RSI-ROUTER: COST-EFFICIENT LLM ROUTING VIA EVOLUTION OF SUBTASK-LEVEL MODEL ASSIGNMENTS WITH EXECUTION SKILLS

Anonymous authors
Paper under double-blind review

ABSTRACT

Practical deployment of large language model (LLM) agents requires strong task performance at affordable inference cost. For long-horizon agentic tasks, this performance–cost trade-off can be improved through within-task large–small model collaboration, as smaller models can handle some stages even when they cannot solve the full task. In this paper, we introduce *RSI-router*, a routing framework that constructs subtask-level model assignments and model-specific execution skills through recursive self-improvement over execution experience. Each iteration consists of four stages: *Subtask Mining* derives subtask definitions and identification rules from training trajectories; *Routing Strategy Evolution* proposes and evaluates diverse model assignments; *Model-Specific Skill Evolution* compares routed and large-model-only trajectories to diagnose failures and develop reusable execution skills; and *Pareto-Optimal Router Selection* updates the Pareto population from historical and new routers while retaining dominated routers for reuse. Across five agentic benchmarks, RSI-router reduces inference cost by **51.7%** on average and up to **82.2%** while outperforming the large-model baseline. It also yields a stronger performance–cost Pareto frontier than existing routing baselines. Further analyses show that RSI-router progressively reduces cost over iterations without performance loss, building on previously accumulated skills.

1 INTRODUCTION

Practical deployment of large language model (LLM) agents requires a balance between strong task performance and cost-efficient inference, especially for complex agentic tasks such as software development (Liu et al., 2024), personal assistance (Sager et al., 2026), and scientific research (Lu et al., 2026). These tasks often unfold over long interaction sequences and require repeated LLM calls, making exclusive reliance on advanced, larger models costly despite their strong capabilities (Gao & Peng, 2026). As smaller LLMs become increasingly capable (Qwen Team, 2026; Abouelenin et al., 2025), large–small model collaboration offers a promising way to improve this performance–cost trade-off by delegating suitable parts of a task to smaller models while reserving larger models for parts that require stronger capabilities (Alazraki et al., 2026).

LLM routing (Hu et al., 2024) enables such collaboration by adaptively selecting which model to use. Task-level methods (Jiang et al., 2023; Ding et al., 2024; Jitkrittum et al., 2026) typically assign an entire request to a single model. However, agentic tasks often contain stages that smaller models can handle effectively, even when they cannot complete the full task. Assigning one model throughout therefore misses opportunities to reduce cost through within-task collaboration. Step-level methods address this limitation by selecting a model at each interaction step, but learning effective routing decisions remains challenging. Some methods (Zhang et al., 2026e; Wang et al., 2026) learn from pre-collected trajectories and are constrained by the coverage of those trajectories. Others explore model assignment through reinforcement learning (Zhang et al., 2025a; 2026a), but can face costly stepwise exploration without task-structure guidance and difficult credit assignment from final-outcome feedback.

To address these challenges, we introduce *RSI-router*, a routing framework that recursively improves its routing strategies and execution skills using execution outcomes and accumulated experience.

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.

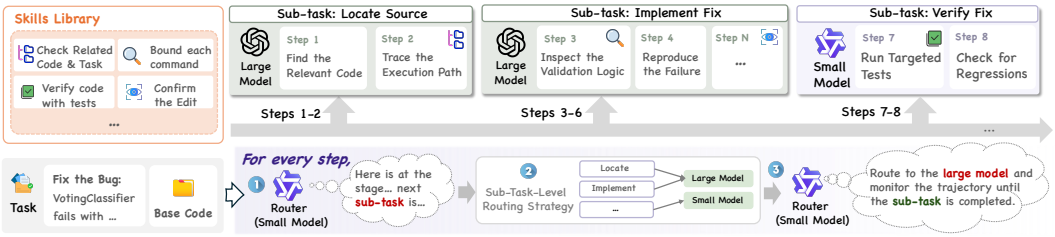

Figure 1: **Illustration of RSI-router execution.** At each step, a small model serves as the router, identifying the upcoming subtask from the routing context. The router selects a model for the subtask and appends the corresponding model-specific skills to the selected model's input. In this SWE-bench example, the large model locates relevant code and implements the fix, while the small model verifies it.

RSI-router is motivated by three observations. First, LLMs typically possess prior knowledge of task objectives and procedures, which can organize the exploration of model assignments around reusable subtasks rather than treating every interaction step as an independent decision. Second, execution trajectories reveal where a particular model fails, succeeds, or performs redundant actions, providing concrete evidence for revising model assignments and developing model-specific execution guidance. Third, such guidance can itself change what a model can reliably accomplish: a smaller model that initially fails on a subtask may become effective when equipped with an appropriate execution skill. Consequently, model assignments and execution skills are interdependent and need to be improved jointly rather than optimized in isolation.

Specifically, RSI-router recursively improves the routing system through a four-stage evolution loop. **Subtask Mining** derives and revises subtask identification from trajectories. **Routing Strategy Evolution** proposes diverse routing strategies over these subtasks in parallel and evaluates candidate routers with selectively inherited skills. **Model-Specific Skill Evolution** compares candidate and large-model-only trajectories on the same tasks to diagnose model-specific failure patterns and redundant actions, then refines or generates execution skills tailored to subtask–model pairs. **Pareto-Optimal Router Selection** compares historical and newly generated routers by performance and cost, updating the Pareto population while retaining dominated routers as negative examples for subsequent evolution. The resulting routers, skills, and execution experience further guide subsequent iterations. While a large LLM drives this evolution process, at inference time, a small model serves as the router, identifying the upcoming subtask, selecting its assigned model, and appending the corresponding execution skills to the selected model's input (Figure 1).

We evaluate RSI-router on five benchmarks of agentic tasks: ALFWorld (Shridhar et al., 2021), ScienceWorld (Wang et al., 2022), WebShop (Yao et al., 2022), SWE-bench Verified (Jimenez et al., 2024; Chowdhury et al., 2024), and Terminal-Bench 2.0 (Merrill et al., 2026). Across these benchmarks, RSI-router reduces inference cost by **8.1–82.2%**, with an average reduction of **51.7%**, while matching or exceeding large-model-only performance. For example, it improves ALFWorld success rate from 90.62% to 98.44% with 75.3% lower cost, and Terminal-Bench 2.0 from 40.00% to 46.67% with 18.0% lower cost. RSI-router also improves the performance–cost Pareto frontier over the evaluated task-level and step-level routing baselines. Further analysis shows repeated cost reductions over iterations and continued reuse of skills acquired in earlier iterations, while ablation studies further corroborate key routing and execution design choices.

Our main contributions are as follows:

- **A recursive self-improvement paradigm for LLM routing.** We introduce *RSI-router*, which constructs cost-efficient routing strategies through iterative improvement over execution outcomes and accumulated experience.
- **Subtask-level routing with model-specific execution skills.** RSI-router discovers reusable subtasks, evolves model assignments at the subtask level, and develops model-specific execution skills, enabling smaller models to reliably handle suitable stages of long-horizon agentic tasks.
- **Advancing the Pareto frontier across five benchmarks.** Across five agentic benchmarks, RSI-router matches or exceeds the performance of the large-model baseline while reducing inference

cost by 51.7% on average, and consistently extends the performance–cost Pareto frontier beyond existing routing baselines.

2 METHOD

Agentic tasks comprise stages with different capability requirements, and the most cost-effective model may vary across stages. We group execution steps with similar objectives into subtasks to explore model assignments and reuse experience. Evaluating these assignments yields trajectories revealing model-specific failures and redundant actions, guiding the development of execution skills. As these skills change models' effective capabilities, model assignments need to be revisited. We therefore propose *RSI-router*, a routing framework that refines subtask definitions, model assignments, and execution guidance through iterative evolution to improve performance–cost trade-offs.

Starting with training trajectories from large-model-only and small-model-only executions, RSI-router evolves a population of routers through a four-stage loop guided by a large-model proposal agent (Figure 2): **Subtask Mining** organizes execution experience into reusable subtasks to guide model assignments; **Routing Strategy Evolution** explores subtask-level model assignments to generate candidate routers with different performance–cost trade-offs; **Model-Specific Skill Evolution** distills the resulting training trajectories into reusable execution skills to extend models' effective capabilities; and **Pareto-Optimal Router Selection** guards against regressions by comparing historical and new routers using validation performance and cost.

2.1 ROUTING AND EVOLUTION SETUP

We consider agentic task execution using a large model L and a small model S, supported by reusable execution skills. To explore how the two models should collaborate, we evolve a population of routers with different model assignments and execution skills. At iteration t, newly proposed candidates share a subtask set Z_t, with definitions describing execution objectives and identification rules for predicting the upcoming subtask. Each candidate router r_i specifies which model executes each subtask; for example, L may implement a fix while S verifies it. Each skill contains execution instructions and specifies the applicable subtasks and execution model.

