ECHOROUTER: FORECAST-GUIDED EVOLUTION OF RUNTIME-ADAPTIVE MULTI-AGENT SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Multi-agent systems face a recurring decision: after each intermediate result, is another model call worth its cost? The right amount of collaboration varies dramatically: $1 + 1$ hardly needs an agent, while an attempt at a Navier–Stokes proof may call for many specialists. Making this decision requires choosing how agents collaborate, which models to assign, what context each call receives, and when to stop under a budget. Code-based workflow optimizers such as AFlow can express conditional execution, but searching through repeated real rollouts is costly, especially on multi-turn tasks. We introduce ECHOROUTER, which uses experience from past execution trajectories as an "echo" to understand where the current policy stands and guide subsequent search. During optimization, it proposes diverse candidate policies, each decomposed into three decisions: how to collaborate during task execution, which models to assign, and what each call receives and returns. Guided by the echo, EchoRouter treats LLM as value model to forecast candidate quality and cost, executes promising policies, reflects on their trajectories, and updates the echo. It also compares models and prompts on saved task states while hiding model names to improve credit assignment. During execution, a lightweight controller uses the selected policy and current task state to construct subgraphs, assign models, and specify each call's prompt and output format. Across various model-pool scales and mathematical, coding, and general reasoning benchmarks, ECHOROUTER achieves quality–cost trade-offs. Results on multi-turn tasks and open-ended artifact generation further show competitive quality under an LLM as Judge evaluation. Its decoupled, model-anonymous design allows new LLMs to be assessed without rebuilding the collaboration policy. Together, these results demonstrate the effectiveness and potential of ECHOROUTER.

1 INTRODUCTION

LLM-based agent systems have rapidly expanded beyond code generation to reasoning, general-purpose tool use, long-horizon tasks, and the creation of open-ended artifacts (Hong et al., 2023; Qian et al., 2023; Du et al., 2023; Wang et al., 2023). Multi-agent systems (MAS) can distribute these tasks among agents with different roles, models, and tools. Yet many MAS architectures still depend on expert-designed interaction patterns, while automatic methods often search over a predefined workflow or restricted choice spaces (Wu et al., 2023; Hong et al., 2023; Li et al., 2023). This design setting is increasingly distant from tasks that involve repeated tool calls, evolving intermediate results, and substantial output artifacts.

Optimizing MAS for these tasks raises two challenges. First, the practical search space is large because jointly exploring collaboration structures, models, prompts, tools, and stopping rules requires costly real rollouts, especially for long-horizon, multi-turn tasks involving many sequential calls. Second, a task-agnostic workflow can misallocate computation. To cover different possible outcomes, a fixed workflow often assigns collaboration steps before they are needed. These steps may waste model calls when a task is simple or proceeds smoothly. Conversely, distributing computation uniformly across a complex task may fail to address the specific needs of individual stages. Effective collaboration should therefore adapt to intermediate results by deciding when additional computation is needed and what action to take next.

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.

Existing methods have made progress on these challenges. For efficient search, EvoMAS searches over topology, models, prompts, and tools, but real-rollout evaluation remains expensive (Hu et al., 2026). AgentSwift reduces this cost with a value model, though its estimates depend on the training task setting (Li et al., 2026). For adaptive collaboration, AFlow supports branching on intermediate results but fixes its collaboration logic before execution (Zhang et al., 2024b), while MasRouter selects collaboration modes, roles, and models from the input task without adapting them to evolving intermediate states (Yue et al., 2025). To address these limitations, we propose ECHOROUTER, which uses past experience to generate candidate policies that guide the adaptive construction of collaboration during execution.

On the one hand, to reduce search cost, we propose *forecast before spending*. ECHOROUTER abstracts lessons from past execution trajectories and records of model performance into an *echo*. Guided by this echo, it generates diverse candidate policies and forecasts their potential. Only promising candidates proceed to real rollouts, reducing the cost of exploration. Each policy is represented as a structured JSON object that separates how agents collaborate, which models are assigned, and how each model responds. This representation supports independent revision of each decision during optimization and provides the controller with explicit guidance during execution.

ECHOROUTER then runs policies on the training set to produce task trajectories, compares candidate models and prompts in the same contexts drawn from those trajectories, and iteratively improves the policies. Each comparison changes one choice while holding the task, upstream state, and other call conditions fixed, enabling more reliable credit assignment. Throughout the process, model names are hidden from the optimizer and controller, encouraging decisions based on observed performance recorded in model performance rather than on model names. This design also makes it easier to integrate newly available LLMs into the policies, improving the method's generalization.

On the other hand, a candidate policy guides a lightweight LLM controller in constructing collaboration adaptively. Based on the current task state and remaining budget, the controller decides whether to stop or construct the next subgraph. It assigns models to roles and specifies the visible context, prompt, available tools, and output format for each call. The workflow can therefore change when a tool returns a result or a disagreement emerges. This stage-by-stage construction allows the system to adjust its computation as the task evolves, making it well suited to multi-turn tool use and iterative artifact generation.

Across model pools of two sizes, ECHOROUTER achieves favorable quality–cost trade-offs on reasoning, coding, and general-purpose benchmarks. Results on multi-turn tasks and open-ended artifact generation further demonstrate its applicability to complex settings while maintaining cost efficiency. Our main contributions are summarized as follows.

① ***State-adaptive collaboration.*** We optimize a policy that constructs each next collaboration stage from the current task state, allowing a lightweight controller to decide which subgraph to execute and achieve effective computation allocation.

② ***Forecast before spending.*** We treat a general-purpose LLM as a value model and use Echo experience to forecast candidate quality and cost before real rollouts, prioritizing evaluations when long-horizon execution makes search expensive.

③ **T*rajectory-based credit assignment.*** We compare candidate models and prompts on matched task states and update anonymous, role-specific model profiles. This evidence helps distinguish the effects of model and prompt choices and lets new models be assessed without rebuilding the collaboration policy.

2 RELATED WORK

Automated MAS design. Automated MAS design has moved from expert-defined coordination toward broader configuration search. Frameworks such as AutoGen, MetaGPT, CAMEL, and AgentVerse provide abstractions for roles, communication, and tools, but still rely heavily on human expertise (Wu et al., 2023; Hong et al., 2023; Li et al., 2023; Chen et al., 2023b). Recent methods automate more of this process. GPTSwarm optimizes prompts and graph connectivity, AFlow searches code-represented workflows, ADAS programs new agent designs, and EvoMAS evolves query-specific configurations of topology, models, prompts, and tools (Zhuge et al., 2024; Zhang

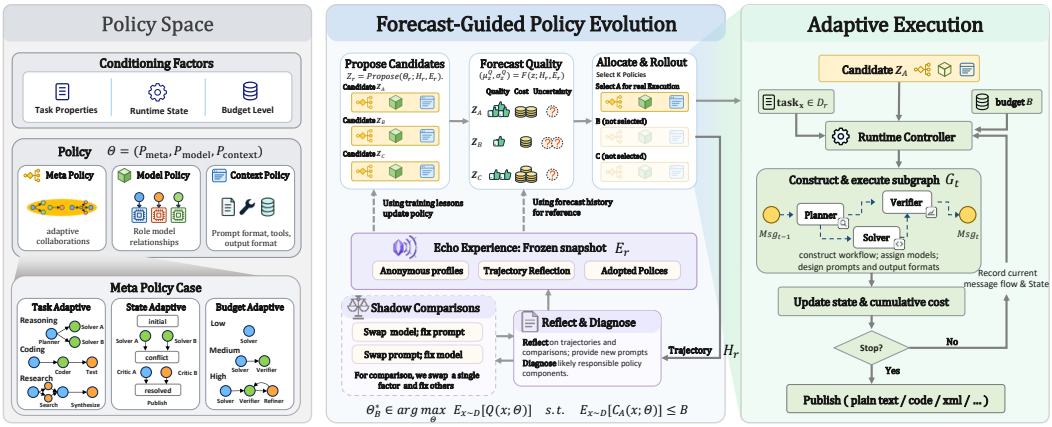

Figure 1: Overview of ECHOROUTER. The meta, model, and context-policy components define the conditional policy space. Historical executions, together with model and prompt comparisons, constitute the Echo experience. Forecasts select candidates for real rollout, after which observed outcomes determine policy admission. At runtime, the frozen policy constructs and executes subgraphs according to the evolving intermediate state.

et al., 2024b; Hu et al., 2024; 2026). However, each candidate still defines a design for an entire run rather than adapting collaboration to intermediate states. Real-rollout evaluation over such spaces is costly. AgentSquare, FLOW-GNN, and AgentSwift reduce this cost through performance prediction (Shang et al., 2024; Zhang et al., 2025c; Li et al., 2026), but depend on examples or training data relevant to the target setting. ECHOROUTER instead uses an LLM grounded in past trajectories to forecast candidate quality and cost without training a task-specific value model.

Runtime collaboration. Runtime methods adapt different components of an MAS at different levels. MaAS samples task-conditioned architectures, DyLAN selects agent teams, and AgentPrune and AgentDropout remove redundant communication or agents (Zhang et al., 2025a; Liu et al., 2023; Zhang et al., 2025b; Wang et al., 2025). Model-routing methods select models by expected quality and cost. RouteProfile profiles newly available models, while EvoRoute updates model choices across workflow steps (Chen et al., 2023a; Xu et al., 2026; Zhang et al., 2026). Structural adaptation remains more limited. G-Designer and MasRouter configure collaboration from the input task, while AFlow supports intermediate-result branching through logic defined before execution (Zhang et al., 2024a; Yue et al., 2025; Zhang et al., 2024b). These methods do not treat collaboration construction as a general stage-level decision conditioned on evolving task states. ECHOROUTER instead constructs collaboration from intermediate states, allowing computation to adapt as execution unfolds.

3 ECHOROUTER

ECHOROUTER uses experience from past executions to generate candidate policies that guide adaptive collaboration during execution, as illustrated in Figure 1. We first define the search space and its three decoupled policy components in §3.1. We then describe the optimization round r in §3.2: reflection and credit assignment update Echo, which guides candidate proposals and forecasting before real rollouts. Finally, §3.3 describes task execution stage t, which the controller uses current execution state and policy to construct next collaboration and allocate computation adaptively.

