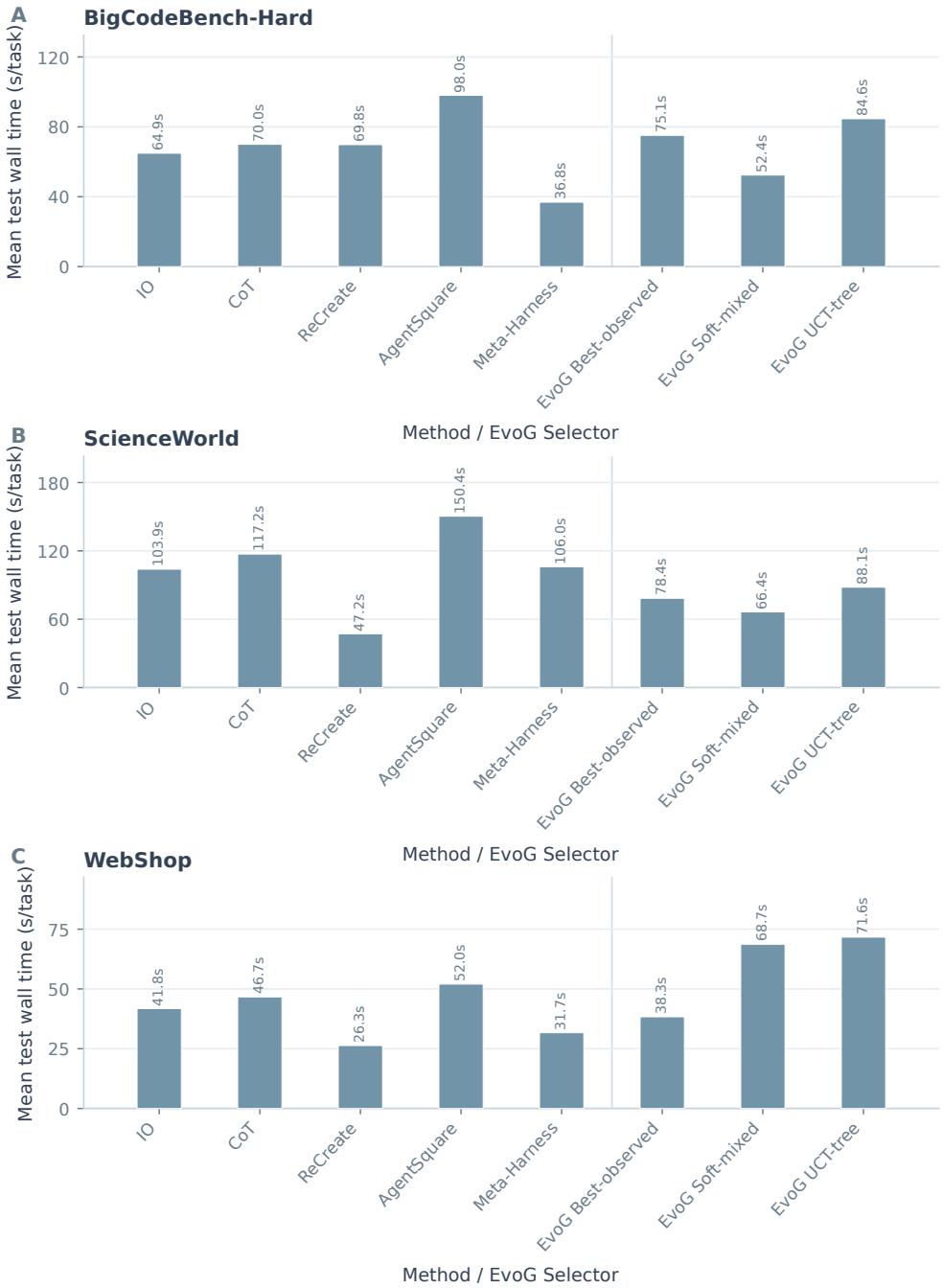


Figure 14: Mean test execution wall time per task for each configuration on code generation and interactive tasks, averaged over the three test repetitions. Vertical labels give seconds per task.