Before each execution step, the small model S predicts the upcoming step's subtask using the definitions and identification rules in Z_t and the routing context c. This context contains the user query, the interaction history from the previous few steps, and the previous subtask prediction. The router then selects the assigned model, appends only the skill text matching the predicted subtask and selected model to its input, and invokes it to generate the response.

Router evolution overview. We run router evolution for G iterations to maximize task performance and minimize inference cost. We maintain a Pareto population $\mathcal{P}_t$ of non-dominated routers and an archive $\mathcal{F}_t$ of dominated routers retained as negative examples for subsequent refinement, initialized as $\mathcal{P}_0 = \{r_L, r_S\}$ and $\mathcal{F}_0 = \emptyset$. The baselines r_L and r_S use L and S, respectively, throughout execution without skills. Skills from historical routers in $\mathcal{P}_t \cup \mathcal{F}_t$ form a shared library for reuse. Each router r is evaluated on disjoint training and validation sets to assess whether training-derived refinements remain effective on other tasks. Aggregate performance and cost from both sets guide evolution, while execution trajectories $\mathcal{T}(r)$ and per-task feedback are provided only for training tasks. Validation performance and cost guide Pareto-based population updates, yielding the final population $\mathcal{P}_G$.

2.2 ROUTER EVOLUTION

Subtask Mining. To construct Z_t, we identify subtasks in training trajectories and derive their definitions and identification rules. For example, code repair may involve locating relevant code, implementing a fix, and verifying the result, with the same subtask recurring within a trajectory.

In iteration t, the proposal agent first analyzes these trajectories and partitions them into segments by execution objective. It then defines and clusters subtasks across trajectories to form Z_t, assigning subtask labels to segments and deriving identification rules. From the second iteration onward, the agent revisits previously defined subtasks, retaining, splitting, or merging them and updating the corresponding definitions and identification rules.

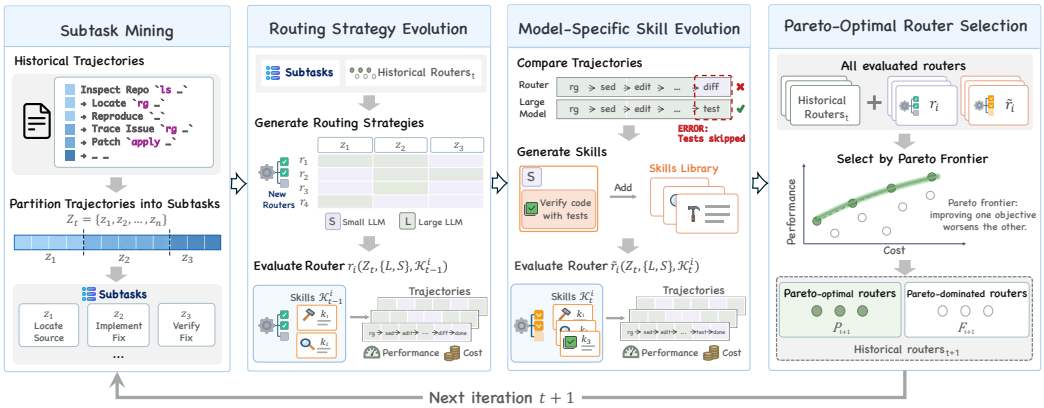

Figure 2: **Router evolution in RSI-router.** A proposal agent iteratively mines subtasks, explores model assignments with selectively inherited skills, and evolves model-specific skills by comparing training trajectories with large-model-only executions. Pareto-optimal and dominated routers are both retained to guide subsequent iterations.

To check whether the resulting subtasks can be identified before execution, we replay sampled training trajectories. At each step, the small model S predicts the subtask label from the pre-execution context c using the current definitions and identification rules. The proposal agent compares these predictions with the annotated labels and refines ambiguous definitions or rules. The finalized Z_t remains fixed for the rest of the iteration.

Routing Strategy Evolution. With Z_t established, we explore model assignments with different performance–cost trade-offs. In each iteration, the proposal agent generates N diverse routing strategies in parallel, indexed by $i = 1, \ldots, N$. It draws on the model assignments, associated subtask definitions, and aggregate training and validation performance and cost of historical routers to explore alternative assignments.

For the i-th strategy, the proposal agent selectively inherits skills from the shared library based on their relevance to the current subtask definitions and model assignments. It reassesses applicability after subtask splits or merges, selecting zero or more skills per subtask to form $\mathcal{K}^i_{t-1}$. Each candidate router is evaluated on the training and validation sets to measure performance and cost. Its training trajectories and per-task feedback are passed to the next stage to guide model-specific skill evolution.

Model-Specific Skill Evolution. Execution skills complement model assignments by providing reusable, model-specific guidance to help models complete their assigned subtasks reliably and avoid redundant actions. As subtask definitions and model assignments evolve, newly collected trajectories reveal where this guidance needs further refinement. For each candidate router r_i, the proposal agent compares its training trajectories with those of r_L on the same tasks, using per-task feedback and cost to identify model-specific failure patterns and redundant actions.

Based on this diagnosis, the agent refines inherited skills or generates new ones for the relevant subtasks and assigned models, updating $\mathcal{K}^i_{t-1}$ to $\mathcal{K}^i_t$. We denote the updated router by $\tilde{r}_i$ and re-evaluate it on the training and validation sets, keeping Z_t and the model assignments unchanged.

Pareto-Optimal Router Selection. To guide evolution towards higher task performance and lower inference cost while guarding against regressions, we jointly compare historical routers in $\mathcal{P}_t \cup \mathcal{F}_t$ and candidates before and after skill evolution using validation performance and cost. The non-dominated routers form $\mathcal{P}_{t+1}$, while the remaining routers are archived in $\mathcal{F}_{t+1}$. Routers in both sets remain available for proposing routing strategies and inheriting execution skills in subsequent iterations. After G iterations, RSI-router returns the final Pareto population $\mathcal{P}_G$.

3 EXPERIMENTS

We evaluate RSI-router on five benchmarks of agentic tasks to examine whether it can reduce online cost without sacrificing task performance. We further analyze the router evolution process and assess key design choices through ablation studies.

3.1 SETUP

Compared methods. We compare against the large-model-only router r_L, the small-model-only router r_S, and random routers that select either model with equal probability at the task or step level. The task-level routing methods include HybridLLM (Ding et al., 2024), FrugalGPT (Chen et al., 2024), RouteLLM (Ong et al., 2025), GraphRouter (Feng et al., 2025), and Avengers-Pro (Zhang et al., 2025b). We also include Router-R1 (Zhang et al., 2025a) and MTRouter (Zhang et al., 2026e) as step-level routing baselines (see Appendix A for details).

Benchmarks. We use ALFWorld (Shridhar et al., 2021), ScienceWorld (Wang et al., 2022), WebShop (Yao et al., 2022), SWE-bench Verified (Jimenez et al., 2024; Chowdhury et al., 2024), and Terminal-Bench 2.0 (Merrill et al., 2026), covering household environment interaction, scientific experimentation, online shopping, code repository repair, and terminal operations, respectively. The full benchmark prompts and tool definitions are provided in Appendix B.

Models and runtime settings. Across all five benchmarks, we use DeepSeek-V4.1-Flash (DeepSeek-AI, 2026) as the large model L and Qwen3.5-9B (Qwen Team, 2026) as the small model S. Our execution harness is built on mini-SWE-agent (Yang et al., 2024), with environment interactions adapted to each benchmark. We use Codex (OpenAI, 2025) as the proposal agent and prompt S for subtask identification using the interaction history from the previous 3 steps.

Data splits and iteration protocol. We use fixed, disjoint training, validation, and test sets ($\mathcal{D}_{\mathrm{tr}}$, $\mathcal{D}_{\mathrm{val}}$, $\mathcal{D}_{\mathrm{test}}$), with 64 tasks per set for ALFWorld, ScienceWorld, and WebShop, 32 per set for SWE-bench Verified, and 32/32/25 tasks for Terminal-Bench 2.0. Router evolution uses aggregate performance and cost from training and validation, but trajectories and per-task feedback only from training. Validation metrics guide candidate router generation and Pareto population updates. After router evolution, we evaluate all routers in the final Pareto population $\mathcal{P}_G$ on $\mathcal{D}_{\mathrm{test}}$, which provides no feedback for router evolution. We run router evolution for $G = 6$ iterations with $N = 4$ candidate routing strategies per iteration.

Evaluation metrics. We report task success rate (%) on all benchmarks except WebShop, where we use reward (0–1). Costs are reported in USD using the official DeepSeek-V4.1-Flash input and output prices. We estimate the corresponding small-model prices using a fixed 1:30 reference ratio informed by GPU-hour measurements on H200 GPUs (see Appendix C for details). Costs include all test-time LLM calls and tokens, including subtask routing and skill context. All results are averaged over three evaluation repeats per test task. For learned baselines, we additionally average over three training seeds.