3.1 CONDITIONAL POLICY SPACE OVER WORKFLOWS

Different task states warrant different amounts of work: a valid answer may stop after one stage, while a failed check may justify a new subgraph. For task x, execution-model pool $\mathcal{M}$, and mean Agent-cost budget B, we represent a policy as

$$\Theta = (P_{\text{meta}}, P_{\text{model}}, P_{\text{context}}). \tag{1}$$

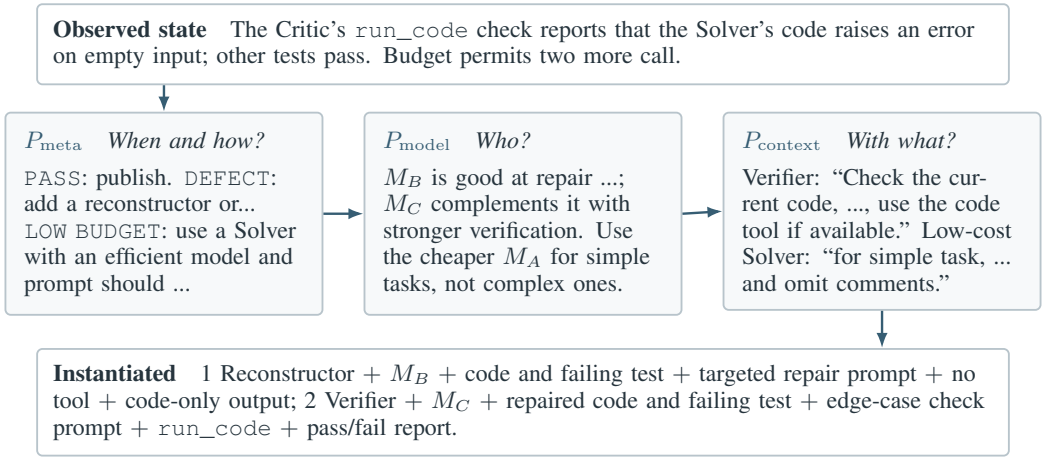

Figure 2: Given the current state and budget, the controller uses the three policy components to construct the next subgraph, assign models, and specify each call's context and output. The example adapts an archived policy (Appendix J); model aliases are schematic.

These three policy components are represented in a structured JSON policy and guide a lightweight Controller at runtime. Given the current observation o_t, the Controller uses P_{meta} to decide *when and how to collaborate*: whether to stop or create a bounded graph G_t, which roles and dependencies it contains, and which observed conditions activate each branch. For every selected role v, it uses P_{model} to decide *who executes it*, binding v to a model $m_v \in \mathcal{M}$ using current model profiles $\mathcal{P}$ and role-specific capability, complementary errors, and cost evidence. P_{context} then guides *how each selected role is presented to its model*. The first two components determine the next subgraph and assign a model to each of its roles:

$$G_t = P_{\text{meta}}(o_t), \qquad m_v = P_{\text{model}}(v, o_t, \mathcal{P}), \quad v \in G_t. \tag{2}$$

Given G_t and its model bindings, P_{context} guides the Controller in configuring each node as a model call. For each role v, the Controller selects which parts of the task state and predecessor artifacts m_v can see, supplies its execution prompt, grants the relevant tool interfaces, and specifies its output format. For example, a Critic may see candidate code and use `run_code`, while a Reconstructor sees the original code and the Critic's feedback and returns corrected code. Thus, different nodes in the same subgraph can receive different contexts and tools. Figure 2 follows one failed-check state through these decisions to the resulting repair call. Additional topologies are shown in Figure 9.

For target task distribution $\mathcal{D}$, we seek

$$\max_{\Theta} \mathbb{E}_{x\sim\mathcal{D}}[Q(x;\Theta)] \quad \text{s.t.} \quad \mathbb{E}_{x\sim\mathcal{D}}[C_{\text{A}}(x;\Theta)] \le B, \tag{3}$$

where Q is the predeclared quality metric and C_{A} is the execution cost at standardized token prices. The mean-cost constraint permits different spending across tasks without per-task hard truncation. This decomposition makes the policy's decisions clear. During reflection, ECHOROUTER can update only the relevant graph, model, or context choices while inheriting the rest from the parent policy.

3.2 FORECAST-GUIDED POLICY EVOLUTION

During evolution, ECHOROUTER treats a general-purpose LLM as a value model conditioned on Echo experience. Each round diagnoses the current policy, proposes M updates, forecasts their quality and expected calls, and estimates cost from those calls and model profiles. It then runs only K promising candidates, saving real rollouts on long-horizon tasks. The reported runs use $K = 2$. Appendix E gives the full loop and Appendix F.1 details its selection protocol.

Echo experience. The round-start Echo snapshot $\mathcal{E}_r$ combines three kinds of evidence. The first consists of model profiles $\mathcal{P}_r$ that record role- and task-specific quality, cost, and failures. The second consists of accepted policies and their measured quality–cost outcomes. The third consists

of reflections on execution trajectories, including intermediate states and the likely causes of failures or wasted calls. The optimizer and Controller see relevant model profiles rather than public model names. Controlled single-model calls initialize profiles, while training executions and comparisons update Echo only at round boundaries. A new model can enter through a fresh slot after comparison on saved task–role states, without rebuilding the collaboration policy.

Matched model and prompt comparisons. Comparisons replay saved task–role states to assess model and prompt alternatives. A model comparison changes only the anonymous model. A prompt comparison changes only the prompt. The task, upstream state, tools, and output contract stay fixed, so the observed difference is easier to attribute to the changed choice. Comparison outputs inform later policy updates and improve credit assignment.

For $R > 1$, the model-comparison activation rate decreases from 1 to 0.2 across rounds:

$$s_r = 1 - 0.8\frac{r-1}{R-1}. \tag{4}$$

For a selected Agent call a using model m_0, we retain cost-matched alternatives $\mathcal{M}_a$ and select the highest-scoring comparison model:

$$U_a(m) = \bar{v}_a(m) + \beta\sqrt{\frac{\log(N_a + |\mathcal{M}_a| + 2)}{n_a(m) + 1}} - \lambda_b \log\frac{\widehat{c}_a(m)}{\widehat{c}_a(m_0)}, \tag{5}$$

where $\bar{v}_a(m)$ is the profile value for the task type and role–tool slot, $n_a(m)$ is its evidence count, $N_a = \sum_m n_a(m)$, and $\widehat{c}_a(m)$ is the estimated call cost. The three terms favor measured capability, underexplored models, and lower cost, respectively. Appendix D details the comparison protocol, and Figure 8 shows candidate execution prompts.

Trajectory reflection. The optimizer reflects on the trajectories and comparison outcomes to identify errors, redundant calls, useful behaviors, and likely responsible policy components. This reflection guides candidate policy and prompt changes. Appendix D details the comparison protocol, and Figure 8 shows candidate execution prompts.

Proposals and forecasting. Guided by Echo, the optimizer proposes M diverse candidate policies $\mathcal{Z}_r$ by adding new choices or revising selected parts of the current policy Θ_r. Each candidate inherits the unchanged parts, allowing efficient generation of different strategies without rebuilding the policy from scratch. Before real rollout, the general-purpose LLM acts as a value model: it reasons over Echo to forecast each candidate's mean quality $\widehat{\mu}_z^Q$, heuristic uncertainty $\widehat{\sigma}_z^Q$, and the conditional activation rate $\widehat{\gamma}_{za}$ for each reachable call $a \in \mathcal{A}_z$. Quality and uncertainty form an optimistic exploration score:

$$U_z = \min\left(1, \widehat{\mu}_z^Q + \beta\widehat{\sigma}_z^Q\right), \qquad \beta = 1.28, \tag{6}$$

The exploration bonus favors uncertain candidates. Topology novelty, information value, and expected improvement inform selection. The program derives a cost proxy from reachable calls. For a call a with parents $\mathrm{Pa}(a)$, it approximates reach and marginal execution rates as

$$\widehat{r}_{za} = \begin{cases} 1, & \mathrm{Pa}(a) = \varnothing, \\ \min_{p\in\mathrm{Pa}(a)} \widehat{\rho}_{zp}, & \mathrm{Pa}(a) \neq \varnothing, \end{cases} \qquad \widehat{\rho}_{za} = \widehat{r}_{za}\widehat{\gamma}_{za}. \tag{7}$$

The smallest parent rate approximates joint reachability. Combining execution rates with model assignment gives the expected Agent cost:

$$\widehat{C}_z = \sum_{a\in\mathcal{A}_z} \widehat{\rho}_{za} \sum_{m\in\mathcal{M}} \widehat{\pi}_{zam}\bar{c}_{am}, \tag{8}$$

where $\widehat{\pi}_{zam}$ is the conditional model-assignment rate, and $\bar{c}_{am}$ is the task-relevant empirical standardized whole-call cost.

Rollout allocation and admission. Among candidates predicted to be cost-feasible, we allocate evaluation slots to those with high predicted mean quality or high U_z and structural novelty. Selected candidates first run on a rotating training minibatch. Candidates considered for admission are then evaluated on the same fixed validation set. Among those meeting the measured mean-cost constraint, the candidate with the highest measured quality becomes Θ_{r+1}, with lower Agent cost breaking ties. Execution trajectories, comparison results, costs, and forecast errors enter $\mathcal{E}_{r+1}$ and inform subsequent forecasts, as shown in Appendix Figure 11.

3.3 RUNTIME-ADAPTIVE EXECUTION

At test time, the Controller follows the learned policy to decide, from the current task state and remaining budget, whether to stop or construct the next collaboration subgraph. This stage-wise decision allocates computation according to the task's evolving needs. At stage t,

$$\begin{aligned} o_t &= (x, \xi_{0:t}, \mathcal{P}, b_t), \\ a_t &= \pi_\Theta(o_t) \in \{\mathrm{Construct}(G_t), \mathrm{Publish}(u_t)\}, \end{aligned} \tag{9}$$

where $\xi_{0:t}$ contains agent outputs, tool results, node statuses, and routing records, and b_t is a soft budget signal derived from B and cumulative cost. Equation 2 instantiates each selected node: the meta-policy determines the graph, the model policy binds its roles, and the context policy packages each model call. Execution updates state and cost:

$$\begin{aligned} (\xi_{t+1}, \Delta C_{\mathrm{A},t}) &= \mathrm{Execute}(G_t; \xi_{0:t}), \\ C_{\mathrm{A},t+1} &= C_{\mathrm{A},t} + \Delta C_{\mathrm{A},t}. \end{aligned} \tag{10}$$

After each subgraph, the Controller either publishes a node output via $\mathrm{Publish}(u_t)$ or updates ξ_{t+1} with the new outputs and statuses before constructing the next subgraph.

4 EXPERIMENTS

Experimental settings. We evaluate quality and cost on mathematics, coding, and general reasoning. The benchmarks are AIME, MATH, and GSM8K for mathematics, LiveCodeBench v5 and HumanEval for coding, and DROP, GPQA-Diamond, and ARC-Challenge for general reasoning. All methods run without external tools on these three task families. Each optimization method runs its native procedure for 10 steps on the same mixed-difficulty optimization set, followed by frozen-policy evaluation on the same held-out test set. For ECHOROUTER, Low (L) and Medium (M) target mean budgets of \$0.006 and \$0.012. All methods share the same optimization steps, maximum token budget and retry strategy, with each baseline retaining its original algorithm for fair comparison. and we report the realized optimization and test-time costs of each method. The small-pool comparison includes GPTSwarm, AFlow, MASRouter, EvoMAS (Hu et al., 2026), and ECHOROUTER. The large-pool comparison additionally includes direct-model and single-model routing baselines. DSQA, HLE Verified, and BFCL V3 MT permit task-specific tools. Tool configurations and other settings are provided in Appendix B. Controller training is described in Appendix G.