3.2 MAIN RESULTS

RSI-router matches or exceeds the large-model baseline at substantially lower inference cost. Table 1 compares RSI-router with single-model, task-level, and step-level baselines across five benchmarks. Since some methods may produce multiple routing strategies, we report, for each method, the minimum inference cost among strategies that achieve *at least* the performance of the large-model baseline (DeepSeek-V4.1-Flash), ensuring a fair comparison under the same performance requirement. RSI-router exceeds DeepSeek-V4.1-Flash in performance across all five benchmarks, while reducing inference cost by 8.1–82.2%. On ALFWorld, success rate increases from 90.62% to 98.44% with a 75.3% cost reduction. Across ALFWorld, ScienceWorld, and WebShop, RSI-router surpasses the large model at around one quarter of its cost or less, demonstrating that large–small model collaboration can improve performance and cost efficiency simultaneously.

RSI-router occupies a substantial portion of the performance–cost Pareto frontier across five benchmarks. While Table 1 compares the inference cost of different methods under a common

performance constraint, Figure 3 provides a more comprehensive view of the performance–cost trade-off by plotting all evaluated candidate strategies for each method. RSI-router improves the performance–cost Pareto frontier, with several configurations matching or outperforming the large model at lower cost. By contrast, the step-level baselines Router-R1 and MTRouter reduce cost relative to the large model but remain below its performance on all five benchmarks under the evaluated settings (Appendix A).

Table 1: **RSI-router matches or exceeds the large-model baseline at substantially lower inference cost.** Cost is total test-set inference cost (USD). Δ denotes relative percentage changes from the indicated baseline.

Method	ALFWorld		ScienceWorld		WebShop		SWE-bench Verified		Terminal-Bench 2.0	
	Perf. ↑	Cost ↓	Perf. ↑	Cost ↓	Perf. ↑	Cost ↓	Perf. ↑	Cost ↓	Perf. ↑	Cost ↓
Single-model baselines										
DeepSeek-V4.1-Flash	90.62	1.50	34.38	1.91	0.59	0.75	83.33	3.46	40.00	17.10
Qwen3.5-9B	65.62	0.13	14.58	0.06	0.48	0.02	62.50	0.69	5.33	0.17
Task-level routing										
Random	79.69	0.82	26.56	0.72	0.55	0.43	64.58	1.23	26.67	8.22
HybridLLM	68.92	0.46	25.69	0.44	0.51	0.14	69.79	2.33	24.00	8.62
FrugalGPT	86.81	1.21	31.42	1.53	0.57	0.69	68.40	2.07	21.78	9.52
RouteLLM	84.38	1.16	32.99	1.55	0.59	0.67	72.92	2.68	40.00	17.10
GraphRouter	90.62	1.50	34.38	1.91	0.59	0.75	83.33	3.46	40.00	17.10
Avengers-Pro	90.62	1.50	33.33	1.71	0.59	0.75	79.17	3.33	40.00	17.10
Step-level routing										
Random	84.38	1.24	20.83	0.72	0.53	0.34	83.33	3.89	28.00	6.87
Router-R1	72.92	0.19	20.31	0.11	0.49	0.23	55.21	0.28	10.67	0.31
MTRouter	85.94	1.06	31.08	1.00	0.54	0.41	71.88	2.38	24.89	5.85
RSI-router	**98.44**	**0.37**	**35.94**	**0.34**	**0.61**	**0.19**	**84.38**	**3.18**	**46.67**	**14.02**
Δ vs DeepSeek	**+8.6%**	*−75.3%*	**+4.5%**	*−82.2%*	**+3.4%**	*−74.7%*	**+1.3%**	*−8.1%*	**+16.7%**	*−18.0%*
Δ vs MTRouter	**+14.5%**	*−65.1%*	**+15.6%**	*−66.0%*	**+13.0%**	*−53.7%*	**+17.4%**	*+33.6%*	**+87.5%**	*+139.7%*

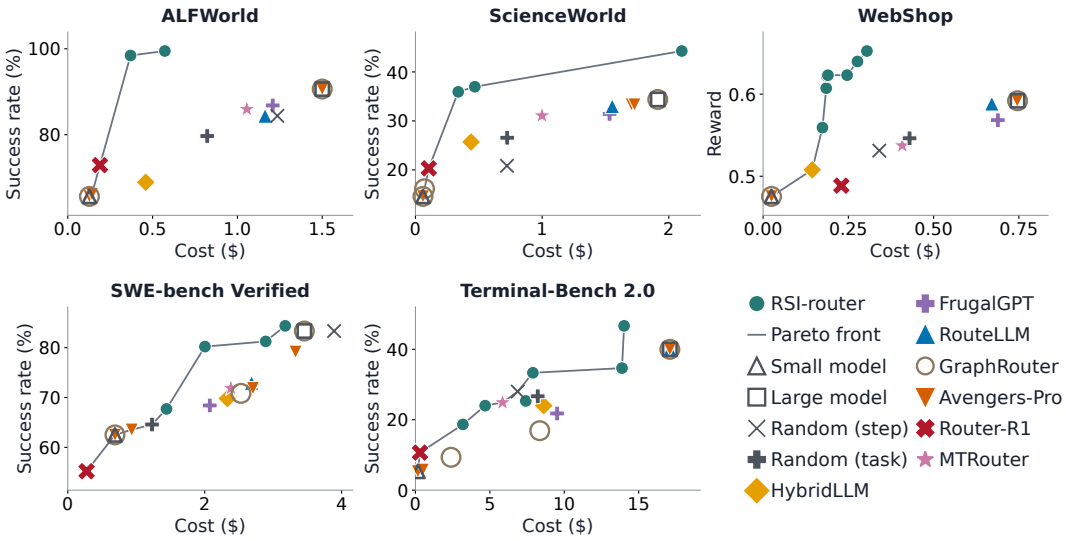

Figure 3: **RSI-router occupies a substantial portion of the performance–cost Pareto frontier across five benchmarks.** Points show test performance and total inference cost (USD).

Several task-level baselines exhibit routing collapse and abrupt transitions. In the evaluated settings, some baselines assign all tasks to either the small or the large model. As performance–cost preferences vary, these baselines switch abruptly between small-model-only and large-model-only routing, with no intermediate trade-offs observed. This pattern is consistent with whole-task selection overlooking the small model's ability to handle individual subtasks within tasks it cannot complete alone. In contrast, RSI-router enables the small model to contribute to tasks it cannot complete independently, better exploiting complementary model capabilities.

RSI-router achieves repeated cost reductions across iterations. Figure 4 tracks the lowest validation cost found through each iteration among routers retaining at least 95% of the large-model baseline's performance. All five benchmarks show further cost reductions after the first iteration, with multiple reductions on four benchmarks. On WebShop, for example, cost savings increase from 28.5% after the first iteration to 65.0%, 68.4%, and 72.5% in subsequent iterations. These successive improvements demonstrate that continued router evolution discovers increasingly cost-efficient routers under the same performance constraint (see Appendix D for representative case studies).

RSI-router accumulates reusable execution skills across iterations. Figure 5 tracks execution skills across the four candidate routers produced by Model-Specific Skill Evolution in each iteration. As new skills are added, skills acquired in earlier iterations continue to appear in later candidates. On ScienceWorld, for example, 22 of the 48 unique skill sources selected across the final iteration's candidates originate from earlier iterations. These results show that router evolution builds a skill library whose accumulated experience is reused in subsequent router construction.

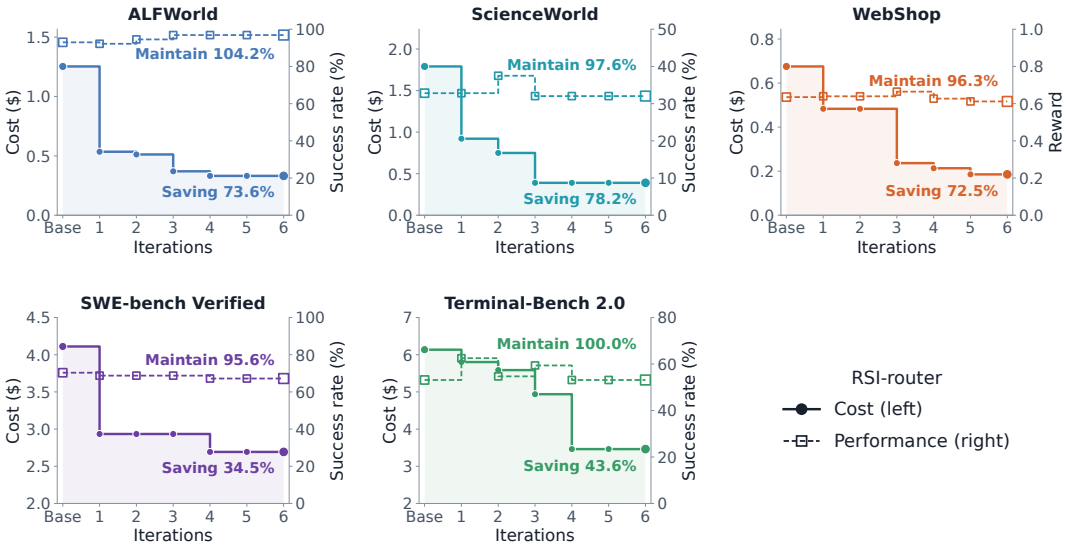

Figure 4: **RSI-router progressively reduces cost during router evolution.** Solid curves show the lowest validation cost found through each iteration under a 95% performance-retention constraint; dashed curves show the corresponding performance. Saving and Maintain report final cost reduction and performance retention relative to DeepSeek-V4.1-Flash.

3.3 Ablation Studies

We conduct ablation studies to assess whether additional interaction history helps subtask identification and whether prompting the small model to select the execution model directly improves performance over step-level random routing. We also remove execution skills from routers that contain them to assess their contribution to performance–cost trade-offs.

Longer interaction history does not improve performance. For the routers reported in Table 1, we increase the interaction history from 3 steps to 4, 8, 16, or all previous steps. As shown in Figure 6(a),

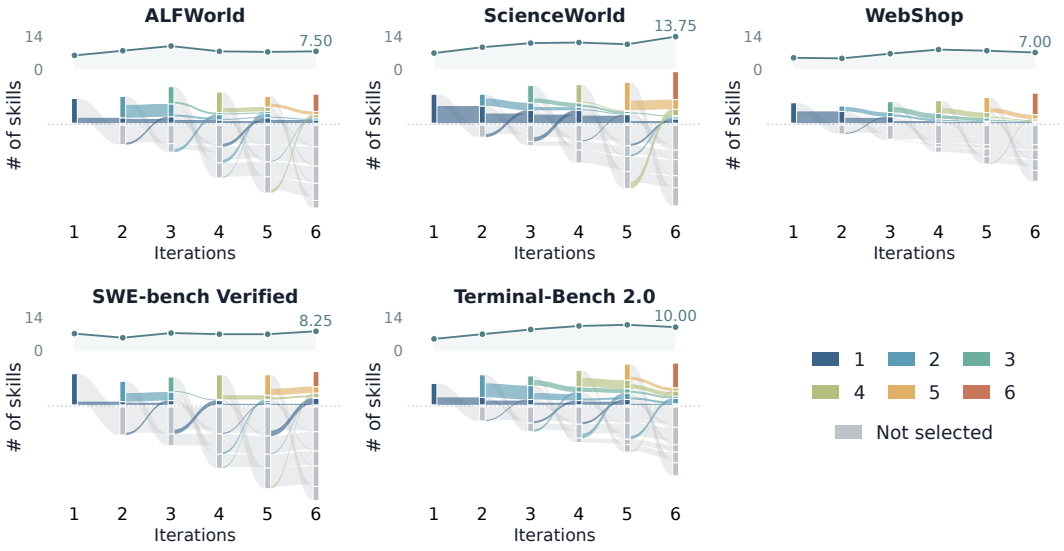

Figure 5: **RSI-router accumulates new skills while reusing experience across iterations.** Upper curves show the mean number of execution skills per candidate router. Lower flows track unique skill sources across iterations.

performance does not improve with longer interaction histories across the five benchmarks. This suggests that short histories are sufficient for effective routing.

Direct difficulty-based routing underperforms step-level random routing. We provide Qwen3.5-9B with the same routing context used by RSI-router, but ask it to directly judge the difficulty of the upcoming step and select the execution model. We compare it with the step-level Random baseline reported in Table 1. As shown in Figure 6(b), direct routing achieves lower mean performance across all five benchmarks. Thus, direct difficulty judgments do not yield a performance advantage over random step-level selection.

Execution skills affect performance–cost trade-offs in task-dependent ways. To isolate their effect, we remove execution skills while keeping the underlying subtask definitions and model assignments unchanged. As shown in Figure 6(c), removing skills lowers performance on ALFWorld, ScienceWorld, and WebShop by 12.4, 11.5, and 7.4 points, respectively. On SWE-bench Verified and Terminal-Bench 2.0, it instead improves performance by 9.7 and 10.5 points, but increases cost by 33.7% and 33.9% of the respective large-model-only costs. These results indicate that execution skills do not uniformly improve performance; they shift the performance–cost trade-off by modifying how the assigned models execute each subtask.

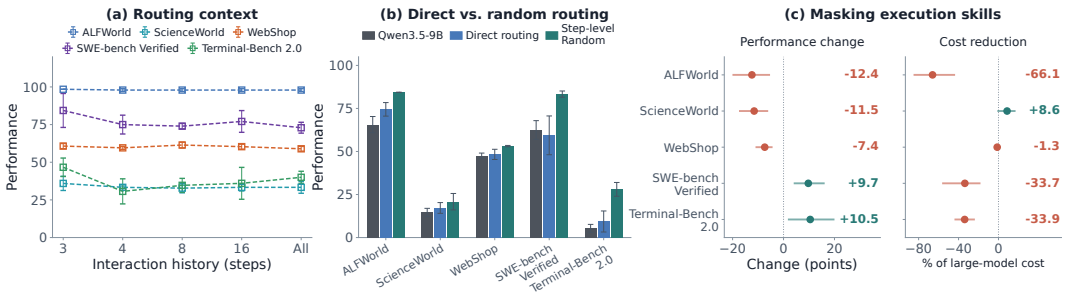

Figure 6: **Ablations of routing context, direct routing, and execution skills.** (a) Interaction history length. (b) Direct model routing by Qwen3.5-9B. (c) Effects of masking execution skills; cost reductions are normalized by the large-model-only USD cost. WebShop reward is scaled by 100. Error bars show standard deviations in (a–b) and 95% bootstrap intervals in (c).

4 Related Work

LLM routing. LLM routing exploits complementary model capabilities to improve performance–cost trade-offs (Hu et al., 2024; Li et al., 2026; Fernandez et al., 2026). At the task level, HybridLLM and RouteLLM select between small and large models using difficulty estimates and response preferences, respectively (Ding et al., 2024; Ong et al., 2025; Song et al., 2025). FrugalGPT instead adopts cascaded inference, sequentially calling models until a response is judged sufficiently reliable (Chen et al., 2024). GraphRouter and Avengers-Pro use graph-based matching and cluster-level performance–cost estimates to select models (Feng et al., 2025; Zhang et al., 2025b).

Recent works extend routing to multi-step reasoning and agentic tasks. MTRouter learns model selection from sampled trajectories (Zhang et al., 2026e), while Router-R1 and Budget-Aware Agentic Routing optimize routing policies through reinforcement learning (Zhang et al., 2025a; 2026a). SWE-Router uses partial execution traces to decide whether to switch to a more expensive model (Son et al., 2026). Agent-as-a-Router and Gated-Memory Routing inform model selection through accumulated outcomes and selective history retention, respectively (Zhou et al., 2026; Rajib et al., 2026). SkillOrchestra extracts skills from pre-collected trajectories to guide the router's model selection (Wang et al., 2026). RSI-router uses subtasks to align model assignments with capability requirements and reuse experience across tasks. It leverages LLM prior knowledge to explore assignments and distills training trajectories into execution skills for the selected models.

Self-evolving agents. Agent self-improvement spans prompt and context adaptation, automated agent design, and harness evolution. GEPA and ACE use execution feedback to refine prompts and contextual guidance, respectively (Agrawal et al., 2026; Zhang et al., 2026d). ADAS (Hu et al., 2025) searches over agent programs, while the Darwin Gödel Machine (Zhang et al., 2026c) iteratively modifies agent code and maintains an expanding archive of candidates. At the harness level, Meta-Harness (Lee et al., 2026) uses an external meta-agent powered by a strong model to optimize the target model's harness, whereas Self-Harness (Zhang et al., 2026b) lets the target agent improve its own harness through failure-driven edits. HarnessBank (Luo et al.) maintains a semantically diverse harness population, combining candidate recombination with gated verification. HarnessX (Chen et al., 2026) evolves composable harness components from execution traces, while Harness Continual Learning (Kang et al., 2026) supports continual harness adaptation under historical-retention constraints. We adopt this feedback-driven evolution process for agentic routing itself, recursively refining model selection and execution guidance from execution outcomes and accumulated experience.

5 Conclusion

We introduced *RSI-router*, a routing framework that combines LLM task knowledge with execution feedback to improve performance–cost trade-offs for agentic tasks. It organizes model assignments around reusable subtasks and uses comparisons with large-model-only trajectories to develop model-specific execution skills. Candidate updates are assessed through re-execution and Pareto-based router selection using validation performance and cost, while historical routers and skills guide subsequent iterations.