Metrics. We report task score, one-time optimization cost, and mean test-time Agent cost per query, excluding local Controller overhead. DROP uses F_1, coding uses test-case success, and Avg. is the unweighted mean of the eight metrics, with AIME averaged over 2024 and 2025. Because both policy instructions and Controller outputs are structured, we use speculative decoding to accelerate local inference. Across benchmarks, cumulative Controller request time averages 1.21 s per query and incurs no external API charge. Details are provided in Appendix H.

4.1 MAIN RESULTS

ECHOROUTER leads across model pools. In the small pool, ECHOROUTER attains the highest average score, although its optimization cost is higher than those of the MAS baselines (Table 1). With access to the larger pool, its policy improves especially on AIME, LCB, and GPQA-D and achieves the highest average score among direct-model, standalone routing, and MAS baselines (Table 2). Overall, ECHOROUTER achieves the strongest average performance in both model pools, demonstrating that its policy optimization remains effective as the set of available models expands. Compared with EvoMAS in the larger-pool setting, its Medium Policy has 63.6% lower mean test agent cost per query at nearly the same optimization cost. ECHOROUTER lies on the empirical quality–cost Pareto frontier in both pools (Figure 10). Its state-conditioned execution can vary the number of Agent calls across tasks even when the policy is optimized for a task family. Call-depth patterns, run-level variation, and detailed costs are reported in Figure 6 and Tables 3 and 4.

ECHOROUTER enables efficient policy evolution on complex tasks. DSQA, HLE Verified, and BFCL V3 MT involve long-context retrieval, challenging question answering, or multi-turn function

Table 1: Performance across eight benchmarks using the small model pool. Run-level standard deviations are reported in Appendix Table 3.

Method	Mathematical			Coding		General			Avg.	Cost	
	AIME	MATH	GSM8K	LCB-v5	HumanEval	DROP	GPQA-D	ARC-C	Score ↑	Opt. ↓	Inf. (USD) ↓
Multi-Agent System: Eight-Model Pool											
GPTSwarm (ICML'24)	72.9	87.5	93.8	**75.0**	91.7	81.3	70.0	93.3	83.2	**0.13**	0.0013
AFlow (ICLR'25)	70.8	87.5	93.8	66.7	87.5	87.8	**73.3**	**94.5**	82.7	0.19	**0.0012**
MASRouter (ACL'25)	60.4	79.2	89.6	56.5	92.5	83.6	68.1	92.1	77.8	0.15	0.0035
EvoMAS (ICML'26)	68.8	87.5	93.8	55.3	**93.1**	80.1	65.3	92.8	79.6	0.18	0.0017
Ours: Eight-Model Pool											
EchoRouter (M)	**77.1**	**89.6**	**95.8**	73.3	91.7	**89.5**	70.0	93.3	**85.0**	0.23	0.0014

Notes. Scores are three-run means, rounded to one decimal. AIME averages AIME 2024 and AIME 2025. Avg. is the unweighted mean over the eight benchmarks. Opt. is the one-time optimization cost per benchmark in USD. Inf. is the mean test-time Agent cost per query in USD and excludes local Controller computation. Controller call counts and request-time estimates are reported in Appendix H. GPTSwarm and AFlow use a fixed GPT-OSS-120B executor in these matched-pool runs.

Table 2: Performance across eight benchmarks using the large model pool. Run-level standard deviations are reported in Appendix Table 4.

Method	Mathematical			Coding		General			Avg.	Cost	
	AIME	MATH	GSM8K	LCB	HumanEval	DROP	GPQA-D	ARC-C	Score ↑	Opt. ↓	Inf. (USD) ↓
Direct Model Baselines											
Random	45.3	62.7	88.4	60.3	87.5	79.6	48.9	84.2	69.6	—	0.0041
Ensemble	57.7	78.3	93.2	67.8	95.6	89.5	73.2	93.8	81.1	—	0.0143
Greatest	70.9	91.7	94.4	75.2	97.1	91.0	76.5	94.2	86.4	—	0.0270
Standalone Model Routing											
MLPRouter (LLMRouter)	26.8	63.1	89.5	22.8	95.2	84.6	60.3	85.3	66.0	1.06	**0.0012**
GraphRouter (ICLR'25)	35.1	11.8	92.3	32.8	95.6	92.3	55.3	**94.8**	63.8	1.13	0.0109
HybridLLM (ICLR'24)	30.5	70.8	86.2	72.3	91.0	92.4	61.8	87.3	74.0	1.01	0.0121
Multi-Agent System											
GPTSwarm (ICML'24)	43.3	78.6	95.9	63.6	97.5	80.2	80.8	94.2	79.3	**0.27**	0.0804
AFlow (ICLR'25)	51.6	84.9	**97.0**	68.3	**98.4**	79.5	81.9	94.6	82.0	0.64	0.0090
MASRouter (ACL'25)	46.6	67.0	86.4	62.8	92.5	82.6	72.0	91.8	75.2	0.37	0.0102
EvoMAS (ICML'26)	70.4	94.2	96.2	85.2	98.1	82.4	80.3	93.6	87.6	1.03	0.0316
Ours											
EchoRouter (L)	73.5	88.4	91.8	71.5	88.5	90.2	73.0	91.3	83.5	0.50	0.0056
EchoRouter (M)	**81.8**	**95.6**	96.5	**85.8**	95.6	**93.2**	**82.4**	94.5	**90.7**	1.04	0.0115

Notes. Scores are three-run means, rounded to one decimal. AIME averages AIME 2024 and AIME 2025. Avg. is the unweighted mean over the eight benchmarks. Opt. is the one-time optimization cost per benchmark in USD. Inf. is the mean test-time Agent cost per query in USD and excludes local Controller computation. Controller call counts and request-time estimates are reported in Appendix H. GPTSwarm and AFlow use GPT-4o as their fixed executor. "—" denotes no optimization.

calls, making real rollouts costly. As shown in Figure 3, EchoRouter reaches EvoMAS's final score by optimization steps 5, 7, and 5, respectively, and ultimately outperforms EvoMAS on all three tasks. At these checkpoints, its mean test-time Agent API cost per query is also lower than that of EvoMAS's final policy. These results show that EchoRouter not only improves final policy quality but also discovers competitive, lower-cost policies earlier in optimization. The trajectory markers attribute these improvements to updates of the meta, model, and context policies, while Figure 5 further confirms the contribution of matched model and prompt comparisons. Representative conditional topologies, the BFCL state gate, and additional protocol details are provided in Figures 7 and 9 and Appendices B.1 and C.

EchoRouter supports adaptive execution on generation tasks. We then optimize across websites, visual apps, slides, games, SVGs, and documents, where there is no unique reference answer and the output must be a usable artifact. Our runtime adaptive execution methods support paying more computation for iterative refinement. The slide-deck lineage in Appendix Figure 13 illustrates such multi-round refinement. On held-out tasks, three LLM judges score the frozen policy against three direct-model baselines across six quality dimensions. As shown in Figure 4, EchoRouter remains close to Gemini 3.5 Flash across dimensions; however, its mean cost per artifact is less than one-quarter of Gemini's. It also scores above the cheaper Claude Haiku 4.5

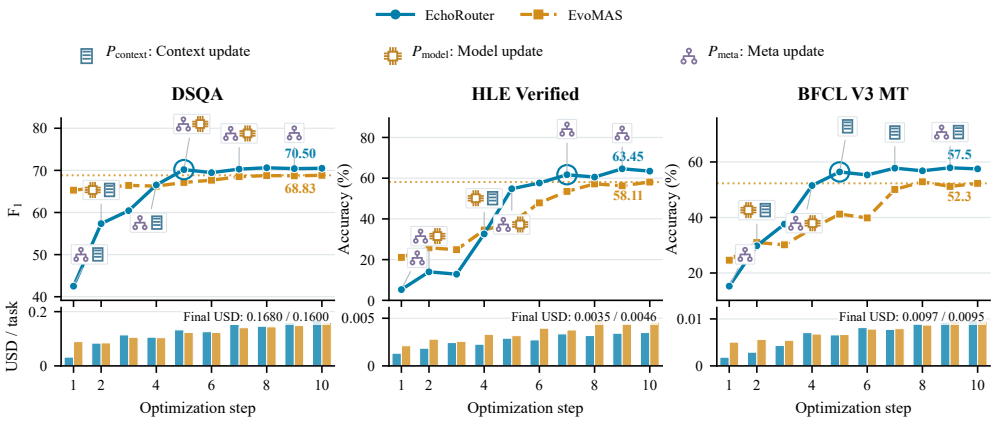

Figure 3: Measured policy evolution on DSQA, HLE Verified, and BFCL V3 MT. The curves show policy scores at successive optimization steps, and the bars show mean Agent cost per query at the corresponding checkpoints. The dotted line marks EvoMAS's final score in each panel. Icons indicate policy-component updates.

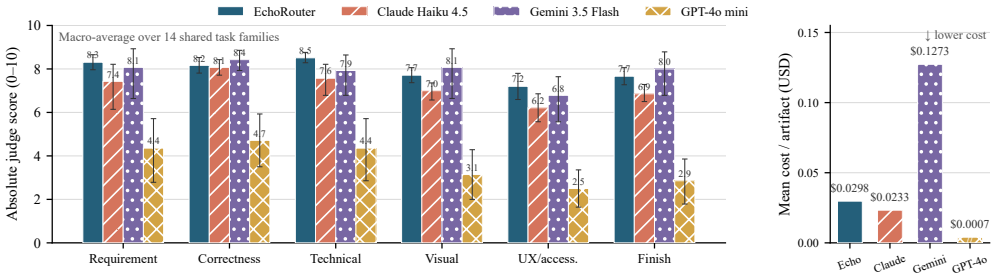

Figure 4: Quality and cost on open-ended generation tasks. The left panel reports the average of three LLM-judge scores across six dimensions, and the right panel reports the mean artifact cost. Each generation task was repeated three times.

on all dimensions. These results extend the observed quality–cost trade-off to open-ended artifact generation. The evaluation protocol and further examples appear in Appendix B.3, Figures 12 and 14.

4.2 Ablation Studies

Each ablation isolates forecast-guided search, trajectory-based comparisons, anonymous profiles, or runtime adaptation. Each study runs for 10 optimization steps on mathematics, coding, and general reasoning. After optimization, we evaluate the frozen policy saved at each step on the same held-out test set (Figure 5). Because these questions are reused, the checkpoint trajectories are descriptive. Variant definitions and shared settings are provided in Appendix B.2.