Across five agentic benchmarks, RSI-router reduces inference cost by 8.1–82.2% while matching or exceeding large-model-only performance and extends the performance–cost Pareto frontier beyond the evaluated task-level and step-level routing baselines. These results demonstrate that smaller models can contribute to tasks they cannot complete independently when assigned suitable subtasks and supported by execution skills. Further analysis shows repeated cost reductions and continued reuse of historical skills, highlighting the value of accumulated execution experience.

More broadly, these findings support using iterative evolution to construct routing strategies for agentic tasks. They highlight the potential to improve agentic performance and efficiency by better exploiting complementary model capabilities, allowing smaller models to contribute to tasks beyond what they can complete independently. RSI-router provides a foundation for constructing cost-aware routing strategies from accumulated execution experience, reducing reliance on continual manual refinement for agentic tasks.

AI USE STATEMENT

In this work, we used Codex to assist with manuscript editing, including improving grammar, clarity, concision, and the overall readability of the presentation. We also used Codex to assist with creating or modifying scientific figures and images, as well as developing experimental code. The authors take full responsibility for the accuracy and integrity of the final manuscript and accompanying research artifacts, including all AI-assisted text, figures, and code, and for the scientific claims and conclusions presented in this work.

REFERENCES

Abdelrahman Abouelenin, Atabak Ashfaq, Adam Atkinson, Hany Awadalla, Nguyen Bach, Jianmin Bao, Alon Benhaim, Martin Cai, Vishrav Chaudhary, Congcong Chen, et al. Phi-4-mini technical report: Compact yet powerful multimodal language models via mixture-of-loras. *arXiv preprint arXiv:2503.01743*, 2025.

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

Lisa Alazraki, William F. Shen, Yoram Bachrach, and Akhil Mathur. Scaling small agents through strategy auctions. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=elXuA5wTWV.

Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=cSimKw5p6R.

Tingyang Chen, Shuo Lu, Kang Zhao, Weicheng Meng, Hanlin Teng, Tianhao Li, Chao Li, Xule Liu, Jian Liang, Zhizhong Zhang, et al. Harnessx: A composable, adaptive, and evolvable agent harness foundry. *arXiv preprint arXiv:2606.14249*, 2026.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench Verified. OpenAI, August 2024. URL https://openai.com/index/introducing-swe-bench-verified/. Accessed: 2026-09-17.

DeepSeek-AI. DeepSeek-V4.1-Flash: Pushing the limits of KV cache compression. Hugging Face model card, 2026. URL https://huggingface.co/deepseek-ai/DeepSeek-V4.1-Flash. Accessed: 2026-09-17.

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks Lakshmanan, and Ahmed H Awadallah. Hybrid llm: Cost-efficient and quality-aware query routing. In *International Conference on Learning Representations*, volume 2024, pp. 41348–41366, 2024.

Tao Feng, Yanzhen Shen, and Jiaxuan You. Graphrouter: A graph-based router for llm selections. In *International Conference on Learning Representations*, volume 2025, pp. 26186–26203, 2025.

Nigel Steven Fernandez, Branislav Kveton, Ryan Rossi, Andrew Lan, and Jack Wang. Radar: Reasoning-ability and difficulty-aware routing for reasoning llms. In *International Conference on Learning Representations*, volume 2026, pp. 109429–109457, 2026.

Pengfei Gao and Chao Peng. More with less: An empirical study of turn-control strategies for efficient coding agents. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pp. 956–967, 2026.

Qitian Jason Hu, Jacob Bieker, Xiuyu Li, Nan Jiang, Benjamin Keigwin, Gaurav Ranganath, Kurt Keutzer, and Shriyash Kaustubh Upadhyay. Routerbench: A benchmark for multi-llm routing system. *arXiv preprint arXiv:2403.12031*, 2024.

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

Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. Llm-blender: Ensembling large language models with pairwise ranking and generative fusion. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14165–14178, 2023.

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

Wittawat Jitkrittum, Harikrishna Narasimhan, Ankit Singh Rawat, Jeevesh Juneja, Congchao Wang, Zifeng Wang, Alec Go, Chen-Yu Lee, Pradeep Shenoy, Rina Panigrahy, et al. Universal model routing for efficient llm inference. In *International Conference on Learning Representations*, volume 2026, pp. 10169–10218, 2026.

Borui Kang, Jinrui Gu, Junhan Lv, Wenbin Li, Lei Wang, and Yang Gao. Harness continual learning: Continual adaptation beyond model parameters. *arXiv preprint arXiv:2608.19013*, 2026.

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.

Hao Li, Yiqun Zhang, Zhaoyan Guo, Chenxu Wang, Shengji Tang, Qiaosheng Zhang, Yang Chen, Biqing Qi, Peng Ye, Lei Bai, et al. Llmrouterbench: A massive benchmark and unified framework for llm routing. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 37733–37754, 2026.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *ACM Transactions on Software Engineering and Methodology*, 2024.

Chris Lu, Cong Lu, Robert Tjarko Lange, Yutaro Yamada, Shengran Hu, Jakob Foerster, David Ha, and Jeff Clune. Towards end-to-end automation of ai research. *Nature*, 651(8107):914–919, 2026.

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*.

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

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, Mohammed Kadous, and Ion Stoica. Routellm: Learning to route llms from preference data. In *International Conference on Learning Representations*, volume 2025, pp. 34433–34448, 2025.

OpenAI. Introducing Codex, May 2025. URL `https://openai.com/index/introducing-codex/`. Accessed: 2026-09-17.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`. Accessed: 2026-09-17.

Rakibul Hasan Rajib, Mengxing Zheng, and Qian Lou. Learning what to retain: Gated-memory routing for efficient collaboration in multi-agent llm systems. *arXiv preprint arXiv:2609.00237*, 2026.

Pascal J Sager, Benjamin Meyer, Peng Yan, Rebekka von Wartburg-Kottler, Layan Etaiwi, Aref Enayati, Gabriel Nobel, Ahmed Abdulkadir, Benjamin F Grewe, and Thilo Stadelmann. A comprehensive survey of agents for computer use: Foundations, challenges, and future directions. *Journal of Artificial Intelligence Research*, 85, 2026.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. {ALFW}orld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=0IOX0YcCdTn.

Seongho Son, Sangwoong Yoon, Jiahua Tang, Shuhan Wang, Lorenz Wolf, and Ilija Bogunovic. Swe-router: Routing in multi-turn agentic software engineering tasks. *arXiv preprint arXiv:2607.00053*, 2026.

Wei Song, Zhenya Huang, Cheng Cheng, Weibo Gao, Bihan Xu, GuanHao Zhao, Fei Wang, and Runze Wu. Irt-router: Effective and interpretable multi-llm routing via item response theory. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15629–15644, 2025.

Jiayu Wang, Yifei Ming, Zixuan Ke, Shafiq Joty, Aws Albarghouthi, and Frederic Sala. Skillorchestra: Learning to route agents via skill transfer. *arXiv preprint arXiv:2602.19672*, 2026.

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.

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

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022.

Caiqi Zhang, Menglin Xia, Xuchao Zhang, Daniel Madrigal, Ankur Mallick, Samuel Kessler, Victor Ruehle, and Saravan Rajmohan. Budget-aware agentic routing via boundary-guided training. *arXiv preprint arXiv:2602.21227*, 2026a.

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

Haozhen Zhang, Tao Feng, and Jiaxuan You. Router-r1: Teaching LLMs multi-round routing and aggregation via reinforcement learning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025a. URL https://openreview.net/forum?id=DWf4vroKWJ.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Tjarko Lange, and Jeff Clune. Darwin gödel machine: Open-ended evolution of self-improving agents. In *The Fourteenth International Conference on Learning Representations*, 2026c. URL https://openreview.net/forum?id=pUpzQZTvGY.

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

Yiqun Zhang, Hao Li, Jianhao Chen, Hangfan Zhang, Peng Ye, Lei Bai, and Shuyue Hu. Beyond gpt-5: Making llms cheaper and better via performance-efficiency optimized routing. In *Proceedings of the 2025 7th International Conference on Distributed Artificial Intelligence*, pp. 122–129, 2025b.

Yiqun Zhang, Hao Li, Zihan Wang, Shi Feng, Xiaocui Yang, Daling Wang, Bo Zhang, Lei Bai, and Shuyue Hu. Mtrouter: Cost-aware multi-turn llm routing with history–model joint embeddings. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 44206–44226, 2026e.

Pengfei Zhou, Zhiwei Tang, Yixing Ma, Jiasheng Tang, Yizeng Han, Zhenglin Wan, Fanqing Meng, Wei Wang, Bohan Zhuang, Wangbo Zhao, et al. Agent-as-a-router: Agentic model routing for coding tasks. *arXiv preprint arXiv:2606.22902*, 2026.

A Baseline Implementation Details

Shared setup. We use the official implementations of all task-level and step-level routing baselines. All methods route between L and S under the experimental setup in Section 3.1. For each benchmark, we fit the task-level and step-level routing baselines on $\mathcal{D}_{\text{tr}} \cup \mathcal{D}_{\text{val}}$ and evaluate all methods on $\mathcal{D}_{\text{test}}$. All results are averaged over three evaluation repeats per test task. For learned baselines trained with three seeds, we additionally average across these training seeds.

Fixed policies. The large-model-only router r_L and the small-model-only router r_S use L and S, respectively, throughout task execution. The task-level random router selects either model with equal probability once per task and uses it throughout execution. The step-level random router selects either model with equal probability before each execution step. These baselines require no router training.

HybridLLM. We use HybridLLM (Ding et al., 2024) with a Qwen3-Embedding-0.6B encoder and a binary classification head to estimate task difficulty. The encoder and classification head are jointly fine-tuned using AdamW, with a learning rate of 10^{-5}, weight decay of 0.1, and batch size 4. Gradient accumulation gives an effective batch size of 64. The warmup ratio is 0.05, and the tolerated performance gap between the two models is set to zero.

FrugalGPT. We adapt FrugalGPT (Chen et al., 2024) to a task-level cascade. For each execution model, we fine-tune a separate Qwen3-Embedding-0.6B scorer that predicts task performance from the user query, using mean observed task performance as soft supervision. Each scorer is trained using AdamW, with a learning rate of 2×10^{-5}, weight decay of 0.01, batch size 4, and a warmup ratio of 0.06. Cost includes the task execution costs of all models invoked by the cascade.

RouteLLM. We use the matrix-factorization router of RouteLLM (Ong et al., 2025), with frozen Qwen3-Embedding-8B query embeddings projected into a 128-dimensional space. Model embeddings and the projection are trained using pairwise preferences derived from observed task performance. Optimization uses Adam with a learning rate of 3×10^{-4}, weight decay of 10^{-5}, and batch size 64. At inference time, the estimated preference probability determines model selection.

GraphRouter. We use GraphRouter (Feng et al., 2025) for graph-based model selection, with Qwen3-Embedding-8B representations of queries, tasks, and model descriptions. All models tied for the highest training utility receive positive labels. Training uses AdamW, with a learning rate of 3×10^{-4} and weight decay of 10^{-4}. We evaluate the three settings defined in the original paper: *Performance First*, *Balance*, and *Cost First*.

Avengers-Pro. We use Avengers-Pro (Zhang et al., 2025b) with Qwen3-Embedding-8B query embeddings and k-means clustering with $k = 8$. Models are ranked using cluster-level performance–cost estimates, and each query is routed according to the ranking in its nearest cluster. We evaluate ten evenly spaced performance coefficients in $\{0, 1/9, \ldots, 1\}$, with the complementary weight assigned to cost.

Router-R1. We adapt Router-R1 (Zhang et al., 2025a) with Qwen3.5-9B as the policy model. At each step, the policy interleaves reasoning with model calls and integrates the returned responses to generate the next action. Training uses PPO with task-level outcome rewards, actor and critic learning rates of 10^{-6} and 10^{-5}, respectively, and a KL penalty coefficient of 10^{-3}. We use a rollout batch size of 64 and a PPO mini-batch size of 32.

MTRouter. We train MTRouter (Zhang et al., 2026e) on trajectories collected by r_L, r_S, and the step-level random router. It selects a model at each step using the interaction history, encoded by frozen Qwen3-Embedding-8B. The outcome network has two hidden layers of width 256 and dropout 0.1. Training uses AdamW with early stopping, a learning rate of 10^{-3}, weight decay of 0.01, and batch size 64.

B BENCHMARK PROMPTS

B.1 ALFWORLD

ALFWorld System Prompt

You are a helpful assistant interacting with ALFWorld, a text-based household environment. Your goal is to complete the household task by issuing text commands.

How to Interact
Write exactly one command in a ```text``` code block. The environment will respond with observations.

```
<format_example>
THOUGHT: I should inspect the room and choose a valid action.
```text
look
```
</format_example>
```

Important Rules

- Issue ONE command per response.
- Prefer commands from the available actions list shown in the observation.
- If no available actions are shown, issue the most likely ALFWorld command.
- Do not claim the task is complete yourself; keep acting until the environment ends the episode.

ALFWorld User Prompt

```
<task_description>
{{task_description}}
</task_description>
<task_type>
{{task_type}}
</task_type>
<initial_observation>
{{initial_observation}}
</initial_observation>
{% if valid_actions %}
<available_actions>
{% for action in valid_actions %}
- {{ action }}
{% endfor %}
</available_actions>
{% endif %}
```

Complete the ALFWorld task. Think briefly, then choose exactly one command.

B.2 SCIENCEWORLD

ScienceWorld System Prompt

You are a helpful assistant interacting with a text-based science simulation environment. Your goal is to complete science experiments by issuing text commands.

How to Interact
You issue commands by writing them in a ```text``` code block. The environment will respond with observations.

```
<format_example>
THOUGHT: I should explore my surroundings first.
```text
look around
```
</format_example>
```

Available Command Types
Common commands include:

ScienceWorld System Prompt *continued*

- Movement: "go to [location]", "open door", "go through door"
- Interaction: "pick up [object]", "put [object] in [container]", "activate [object]"
- Observation: "look around", "look at [object]", "inventory"
- Task-specific: "focus on [object]", "use [tool] on [object]"

Query Commands (Free Actions)
You can query available actions without consuming a game turn:

- ?navigation - Show movement actions (go, walk, move)
- ?object - Show object manipulation (pick up, put, pour)
- ?observation - Show observation actions (look, examine, inventory)
- ?device - Show device control (activate, turn on/off, use)
- ?door - Show door/container actions (open, close)
- ?electrical - Show electrical actions (connect, disconnect)
- ?interaction - Show interaction actions (mix, eat, focus)
- ?all - Show all valid actions
- ?categories - Show query help

Use queries to explore available actions before deciding your next move.

Important Notes

- Issue ONE command per response
- Include a THOUGHT section explaining your reasoning
- The environment is turn-based - wait for observations before issuing the next command
- Some tasks require multiple steps to complete

ScienceWorld User Prompt

<task_description>
{{task_description}}
</task_description>

<initial_observation>
{{initial_observation}}
</initial_observation>

<instructions>
Complete the science experiment described above. You are interacting with a simulated environment. Issue commands one at a time and observe the results.

Tip: Use query commands like ?navigation or ?object to explore available actions without consuming a turn.

Strategy hints:

- First explore: use "open door to [room]" then "go to [room]" to navigate
- Look around each room to find objects you need
- Pick up objects with "pick up [object]"
- For heating: find a stove, turn it on with "activate [stove]", place container on it
- For cooling: use a freezer or refrigerator
- Use "focus on [object]" to examine substances

To complete the task, perform the necessary actions described in the task description. When you believe the task is complete, issue the command:

```text
task completed
```

Remember:

- Think step by step about what actions are needed
- Explore your environment to find needed objects
- Some actions may require prerequisites (e.g., picking up an object before using it)

</instructions>

B.3 WEBSHOP

WebShop System Prompt

You are an expert autonomous agent operating in the WebShop e-commerce environment. Your goal is to buy the product that best satisfies the user instruction.

Valid command formats are:

- search[keywords]
- click[button or product id or option]

For each step, think internally if needed, then output exactly one admissible WebShop command in `<action>...</action>`.

The command inside `<action>` must be chosen from the current admissible actions list, except `search[...]` may replace `search[<your query>]` with useful keywords.

Your final visible response must contain the `<action>...</action>` tag. Do not spend the whole response on hidden reasoning.
Do not claim the task is complete yourself; keep acting until the environment ends the episode.

WebShop User Prompt

```
<task_description>
{{task_description}}
</task_description>
<task_type>
{{task_type}}
</task_type>
<history>
{{action_history}}
</history>
<current_observation>
{{current_observation}}
</current_observation>
<admissible_actions>
{{valid_actions_text}}
</admissible_actions>
```

You are at WebShop step `{{current_step}}`. Choose one action now.

```
<format_example>
<think>I should search for the requested product type.</think>
<action>search[black running shoes]</action>
</format_example>
```

B.4 SWE-BENCH VERIFIED

The assistant issues native `bash` tool calls with a `command` argument. Execution results are returned as `tool` messages.

SWE-bench Verified System Prompt

You are a helpful assistant that can interact with a computer shell to solve programming tasks.

SWE-bench Verified User Prompt

```
<pr_description>
Consider the following PR description:
{{task}}
</pr_description>
```

SWE-bench Verified User Prompt *continued*

<instructions>

Task Instructions

Overview

You're a software engineer interacting continuously with a computer by submitting commands. You'll be helping implement necessary changes to meet requirements in the PR description. Your task is specifically to make changes to non-test files in the current directory in order to fix the issue described in the PR description in a way that is general and consistent with the codebase.
<IMPORTANT>This is an interactive process where you will think and issue AT LEAST ONE command, see the result, then think and issue your next command(s).</important>

For each response:

1. Include a THOUGHT section explaining your reasoning and what you're trying to accomplish
2. Provide one or more bash tool calls to execute

Important Boundaries

- MODIFY: Regular source code files in /testbed (this is the working directory for all your subsequent commands)
- DO NOT MODIFY: Tests, configuration files (pyproject.toml, setup.cfg, etc.)