A stronger optimizer improves optimization efficiency. Replacing GPT-5.6 Luna with GPT-5.6 Sol raises the trajectory-wide mean score from 82.34 to 86.01. Across the three task families, Sol generally reaches high scores earlier and attains a higher performance ceiling. A stronger optimizer therefore improves both convergence speed and the quality reached by policy evolution.

Model and prompt comparisons are important. We find that removing comparisons reduces the method's ability to assess model and prompt alternatives. The score drops by 10.11 percentage points and shifts the curves to a lower range. During real training-task executions, these comparisons change only the model or prompt while holding the task, role, upstream state, tools, and output contract fixed. Their measured outcomes inform profile updates and later candidate revisions. The ablation measures the full comparison-and-update path, including its downstream effects.



Figure 5: Ten-step ablation trajectories on mathematics, coding, and general reasoning. The lines report the held-out test scores of the frozen policies saved at each optimization step, and the lower grouped bars report their normalized evaluation costs. After all optimization runs were complete, we evaluated these checkpoints on the same held-out test set. Test outcomes were not available during optimization or policy selection. Because the same test items are reused across checkpoints, the trajectories are descriptive.

Runtime adaptation improves the quality–cost trade-off. The static variant follows a fixed workflow throughout each execution, whereas the adaptive variant selects subsequent subgraphs based on intermediate outputs. Although the static variant leads at some early checkpoints, the adaptive variant achieves a higher mean score (86.01 versus 83.59) and a lower mean normalized evaluation cost (0.976 versus 1.052) across the ten-step trajectories, as shown in Figure 5. These results support allocating subsequent Agent calls according to the observed execution state.

Anonymous model identifiers reduce reliance on name-based priors. Under matched Luna conditions, the anonymous variant achieves a trajectory-wide mean score of 82.34, compared with 80.80 when model names are exposed. The corresponding normalized costs are 0.977 and 1.007. In the early optimization traces, public names sometimes influenced model selection or failure attribution despite contrary measured evidence. Appendix D.1 documents examples on AIME and GPQA-D. Together, the ablation and these traces suggest that anonymous profiles help the optimizer rely on observed model behavior when capability evidence is sparse.

4.3 Practicality and Transfer

We evaluate practicality from three perspectives. Forecast-guided selection outperforms random with all four value models on frozen Best-of-2 pools, while stronger forecasters generally perform better. Notably, Jev approaches DeepSeek V4 Flash at 39% lower ranking cost, showing that search acceleration is not tied to a specific value model (Table 5). After policy optimization is frozen, a new model only requires matched comparisons on saved trajectories to construct its anonymous profile and compete with existing models without rebuilding the policy (Appendix K). Moreover, initializing from an EvoMAS policy and applying one round of EchoRouter refinement improves performance in mathematics, coding, and general reasoning (Table 7). Together, these results demonstrate transfer across value models, model pools, and initial MAS policies. As general-purpose value models improve, such flexibility may facilitate the shift toward state-conditioned dynamic MAS construction while enabling rapid integration of new models and existing policies.

5 Conclusion

We propose EchoRouter, which uses past experience to generate candidate policies that guide the adaptive construction of collaboration during execution. Rather than committing to a fixed workflow, it constructs collaboration stage by stage and allocates computation according to task progress. Across diverse tasks, EchoRouter achieves favorable quality–cost trade-offs. These results demonstrate the effectiveness of our method, and runtime-adaptive collaboration remains a promising future direction.

AI USE STATEMENT

The authors used generative AI tools to assist with reading and understanding related literature and as coding assistants during implementation. The authors checked the relevant literature against the original sources and reviewed and tested AI-assisted code. The manuscript was written by the authors. The authors take responsibility for all content, claims, experimental results, and artifacts in this paper.

REPRODUCIBILITY STATEMENT

The paper describes the EchoRouter algorithm, including policy optimization, candidate evaluation, and task-time execution. It also specifies the experimental setup and the training and evaluation procedures needed to reproduce the reported results. Details of controller fine-tuning, along with the relevant configurations and implementation choices.

REFERENCES

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

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors, 2023b. URL https://arxiv.org/abs/2308.10848.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate, 2023. URL https://arxiv.org/abs/2305.14325.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for multi-agent collaborative framework, 2023. URL https://arxiv.org/abs/2308.00352.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems, 2024. URL https://arxiv.org/abs/2408.08435.

Yuntong Hu, Yuting Zhang, Matthew Trager, Yi Zhang, Shuo Yang, Wei Xia, and Stefano Soatto. EvoMAS: Evolutionary generation of multi-agent systems. In *International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2602.06511.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society, 2023. URL https://arxiv.org/abs/2303.17760.

Yu Li, Lehui Li, Zhihao Wu, Qingmin Liao, Jianye Hao, Kun Shao, Fengli Xu, and Yong Li. AgentSwift: Efficient LLM agent design via value-guided hierarchical search. In *AAAI Conference on Artificial Intelligence*, 2026. URL https://arxiv.org/abs/2506.06017.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. A dynamic LLM-powered agent network for task-oriented agent collaboration, 2023. URL https://arxiv.org/abs/2310.02170.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development, 2023. URL https://arxiv.org/abs/2307.07924.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. AgentSquare: Automatic LLM agent search in modular design space, 2024. URL https://arxiv.org/abs/2410.06153.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023. URL https://arxiv.org/abs/2305.16291.

Zhexuan Wang, Yutong Wang, Xuebo Liu, Liang Ding, Miao Zhang, Jie Liu, and Min Zhang. Agentdropout: Dynamic agent elimination for token-efficient and high-performance llm-based multi-agent collaboration. *arXiv preprint arXiv:2503.18891*, 2025.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation, 2023. URL https://arxiv.org/abs/2308.08155.

Jingjun Xu, Hongji Pu, Tao Feng, Haozhen Zhang, Jiaxuan You, and Ge Liu. Routeprofile: Graph-based profiling for cold-start llm routing, 2026.

Yanwei Yue, Guibin Zhang, Boyang Liu, Guancheng Wan, Kun Wang, Dawei Cheng, and Yiyan Qi. Masrouter: Learning to route llms for multi-agent systems, 2025.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. G-Designer: Architecting multi-agent communication topologies via graph neural networks, 2024a. URL https://arxiv.org/abs/2410.11782.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet, 2025a. URL https://arxiv.org/abs/2502.04180.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Xu Yu, and Tianlong Chen. Cut the crap: An economical communication pipeline for LLM-based multi-agent systems. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=LkzuPorQ5L.

Guibin Zhang, Haiyang Yu, Kaiming Yang, Bingli Wu, Fei Huang, Yongbin Li, and Shuicheng Yan. EvoRoute: Experience-driven self-routing LLM agent systems, 2026. URL https://arxiv.org/abs/2601.02695.

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, 2024b.

Yuanshuo Zhang, Yuchen Hou, Bohan Tang, Shuo Chen, Muhan Zhang, Xiaowen Dong, and Siheng Chen. GNNs as predictors of agentic workflow performances, 2025c. URL https://arxiv.org/abs/2503.11301.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Gptswarm: Language agents as optimizable graphs. In *International Conference on Machine Learning*, 2024.

A Detailed Main Results and Visualizations

Tables 3 and 4 add run-level standard deviations to the task-level means in main-text Tables 1 and 2.

Table 3: Small-pool results from Table 1 with run-level standard deviations for average score and Agent API cost.

Method	Mathematical			Coding		General			Avg.	Cost	
	AIME	MATH	GSM8K	LCB-v5	HumanEval	DROP	GPQA-D	ARC-C	Score ↑	Opt. ↓	Inf. (USD) ↓
Multi-Agent System: Eight-Model Pool											
GPTSwarm (ICML'24)	72.9	87.5	93.8	**75.0**	91.7	81.3	70.0	93.3	83.2 ±1.9	**0.13** ±0.14	0.0013 ±0.0007
AFlow (ICLR'25)	70.8	87.5	93.8	66.7	87.5	87.8	**73.3**	**94.5**	82.7 ±2.1	0.19 ±0.11	**0.0012** ±0.0005
MASRouter (ACL'25)	60.4	79.2	89.6	56.5	92.5	83.6	68.1	92.1	77.8 ±3.7	0.15 ±0.27	0.0035 ±0.0018
EvoMAS (ICML'26)	68.8	87.5	93.8	55.3	**93.1**	80.1	65.3	92.8	79.6 ±2.3	0.18 ±0.31	0.0017 ±0.0008
Ours: Eight-Model Pool											
EchoRouter (M)	**77.1**	**89.6**	**95.8**	73.3	91.7	**89.5**	70.0	93.3	**85.0** ±2.2	0.23 ±0.25	0.0014 ±0.0005

Notes. Scores are averaged over three runs and rounded to one decimal place. AIME averages AIME 2024 and AIME 2025, and Avg. is the unweighted mean over the eight displayed benchmarks. Opt. reports the average one-time optimization cost per benchmark in USD. Inf. reports the average test-time Agent cost per query in USD. In the three summary columns, the first line is the mean and the second line is the run-level standard deviation. Router overhead is excluded from Inf.

Table 4: Large-pool results from Table 2 with run-level standard deviations for average score and Agent API cost.

Method	Mathematical			Coding		General			Avg.	Cost	
	AIME	MATH	GSM8K	LCB	HumanEval	DROP	GPQA-D	ARC-C	Score ↑	Opt. ↓	Inf. (USD) ↓
Direct Model Baselines											
Random	45.3	62.7	88.4	60.3	87.5	79.6	48.9	84.2	69.6 ±1.3	—	0.0041 ±0.0024
Ensemble	57.7	78.3	93.2	67.8	95.6	89.5	73.2	93.8	81.1 ±3.8	—	0.0143 ±0.0030
Greatest	70.9	91.7	94.4	75.2	97.1	91.0	76.5	94.2	86.4 ±2.5	—	0.0270 ±0.0017
Standalone Model Routing											
MLPRouter (LLMRouter)	26.8	63.1	89.5	22.8	95.2	84.6	60.3	85.3	66.0 ±3.9	1.06 ±0.21	**0.0012** ±0.0007
GraphRouter (ICLR'25)	35.1	11.8	92.3	32.8	95.6	92.3	55.3	**94.8**	63.8 ±5.3	1.13 ±0.13	0.0109 ±0.0034
HybridLLM (ICLR'24)	30.5	70.8	86.2	72.3	91.0	92.4	61.8	87.3	74.0 ±2.1	1.01 ±0.14	0.0121 ±0.0039
Multi-Agent System											
GPTSwarm (ICML'24)	43.3	78.6	95.9	63.6	97.5	80.2	80.8	94.2	79.3 ±4.7	**0.27** ±0.12	0.0804 ±0.0355
AFlow (ICLR'25)	51.6	84.9	**97.0**	68.3	**98.4**	79.5	81.9	94.6	82.0 ±2.2	0.64 ±0.26	0.0090 ±0.0041
MASRouter (ACL'25)	46.6	67.0	86.4	62.8	92.5	82.6	72.0	91.8	75.2 ±3.0	0.37 ±0.13	0.0102 ±0.0036
EvoMAS (ICML'26)	70.4	94.2	96.2	85.2	98.1	82.4	80.3	93.6	87.6 ±2.1	1.03 ±0.23	0.0316 ±0.0142
Ours											
EchoRouter (L)	73.5	88.4	91.8	71.5	88.5	90.2	73.0	91.3	83.5 ±3.5	0.50 ±0.14	0.0056 ±0.0024
EchoRouter (M)	**81.8**	**95.6**	96.5	**85.8**	95.6	**93.2**	**82.4**	94.5	**90.7** ±2.6	1.04 ±0.35	0.0115 ±0.0087

Notes. Scores are averaged over three runs and rounded to one decimal place. AIME averages AIME 2024 and AIME 2025, and Avg. is the unweighted mean over the eight displayed benchmarks. Opt. reports the average one-time optimization cost per benchmark in USD. Inf. reports the average test-time Agent cost per query in USD. In the three summary columns, the first line is the mean and the second line is the run-level standard deviation. Router overhead is excluded from Inf. "—" means no optimization is performed.