Recommended Workflow

1. Analyze the codebase by finding and reading relevant files
2. Create a script to reproduce the issue
3. Edit the source code to resolve the issue
4. Verify your fix works by running your script again
5. Test edge cases to ensure your fix is robust

Command Execution Rules

You are operating in an environment where

1. You issue at least one command
2. The system executes the command(s) in a subshell
3. You see the result(s)
4. You write your next command(s)

Each response should include:

1. **Reasoning text** where you explain your analysis and plan
2. At least one tool call with your command

CRITICAL REQUIREMENTS:

- Your response SHOULD include reasoning text explaining what you're doing
- Your response MUST include AT LEAST ONE bash tool call. You can make MULTIPLE tool calls in a single response when the commands are independent (e.g., searching multiple files, reading different parts of the codebase).
- Directory or environment variable changes are not persistent. Every action is executed in a new subshell.
- However, you can prefix any action with `MY_ENV_VAR=MY_VALUE cd /path/to/working/dir && ...` or write/load environment variables from files

Example of a CORRECT response:

```
<example_response>
I need to understand the Builder-related code.  Let me find relevant
files and check the project structure.
[Makes multiple bash tool calls:  {"command":  "ls -la"}, {"command":
"find src -name '*.java' | grep -i builder"}, {"command":  "cat
README.md | head -50"}]
</example_response>
```

Environment Details

- You have a full Linux shell environment
- Always use non-interactive flags (-y, -f) for commands
- Avoid interactive tools like vi, nano, or any that require user input
- You can use bash commands or invoke any tool that is available in the environment
- You can also create new tools or scripts to help you with the task

SWE-bench Verified User Prompt *continued*

- If a tool isn't available, you can also install it

Submission

When you've completed your work, you MUST submit your changes as a git patch. Follow these steps IN ORDER, with SEPARATE commands:

Step 1: Create the patch file
Run `git diff -- path/to/file1 path/to/file2 > patch.txt` listing only the source files you modified. Do NOT commit your changes.

<IMPORTANT>
The patch must only contain changes to the specific source files you modified to fix the issue. Do not submit file creations or changes to any of the following files:

- test and reproduction files
- helper scripts, tests, or tools that you created
- installation, build, packaging, configuration, or setup scripts unless they are directly part of the issue you were fixing (you can assume that the environment is already set up for your client)
- binary or compiled files

</IMPORTANT>

Step 2: Verify your patch
Inspect patch.txt to confirm it only contains your intended changes and headers show `--- a/` and `+++ b/` paths.

Step 3: Submit (EXACT command required)
You MUST use this EXACT command to submit:

```bash
echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && cat patch.txt
```

If the command fails (nonzero exit status), it will not submit.

<CRITICAL>

- Creating/viewing the patch and submitting it MUST be separate commands (not combined with &&).
- If you modify patch.txt after verifying, you SHOULD verify again before submitting.
- You CANNOT continue working (reading, editing, testing) in any way on this task after submitting.

</CRITICAL>
</instructions>

SWE-bench Verified Bash Tool Definition

```
{
  "type": "function",
  "function": {
    "name": "bash",
    "description": "Execute a bash command",
    "parameters": {
      "type": "object",
      "properties": {
        "command": {
          "type": "string",
          "description": "The bash command to execute"
        }
      },
      "required": [
        "command"
      ]
    }
  }
}
```

B.5 TERMINAL-BENCH 2.0

The assistant issues native `bash` tool calls with a `command` argument. Execution results are returned as `tool` messages.

Terminal-Bench 2.0 System Prompt

You are a helpful assistant that solves tasks by interacting with a computer shell.

Terminal-Bench 2.0 User Prompt

`{{task}}`

Work in the current task environment and continue until the task is fully complete.

Each response must include reasoning about the next step and at least one bash tool call. Directory and environment-variable changes are not persistent between tool calls, so include them in each command when needed.

When the task is complete, run `echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT` as a standalone command. Do not combine it with another command; after it runs you cannot continue working.

```
<system_information>
{{system}} {{release}} {{version}} {{machine}}
</system_information>
```

Terminal-Bench 2.0 Bash Tool Definition

```
{
  "type": "function",
  "function": {
    "name": "bash",
    "description": "Execute a bash command",
    "parameters": {
      "type": "object",
      "properties": {
        "command": {
          "type": "string",
          "description": "The bash command to execute"
        }
      },
      "required": [
        "command"
      ]
    }
  }
}
```

C ESTIMATING SMALL-MODEL COST FROM GPU HOURS

Pricing reference. We use the official DeepSeek-V4.1-Flash API prices as the USD reference: \$0.30 per million input tokens and \$1.20 per million output tokens.[1] We estimate Qwen3.5-9B's relative cost using GPU-hour measurements, then express its input and output prices relative to these official rates.

Measurement protocol. We benchmark DeepSeek-V4.1-Flash on eight NVIDIA H200 GPUs and Qwen3.5-9B on one H200 using SGLang. Both deployments receive identical synthetic texts at three reference input lengths and two fixed output lengths (Table 2). Each setting uses 1,024 requests with 64 concurrent requests, replenished as requests finish. For model m using n_m GPUs over T_m seconds, we compute

$$G_m = \frac{n_m T_m}{3600} \quad \text{GPU hours.} \tag{1}$$

[1] `https://api-docs.deepseek.com/quick_start/pricing/`

Table 2: GPU hours for 1,024 matched requests per model at concurrency 64. Input lengths use the Qwen3.5-9B tokenizer as reference. Each setting is measured once. Ratios are DeepSeek/Qwen.

Tokens per request		GPU hours		
Input	Output	DeepSeek-V4.1-Flash 8 H200s	Qwen3.5-9B 1 H200	Ratio
2,048	128	0.9842	0.0199	49.41
2,048	1,024	5.4078	0.0545	99.15
8,192	128	2.0729	0.0679	30.55
8,192	1,024	6.4912	0.1153	56.28
32,768	128	7.1375	0.2819	25.32
32,768	1,024	11.5576	0.3962	29.17
All six settings		33.6512	0.9357	35.96

Cost estimate. Across the six synthetic workloads, the total GPU-hour ratio between DeepSeek-V4.1-Flash and Qwen3.5-9B is 35.96. We use 1:30 as an approximate reference for pricing Qwen3.5-9B, yielding input and output rates of $0.01 and $0.04 per million tokens, respectively.

D Case Studies of Router Evolution

We illustrate representative routing changes and execution skills during router evolution.

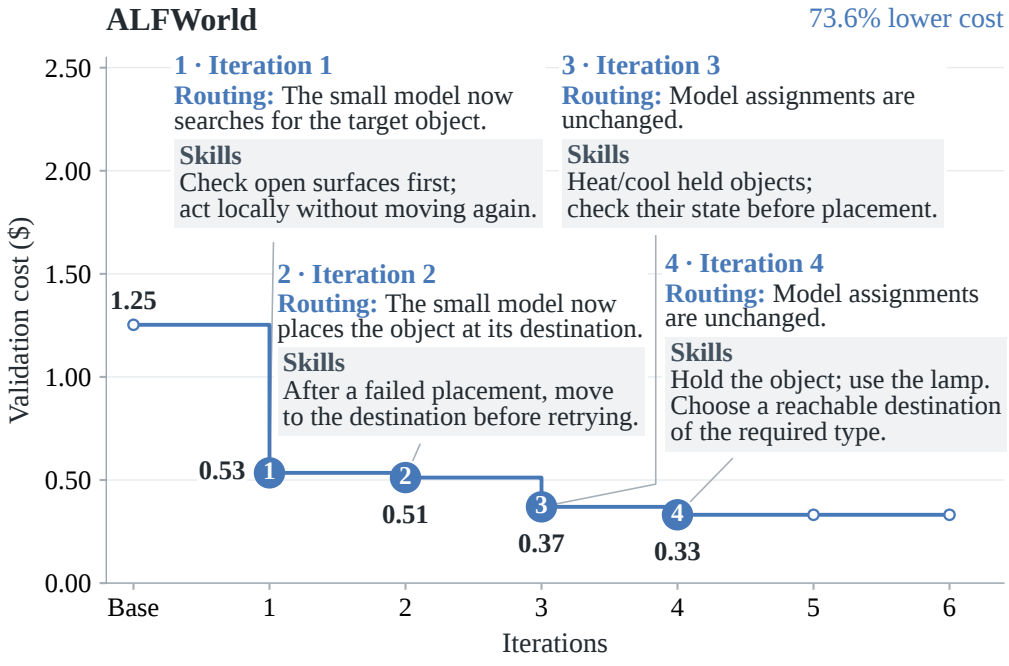

Figure 7: **Router evolution on ALFWorld.**

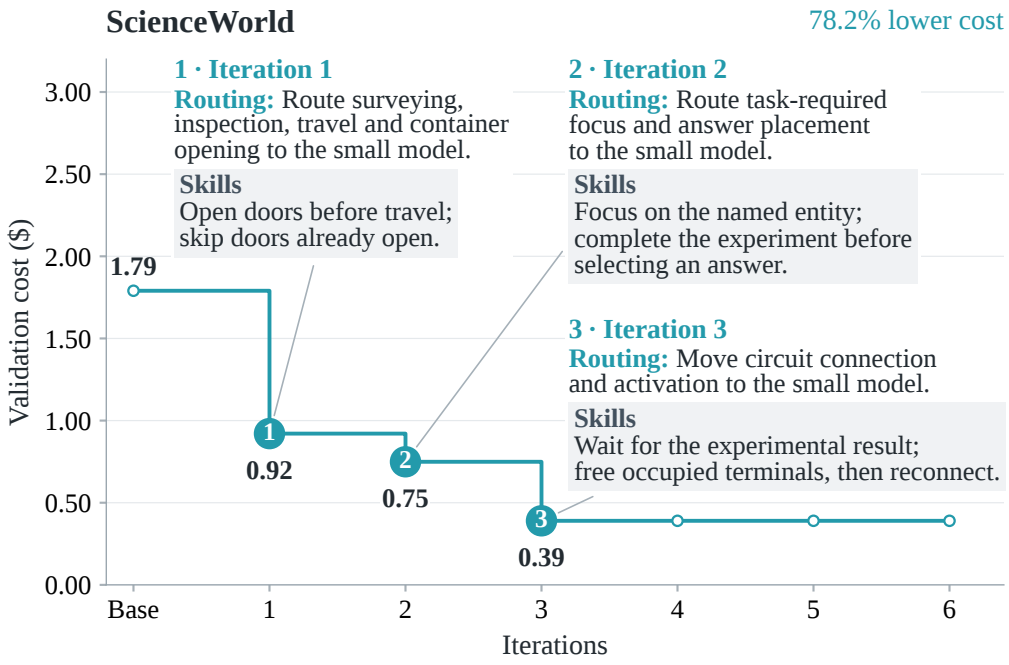

Figure 8: **Router evolution on ScienceWorld.**

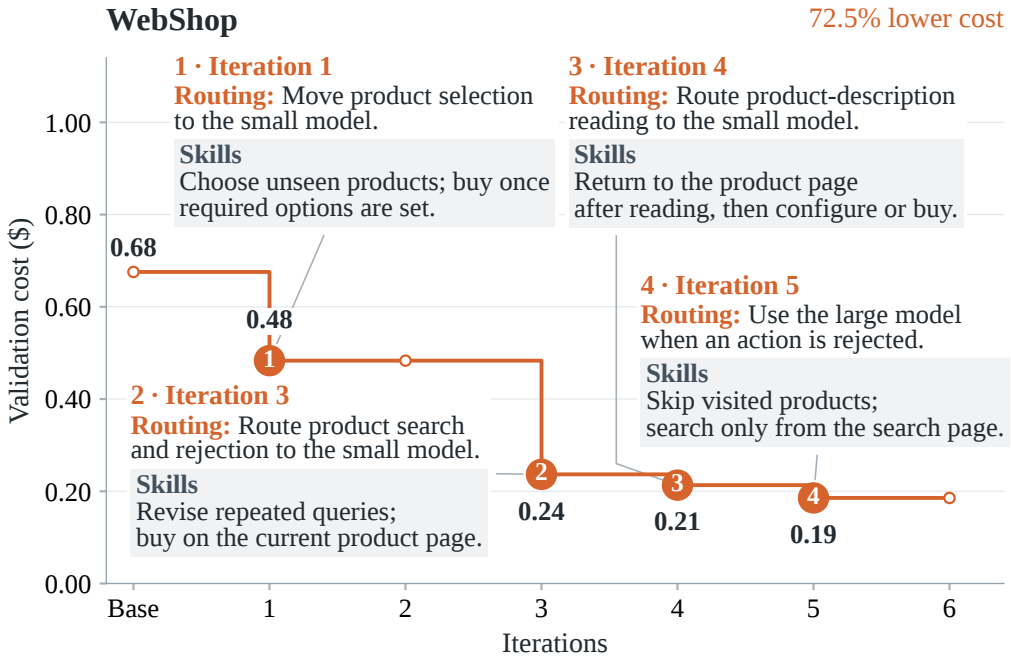

Figure 9: **Router evolution on WebShop.**

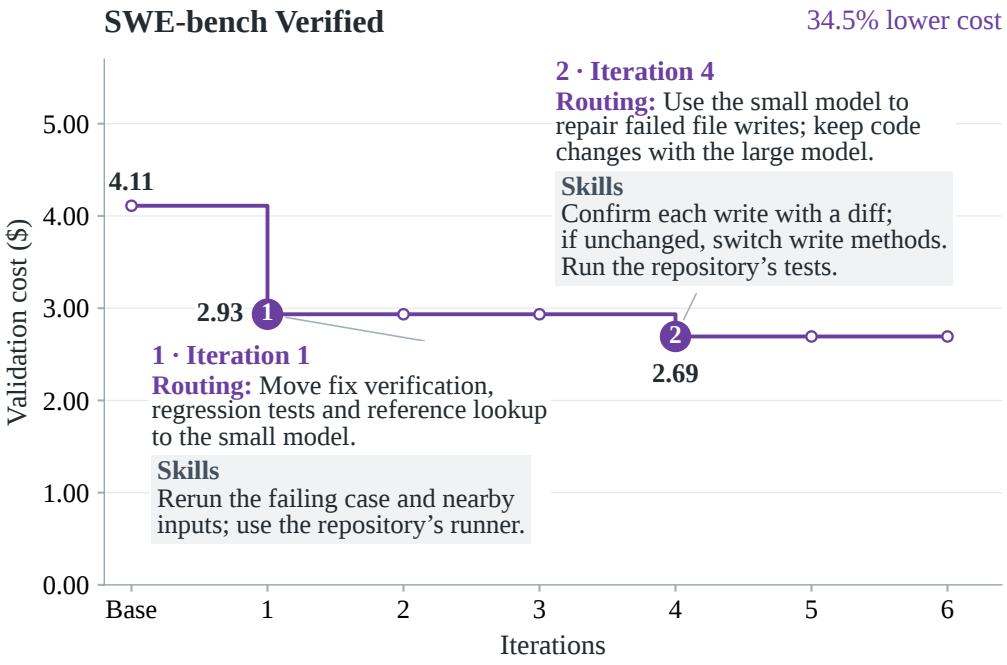

Figure 10: **Router evolution on SWE-bench Verified.**

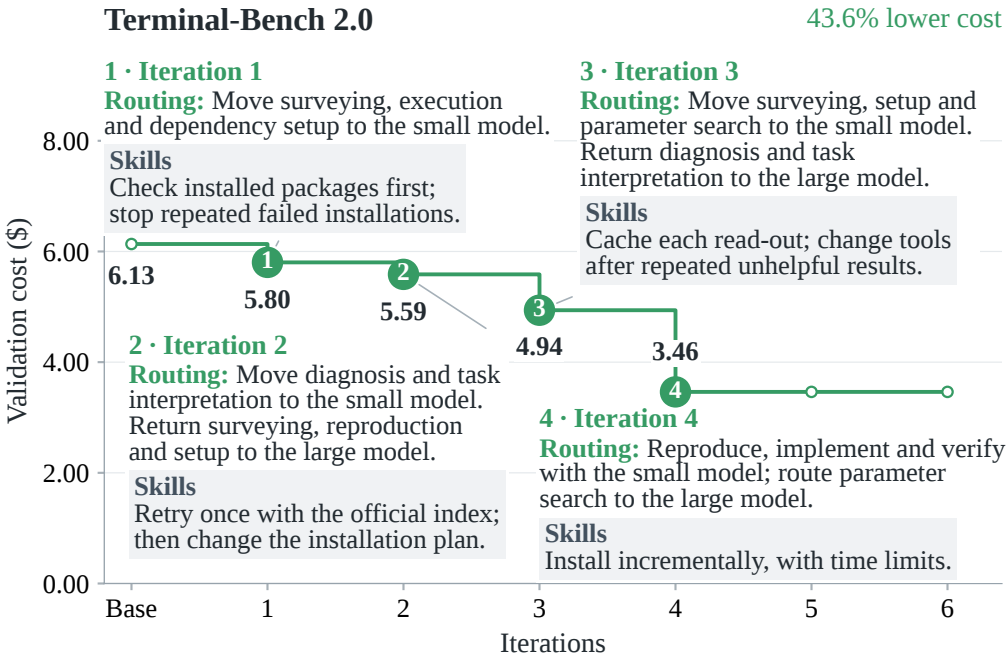

Figure 11: **Router evolution on Terminal-Bench 2.0.**