Table 5: Selection performance when LLMs serve as general-purpose value models on frozen candidate pools. Gain is measured relative to random selection. Cost covers one selector pass and excludes candidate execution.

Selector	Best-of-2 ↑	Gain (pp)	Cost ($) ↓
Random	83.93	0.00	0
Forecast-guided candidate selection			
GPT-OSS-20B	88.65	+4.72	**0.000878**
Luna (low)	92.32	+8.39	0.012080
DeepSeek V4 Flash	**92.84**	**+8.91**	0.007131
Jev (pairwise)	92.11	+8.18	0.004350

Generation Tasks and Comparison Setup. The shared generation policy operates on a single normalized artifact that its nodes iteratively create, inspect, and revise. A Publish node exports the final artifact as a webpage, image, slide deck, document, or other requested format. GPT-5.6 Luna, Gemini-2.5 Flash and DeepSeek V4 Flash evaluate requirement adherence, correctness and completeness, technical validity, visual hierarchy, user experience and accessibility, and production readiness. We compare against three direct-model baselines: Claude Haiku 4.5, Gemini 3.5 Flash, and GPT-4o mini. As shown in Figure 4, ECHOROUTER remains competitive with Gemini 3.5 Flash on the remaining three reported dimensions. The complete evaluation protocol is provided in Appendix B.3. Representative outputs and their revision trajectories are presented in Figures 12, 13, and 14. An example generation policy graph is shown in Figure 7(f).

B EXPERIMENTAL SETTINGS AND PROTOCOLS

Data Construction and Isolation. The mathematics optimization set combines examples from AIME 2022/2023, MATH, and GSM8K. The coding tasks use LiveCodeBench v5 and HumanEval, while the general reasoning tasks use DROP, GPQA-Diamond, and ARC-Challenge. Policy optimization, profile updates, prompt updates, forecast generation, and candidate selection are conducted exclusively on the optimization split. After optimization, each frozen policy is evaluated independently on the held-out test split.

Budgets and Model Pools. The main experiments consider Low (L) and Medium (M) mean Agent-cost budgets of $0.006 and $0.012 per query, respectively. GPTSwarm, AFlow, MASRouter, EvoMAS, and ECHOROUTER share the same small model pool to support a controlled comparison. A separate large-pool evaluation additionally includes direct-model and standalone-routing baselines. Exact model identifiers, checkpoints, price-accounting rules, and other implementation settings are reported in Appendix I.

B.1 TOOL CONFIGURATIONS

In the main comparison across eight benchmarks, all methods operate in a text-only setting for mathematics, coding, and general reasoning. Separate tool-enabled coding analyses may use the deterministic sandboxed interfaces `run_code` and `run_tests` to verify candidate solutions, but do not permit web search. DeepSearchQA provides `web_search` to search-oriented roles, whereas BFCL V3 MT exposes only its native multi-turn function interface. The six-task HLE validation policy shown in Figure 7 is also text-only. Both frozen Agents use `tools: []`, a 600-second execution timeout, and no retries. In the separate frozen Extra-High EvoMAS configuration for HLE, `web_search` is available only to general and research workers. Each CodeAgent has a 900-second execution timeout and may retry once. Python and other external tools are not added to the frozen graph.

B.2 ABLATION PROTOCOL

Each ablation runs for 10 optimization steps on mathematics, coding, and general reasoning. After optimization, we evaluate the frozen policy saved at each step on the same held-out test set

Table 6: Execution-model pools used in the experiments. The local Qwen3.5-9B Runtime Controller is listed only when it is an execution-model candidate. Controller calls are accounted for separately from Agent cost.

API model identifier	small pool	Large pool
`mistralai/mistral-small-2603`	✓	✓
`mistralai/ministral-3b-2512`	✓	✓
`openai/gpt-oss-20b`	✓	✓
`openai/gpt-oss-120b`	✓	✓
`xiaomi/mimo-v2.5`	✓	✓
`xiaomi/mimo-v2.5-pro`	✓	–
`qwen/qwen3.5-9b`	✓	–
`deepseek/deepseek-chat-v3-0324`	✓	✓
`google/gemini-3.5-flash-lite`	–	✓
`google/gemini-3.5-flash`	–	✓
`openai/gpt-4o-mini`	–	✓
`openai/gpt-4o`	–	✓
`openai/gpt-5.4-nano`	–	✓
`anthropic/claude-sonnet-4.6`	–	✓
`anthropic/claude-haiku-4.5`	–	✓
`openai/gpt-4o-mini-2024-07-18`	–	✓
`google/gemini-2.5-flash`	–	✓
`meta-llama/llama-3.1-70b-instruct`	–	✓

(Figure 5). Because the same questions are reused across checkpoints, comparisons between checkpoints are interpreted descriptively. The public-name and random-selection variants use GPT-5.6 Luna, with Full (Luna) serving as their matched reference. The optimizer comparison evaluates the same full method using GPT-5.6 Sol and GPT-5.6 Luna.

B.3 Open-Ended Generation Protocol

The generation suite covers websites, visual applications, slide decks, games, SVGs, and documents. All nodes operate on a shared normalized artifact by creating, inspecting, or revising it. The Publish node then exports the artifact to the requested target format. Because these tasks do not have a unique ground-truth answer, the shared policy is optimized for 10 steps and then frozen for evaluation. The reported results aggregate the 14 fine-grained task categories shared by all compared methods.

C Task-Conditional Structure Evolution

The policy language may expand into independent sources, specialists, and conditional verification, but it may also retain a single executor. We report both outcomes and distinguish the union of planned routes from calls realized by an individual item. Figure 6 gives the corresponding call-depth view.

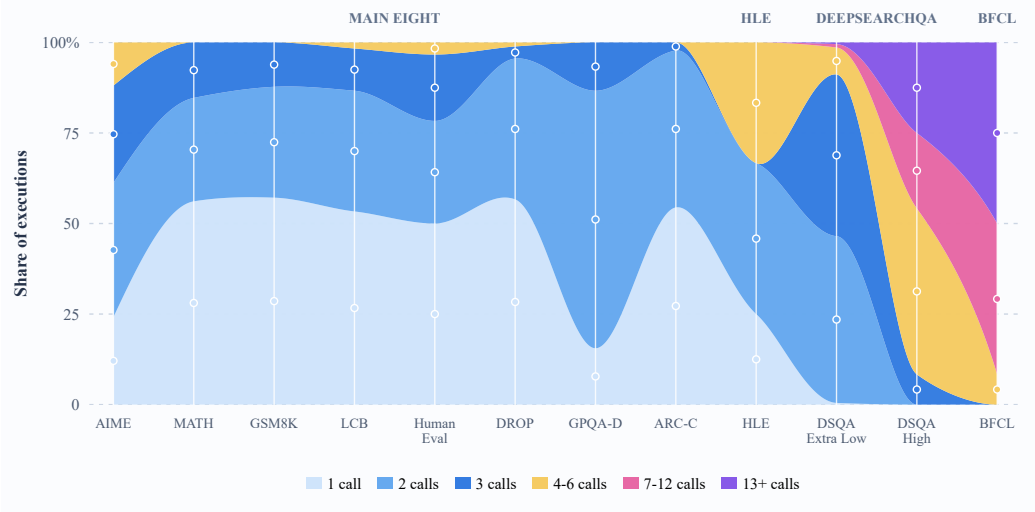

Figure 6: Illustrative Agent-call depth across representative tasks. Markers show task-level summaries. Bands between tasks are visual guides.

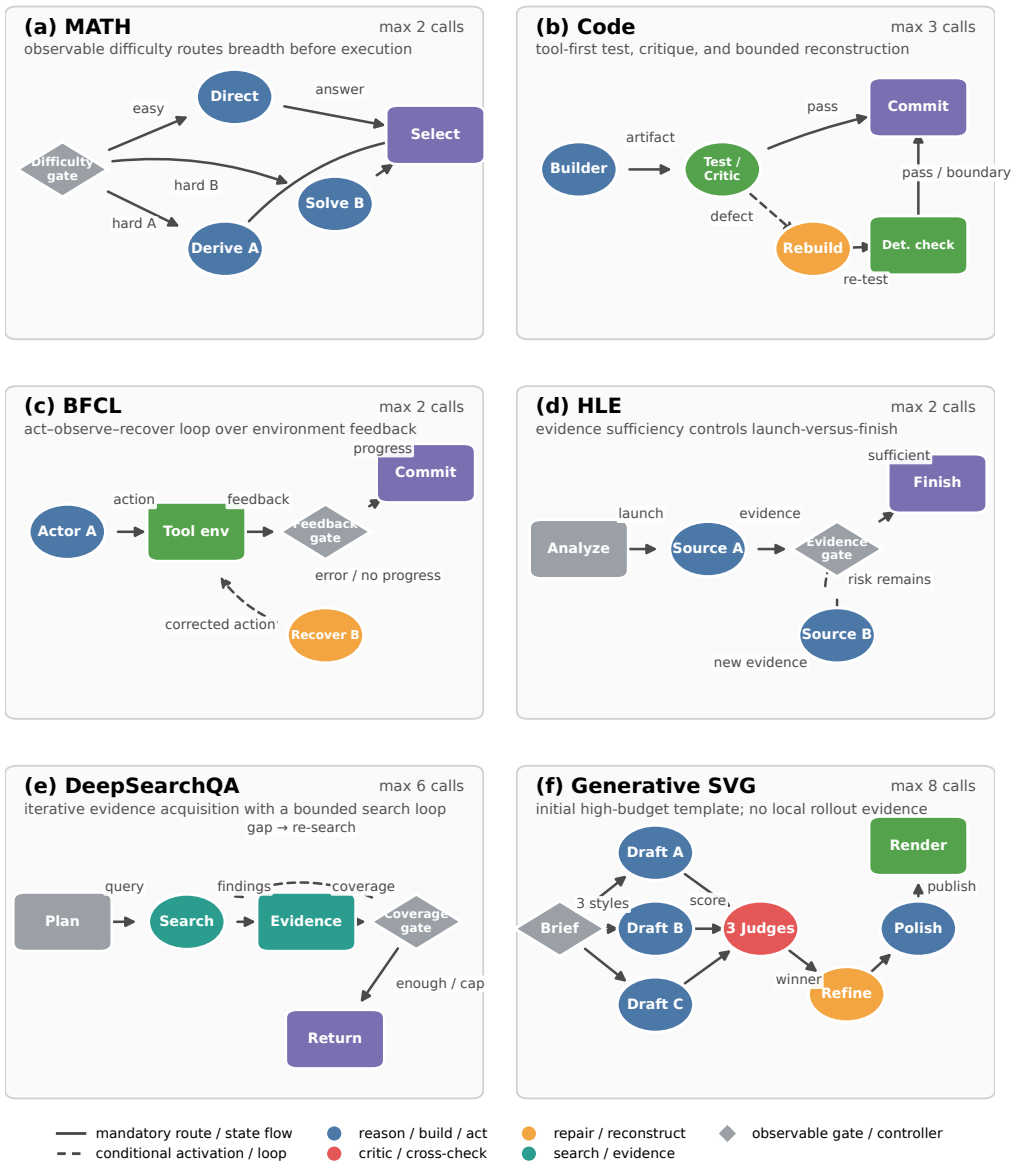

Figure 7: Examples of policy structures. (a) MATH routes by observable difficulty and expands the hard branch to two independent derivations. (b) Code adds tool-grounded diagnosis and bounded reconstruction. (c) BFCL alternates environment action, observation, and conditional recovery. (d) HLE launches a second evidence source only when the first leaves unresolved risk. This policy executed on six Medium validation tasks. (e) DeepSearchQA shows a six-turn bounded evidence-acquisition example. (f) The Generative SVG graph is a high-budget initialization template with three diverse drafts, three independent judges, refinement, and polishing. Diamonds and boxes denote controller/environment operations and do not count as Agent calls.

C.1 Execution Agent Prompt Candidates

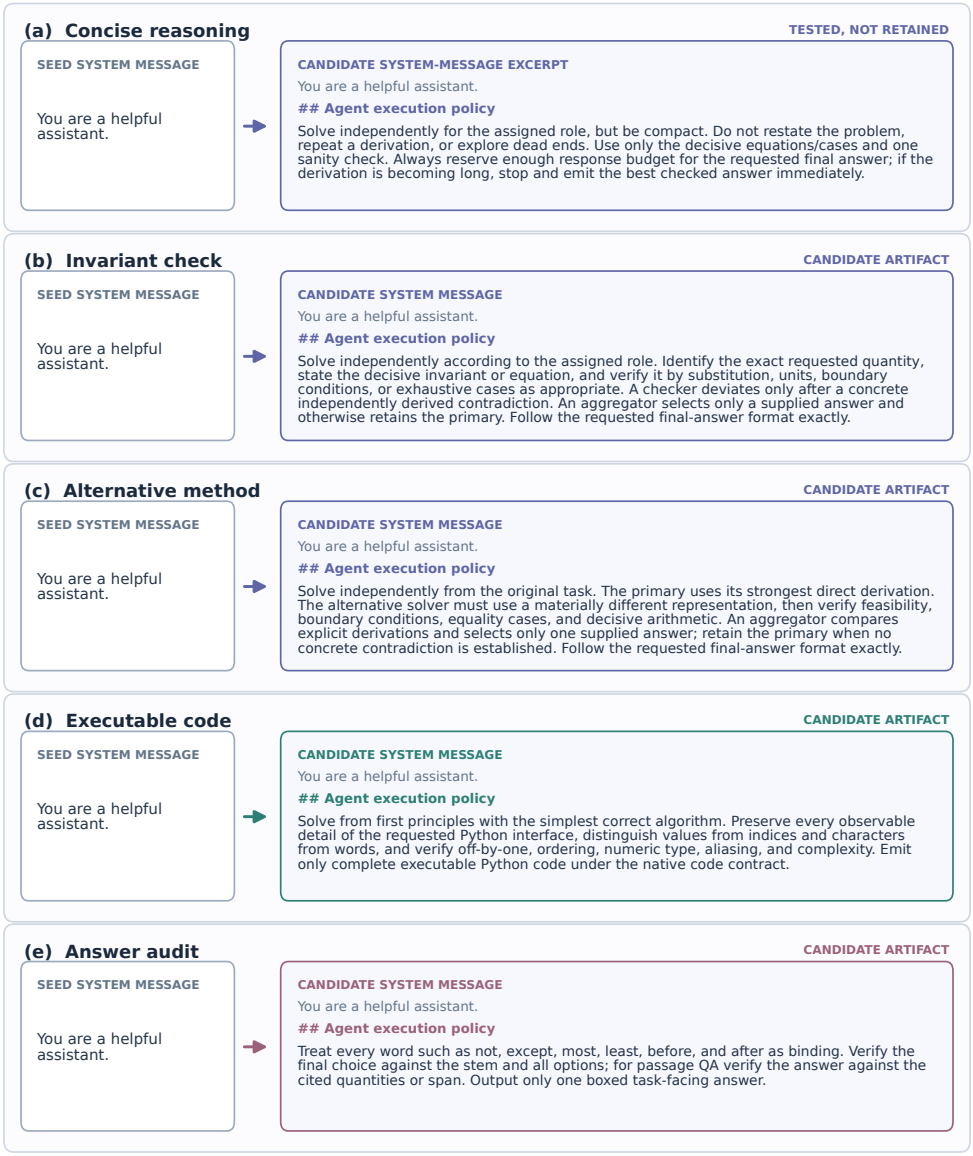

Figure 8: Dataset-neutral views of candidate execution system messages. Each right card joins the runtime's generic system line and `Agent execution policy` heading with a saved `agent_policy` candidate. Cards (b)–(e) reproduce that field in full. Card (a) is a contiguous prefix excerpt that omits the remaining checker/aggregator and benchmark-specific formatting instructions. The displayed instructions cover concise reasoning, invariant checking, alternative methods, executable code, and answer auditing. Task-specific user messages and output contracts are omitted. Candidate (a) was tested on eight optimization-selection items (0.625 versus 0.875 for its parent). The cards show prompt diversity, not an isolated prompt-only effect or five final adopted policies.

The `agent_policy` component is inserted into each execution Agent's system message under the `Agent execution policy` heading. The task, direct dependencies, and local objective appear in the user message. The output contract is appended separately to the system message. Figure 8 renders five saved candidates in that format. The left column is the runtime's generic default system

line. The shared `context_policy` controls which task and dependency artifacts are visible to an Agent. It is distinct from the optimized `agent_policy` block in these cards. Reflection-context compression applies to optimization-side diagnosis and should likewise not be interpreted as a change to execution Agent wording.

C.2 Refining an EvoMAS Policy with EchoRouter

Table 7 shows concrete changes to three frozen EvoMAS policies. Each before–after score comes from the same fixed validation set within its task family ($n = 120$ mathematics, $n = 120$ coding, and $n = 120$ general). None is an additional held-out test result.

Table 7: Observed validation-side refinement of imported EvoMAS policies. The refinement changes execution contracts and conditional routing, while retaining useful structure where its measured value remains positive.

Domain	Imported EvoMAS route	EchoRouter refinement	Validation
Mathematics	Four scheduled calls consisting of a solver, two verifiers, and a finalizer, all using the same anonymous model slot.	Retain the four-role graph while tightening task and output contracts. Normalize a semantically correct boxed answer deterministically. When disagreements occur, select the stronger model to provide the final recommendation.	85.0% $\rightarrow$ 90.0%
Coding	Four scheduled calls consisting of a solver, critic, verifier, and merger.	Select different primary and fallback model orders by task type. Invoke the fallback only when an artifact is missing, invalid, or affected by an infrastructure failure. Preserve valid executable code unless a specific check fails.	83.3% $\rightarrow$ 96.7%
General	Two independent solvers using the same model slot, followed by a scheduled verifier for every item.	Use task-type-specific routes. Normalize answers before voting, verify conditionally, and adjudicate only when there is a genuine disagreement or no majority.	62.50% $\rightarrow$ 72.68%

The mathematics refinement spends additional Agent cost on the same graph. The general-reasoning refinement reduces observed mean cost by making verification conditional. Thus, the intervention is not uniformly an extra-call ensemble.

C.3 Negative and Degenerate Cases

Collaboration is beneficial only when the available models have complementary errors and the task offers useful evidence for checking or repair. Otherwise, extra sources can repeat the same mistake, introduce incompatible intermediate answers, or cause an aggregator to replace a correct first answer. These are additional error channels as well as additional costs. In our smaller eight-model setting, some multi-Agent candidates regressed relative to a single-executor route, and the coding search ultimately retained one direct Solver. In the larger model-pool setting, the retained AIME policy did not show the same degeneration. This contrast is consistent with pool-dependent complementarity, but because the pools and policies differ it is not a controlled estimate of the effect of pool size alone.

Budget alone is also not a guarantee of higher accuracy: on the same 30 AIME 2024 items, representative Extra-Low, Medium, and Extra-High policies used 1.07, 2.90, and 7.63 calls per item, respectively, yet all scored 25/30. The method therefore treats early stopping and one-Agent execution as valid outcomes when the measured marginal value of coordination is non-positive.

The topology gallery and prompt cards use optimization and validation examples. The EvoMAS refinement scores in Table 7 come from the fixed-validation comparison described above.

C.4 State-Conditioned Policies

Figure 9 presents policies as observable-state controllers rather than static collections of Solver and Verifier roles. A topology is expanded, repaired, or stopped only after the runtime observes a relevant state transition in the task history, artifact, tool feedback, or evidence set.

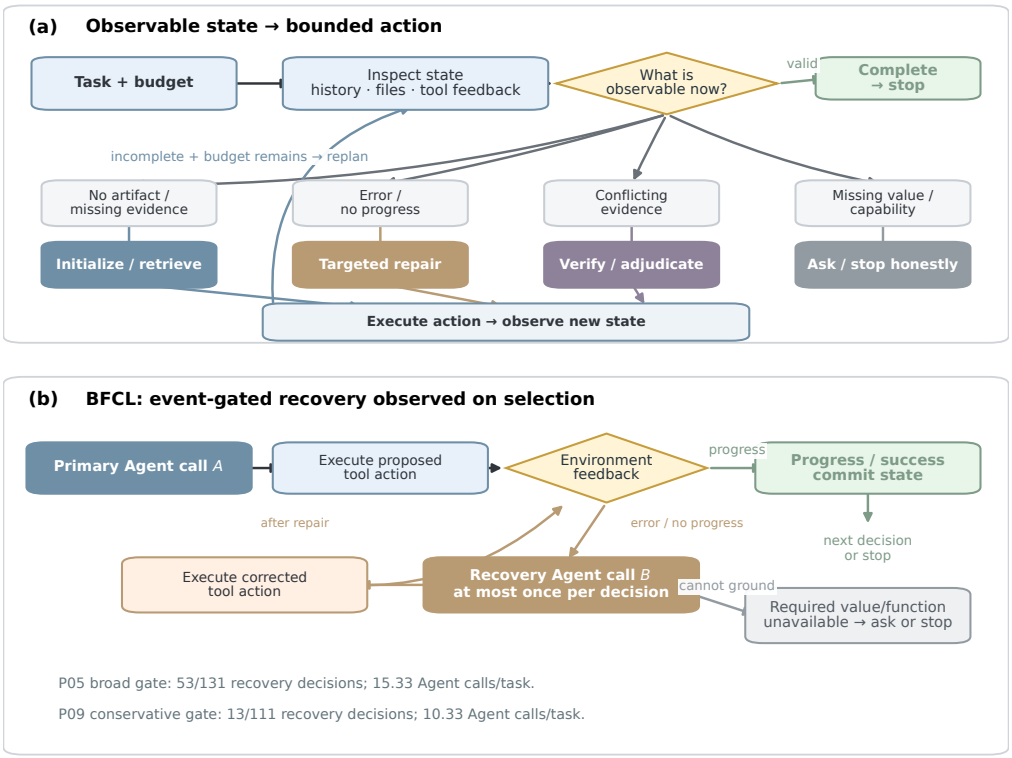

Figure 9: State-conditioned routing backed by executable source contracts and selection-side traces. (a) The controller maps observable states to bounded actions and replans only after the state changes. (b) A BFCL conditional-cascade candidate normally uses a primary Agent call to propose a tool action. It activates at most one recovery Agent call per environment decision after an observable trigger. The displayed activation and call counts are computed from the fixed selection rows.

The BFCL panel exposes two distinct optimized behaviors. The broad P05 gate activates its recovery slot after execution error, missing-parameter, no-progress, multi-action, or later-step evidence. The conservative P09 gate preserves the same native path but activates recovery only after an explicit execution error. Thus "multi-Agent" describes a conditional control action, with different cost paid at every environment decision. The P05 and P09 rates use policy-specific decision counts (131 and 111, respectively) from the same 12 selection tasks, so they are behavioral summaries rather than same-denominator precision estimates.

C.5 Quality–Cost Pareto Views of the Main Result Tables

Figure 10 re-expresses the two main result tables as quality–cost operating points. The horizontal axis is the reported mean test-time Agent cost per query. One-time optimization cost is deliberately not included. A point is empirically non-dominated when no displayed method has both lower or equal inference cost and higher or equal average score, with one strict inequality.

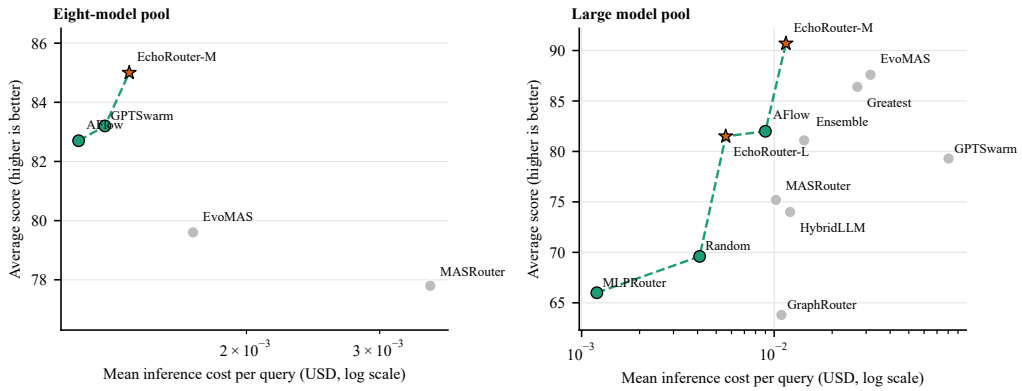

Figure 10: Quality–cost Pareto views reconstructed directly from the two main-result tables. The dashed curve connects the displayed non-dominated points, gray points are dominated, and stars denote ECHOROUTER configurations. The log-scaled cost axis makes both inexpensive routers and expensive multi-Agent systems visible without changing dominance.

In the eight-model pool table, AFlow and ECHOROUTER form the empirical frontier. AFlow is the lowest-cost point, while ECHOROUTER achieves a higher score at greater cost. In the large-pool table, the frontier contains several distinct operating points, including both Low and Medium ECHOROUTER. The latter has the highest displayed average score while remaining below several dominated multi-Agent systems in inference cost. These are finite-table Pareto statements, not universal dominance claims outside the reported model pools and protocols.

D EVOLUTION OF MODEL–ROLE–TOOL–TASK COMPATIBILITY AND PROMPTS

When a trajectory fails, ECHOROUTER distinguishes between limitations of the collaboration structure and mismatches among the model, role, tool, task, and prompt. This distinction determines whether subsequent evolution should revise the topology and system policy or adjust a specific execution component. The topology examples in Figure 7, the prompt revisions in Figure 8, and the refinements of EvoMAS policies in Table 7 illustrate these complementary forms of adaptation.

We use controlled comparisons to support this attribution. A model comparison holds the task, role, visible upstream state, and final prompt fixed. A context comparison holds the model fixed and changes only one declared contextual factor. When both the model and context change, we treat the evidence as a joint policy mutation rather than attributing the outcome to either factor alone.

D.1 EFFECTS OF MODEL-NAME PRIORS

The optimization traces provide two representative examples of how public model names can bias decisions when empirical evidence remains sparse. On AIME, the public-name condition selected GPT-4o based on the prior expectation that it would be stronger, whereas the matched empirical comparison favored GPT-OSS-120B. On GPQA-D, the public-name condition attributed an unsuccessful trajectory to the workflow architecture, while the anonymous condition and the observed comparison identified model capability as the primary limitation. Together, these cases suggest that model-name priors can distort both candidate selection and failure attribution during early optimization. They provide qualitative evidence for the mechanism examined in the ablation rather than an additional aggregate performance estimate.

The experiment records also contain training-only comparisons conducted by the Model Intern and Prompt Intern. The system-policy excerpts above document accepted policy revisions, but they are not interpreted as isolated Prompt Intern effects. We additionally retain the negative cases in Appendix C.3 to clarify that additional calls and prompt revisions are beneficial only when supported by the observed execution state.

E DETAILED ALGORITHMS

Algorithms 1 and 2 describe the two time scales of ECHOROUTER. The outer loop allocates real evaluations among proposed policies and updates Echo experience between rounds. The inner loop executes a given policy by constructing stage-specific subgraphs from intermediate evidence. The same inner procedure is used during optimization and testing. During testing, the policy, profiles, and alias map are frozen.

Outer-loop operations. To support controlled attribution, SHADOWCOMPARE holds the task, role, and upstream node input fixed. In a model comparison, only the anonymous model changes, while the prompt and visible information remain unchanged. The challenger is the highest-UCB cost-matched model under Equation 5. In a prompt comparison, the model remains fixed, and the candidate prompt is selected from $\mathcal{H}_r$ when its failure hypothesis applies to the activated call. The schedule s_r determines the activation of eligible comparisons. Shadow outputs are not inserted into the task trajectory, so local comparisons provide only local evidence. Their measured outcomes enter $\mathcal{E}_{r+1}$ as profile or sparse-adapter evidence for future proposals. Any resulting policy update must still be evaluated through candidate rollout before admission. Comparisons therefore inform later policy evolution without directly modifying an admitted policy.

ADMIT relies exclusively on measured outcomes. Candidates must satisfy the mean Agent-cost constraint, after which quality is maximized. Ties are broken by lower cost and then fewer calls. The incumbent is retained when no eligible update is admitted. Retention alone does not certify budget feasibility. Forecasts are used only to allocate evaluation slots and do not replace measured admission scores. The reachability approximation and U_z are heuristic signals used only for evaluation allocation. They are not interpreted as calibrated probabilistic estimates and provide no statistical coverage guarantees. All optimization runs, including initialization and shadow calls, are accounted for separately from the final frozen-policy execution cost.

Execution semantics. Dependency-ready nodes may execute concurrently when their inputs are independent. The updated state includes invalid outputs, failed checks, disagreements, and tool results, allowing the next stage to request verification, revision, or adjudication. Stopping and exceptional termination follow the declared policy and execution safeguards. A failed execution retains its status rather than being treated as a successful publication. The budget B is a mean-cost target rather than a per-task hard cap. Individual tasks may exceed B, while the declared maximum numbers of calls and rounds keep execution bounded. Neither reference answers nor evaluation scores are available to the inner Controller. Profiles and policies remain fixed throughout each call to RUNPOLICY. Only task-local state and accumulated cost change. Under the Agent-cost definition used here, the local Router incurs no external model API charge.

F FORECAST-ALLOCATION PROTOCOL

F.1 FROZEN CANDIDATE-POOL SELECTOR STUDY

The controlled selector study uses eight frozen candidate pools covering mathematics, coding, and general reasoning. The pools contain 58 validation questions and 16 candidates per pool. Each forecaster receives candidate descriptions and optimization history, then selects two candidates without access to their validation outcomes. Best-of-K reports the higher validation mean of the two selected candidates. The random baseline exactly enumerates all 120 candidate pairs. Selector cost covers one selection pass and excludes candidate execution.

One seed produces a global pool of candidate policy blueprints. Forecast Top-K is frozen before rollout. Only the selected candidates receive a real evaluation for selection and policy admission. The eight-pool study above additionally measured all 16 candidates on the same validation items for a counterfactual random-pair comparison. Unselected audit controls did not enter parent selection, Profile updates, or evolution.

Algorithm 1 Outer loop: forecast-guided policy evolution

Require: Initial policy Θ_1; training data $\mathcal{D}_{\mathrm{tr}}$ and fixed validation data $\mathcal{D}_{\mathrm{val}}$; model pool $\mathcal{M}$; mean Agent-cost budget B; rounds R; proposal count M; rollout slots K; exploration slots $K_{\mathrm{exp}} \le K$.
Ensure: Frozen policy Θ_{R+1}, profiles $\mathcal{P}_{R+1}$, and alias map α.

Create a seeded anonymous alias map α for $\mathcal{M}$.
Initialize $\mathcal{E}_1$, including $\mathcal{P}_1$, using controlled single-model runs on training data.
for $r = 1, \dots, R$ **do**
    Freeze the round-start snapshot $\mathcal{E}_r$ and its profiles $\mathcal{P}_r$.
    Sample the rotating training minibatch $\mathcal{D}_r^{\mathrm{tr}} \subset \mathcal{D}_{\mathrm{tr}}$.
    $\mathcal{T}_r \leftarrow \{\textsc{RunPolicy}(x, \Theta_r, \mathcal{P}_r, \alpha, B) : x \in \mathcal{D}_r^{\mathrm{tr}}\}$.
    $\mathcal{H}_r \leftarrow \textsc{Diagnose}(\mathcal{T}_r)$ ▷ Failures, useful behaviors, candidate prompts
    Set $s_r \leftarrow 1 - 0.8(r-1)/(R-1)$ if $R > 1$, and $s_r \leftarrow 1$ otherwise.
    $\mathcal{W}_r \leftarrow \textsc{ShadowCompare}(\mathcal{T}_r, \mathcal{H}_r, s_r, \mathcal{P}_r)$.
    Buffer $\mathcal{W}_r$ for the next round; do not alter current answers or profiles.
    $\mathcal{Z}_r \leftarrow \mathrm{Propose}(\Theta_r; \mathcal{H}_r, \mathcal{E}_r)$, with $|\mathcal{Z}_r| = M$.
    for each candidate $z \in \mathcal{Z}_r$ **do**
        Forecast $(\widehat{\mu}_z^Q, \widehat{\sigma}_z^Q, \{\widehat{\gamma}_{za}\})$ from $(z, \mathcal{H}_r, \mathcal{E}_r)$.
        $\widehat{\sigma}_z^Q \leftarrow \min(0.5, \max(0.005, \widehat{\sigma}_z^Q))$.
        $U_z \leftarrow \min(1, \widehat{\mu}_z^Q + \beta \widehat{\sigma}_z^Q)$, where $\beta \approx 1.28$.
        for each reachable call $a \in \mathcal{A}_z$ in dependency order **do**
            $\widehat{r}_{za} \leftarrow 1$ if $\mathrm{Pa}(a) = \varnothing$; otherwise $\widehat{r}_{za} \leftarrow \min_{p \in \mathrm{Pa}(a)} \widehat{\rho}_{zp}$.
            $\widehat{\rho}_{za} \leftarrow \widehat{r}_{za} \widehat{\gamma}_{za}$.
        end for
        $\widehat{C}_z \leftarrow \sum_{a \in \mathcal{A}_z} \widehat{\rho}_{za} \sum_{m \in \mathcal{M}} \widehat{\pi}_{zam} \bar{c}_{am}$.
    end for
    $\mathcal{F}_r \leftarrow \{z \in \mathcal{Z}_r : \widehat{C}_z \le B\}$.
    Select up to $K - K_{\mathrm{exp}}$ policies from $\mathcal{F}_r$ by predicted mean quality.
    Fill up to K_{exp} exploration slots from the remaining pool using U_z,
        topology novelty, information value, and expected improvement; let $\mathcal{S}_r$ be the selected set.
    Evaluate $z \in \mathcal{S}_r$ on $\mathcal{D}_r^{\mathrm{tr}}$ via $\textsc{RunPolicy}(x, z, \mathcal{P}_r, \alpha, B)$.
    Evaluate admission candidates on the same fixed $\mathcal{D}_{\mathrm{val}}$.
    Obtain comparable measured validation quality, mean Agent cost, and call counts.
    $\Theta_{r+1} \leftarrow \textsc{Admit}(\Theta_r, \mathcal{S}_r, \text{measured outcomes}, B)$.
    $\mathcal{E}_{r+1} \leftarrow \textsc{UpdateEcho}(\mathcal{E}_r, \mathcal{T}_r, \mathcal{W}_r, \text{evaluation and admission records})$.
    Update profiles and adaptation evidence; record forecast errors for evaluated candidates.
end for
return $\mathrm{Freeze}(\Theta_{R+1}, \mathcal{P}_{R+1}, \alpha)$.

Algorithm 2 Inner loop: evidence-conditioned adaptive execution (RUNPOLICY)

Require: Task x; policy $\Theta = (P_{\text{meta}}, P_{\text{model}}, P_{\text{context}})$; fixed profiles $\mathcal{P}$; alias map α; mean Agent-cost target B.
Ensure: Published output u, execution trace $\mathcal{T}$, and measured Agent cost C_{A}.

Initialize task state ξ_0 with x; set $t \leftarrow 0$, $C_{\text{A}} \leftarrow 0$, and $\mathcal{T} \leftarrow \varnothing$.
loop
  $b_t \leftarrow \text{SOFTBUDGET}(B, C_{\text{A}})$.
  $o_t \leftarrow (x, \xi_{0:t}, \mathcal{P}, b_t)$.
  $a_t \leftarrow \pi_\Theta(o_t)$ ▷ Construct a subgraph or publish an available output
  if $a_t = \text{Publish}(u_t)$ **then**
    Record the selected node output and stopping decision in $\mathcal{T}$.
    return $(u_t, \mathcal{T}, C_{\text{A}})$.
  end if
  Let G_t be the stage-specific subgraph specified by $a_t = \text{Construct}(G_t)$.
  Use P_{meta} to specify roles, dependencies, activation conditions, and output requirements.
  Initialize node statuses in G_t and the stage record $\Delta\xi_t$.
  while G_t has an executable node **do**
    Select dependency-ready nodes whose activation conditions hold.
    for each selected node a **do**
      Choose anonymous model $\widetilde{m}_a$ using P_{model}, $\mathcal{P}$, the role, and current evidence.
      Construct its objective, prompt, tools, and visible inputs using P_{context}.
      Resolve $\widetilde{m}_a$ through α inside the execution controller.
      Execute node a and collect its output, tool results, status, and usage record.
      $C_{\text{A}} \leftarrow C_{\text{A}} + \Delta C_{\text{A},a}$ ▷ Standardized external Agent API cost
      Append the node record to $\Delta\xi_t$; update dependency and branch statuses.
    end for
  end while
  Mark inactive or blocked nodes and record their reasons in $\Delta\xi_t$.
  $\xi_{t+1} \leftarrow \text{UPDATESTATE}(\xi_{0:t}, G_t, \Delta\xi_t)$.
  Append the graph, routing decisions, node records, and stage cost to $\mathcal{T}$.
  $t \leftarrow t + 1$ ▷ Reconsider stopping, further solving, or targeted repair
end loop

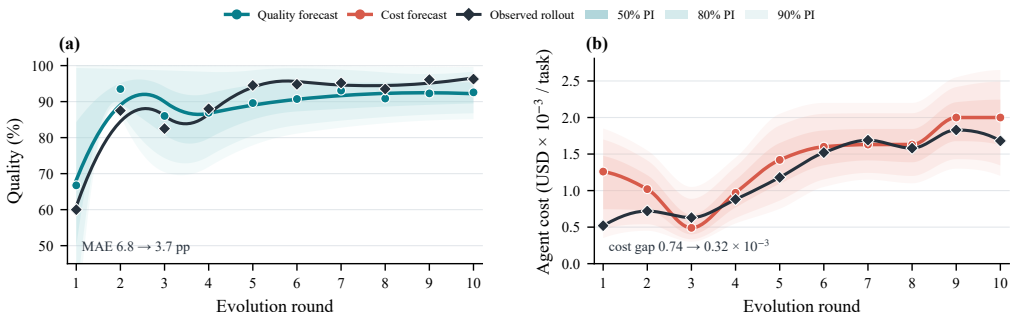

Figure 11: Forecast and observed rollout trajectories over ten optimization rounds in the general-medium audit. Each point averages the evaluated candidates in that round. Quality and Agent cost use optimization-side training and validation results.

Table 8: Qwen3.5-9B controller-training configurations. Source-record counts precede token-length filtering. Validation loss measures SFT fit.

Run	Training data, optimizer settings, and completed artifact
Full SFT	2,224 source records; 5% training-data validation split; 8,192-token cutoff; Qwen3 no-think template. Three epochs on 8 devices; per-device batch 1, gradient accumulation 8 (effective batch 64); AdamW, learning rate 10^{-5}, cosine schedule with 10% warmup, BF16, seed 42, and ZeRO-3. The saved final state completed 99 steps; training loss 0.2465 and validation loss 0.3343.
Standard LoRA	2224 source records; 8,192-token cutoff and required message format. Nine epochs on 2 devices; per-device batch 1, gradient accumulation 1; fused AdamW, learning rate 10^{-4}, BF16, and gradient checkpointing. LoRA rank 8, scale 16, dropout 0.05, targeting query/key projections. The saved final state completed 99 steps; no validation split or end-task score is attached to this run.

For the reported $K = 2$ study, random comparison uses pairs from the exact measured pool. Candidate-rollout saving is

$$1 - \frac{\sum_{j \in \mathcal{Z}_K} C_j}{\sum_{j=1}^{M} C_j}. \tag{11}$$

Forecast-generation cost is reported separately. Full-pool audit cost is excluded from production savings because it is purchased only for counterfactual evaluation.

Candidate generation, selected rollout, and scientific-audit execution have separate ledgers. Full-pool audit cost is excluded from production savings.

G Compact-Controller Training Configurations

The controller-data pipeline uses GPT Luna to extract decision traces from optimization-side runs spanning different budget levels. Training data come from GSM8K, MMLU, OlympiadBench, MBPP, the generation validation set, and FinanceQA. We retain a trace for fine-tuning only when its action sequence has the required executable structured format and the resulting task artifact passes its task-specific correctness check. This filters malformed calls and incorrect outcomes before constructing Controller input–action examples. Final-test tasks are never used for this selection. We report Qwen3.5-9B full-SFT and standard-LoRA training configurations. Table 8 reports launch settings and resulting training losses. The runs used different data files and training recipes, so their losses are not directly comparable. The standard-LoRA run used a separate real-name diagnostic set. The main results use anonymous policies.

H Controller Serving and Runtime Accounting

Local serving. The Qwen3.5-9B Controller was served on a single NVIDIA RTX A6000 GPU with vLLM and an EAGLE-3 draft head for speculative decoding during optimization and frozen-policy evaluation. At each decision, it receives the current execution state and policy instructions

Table 9: Mean Controller calls and cumulative local request time per frozen-policy test query. Time is calculated from the mean call count and the measured mean request duration for each task family. AIME combines AIME 2024 and 2025. The final row is the unweighted mean over the eight benchmarks.

	Eight-model pool		Larger pool L		Larger pool M	
Benchmark	Calls	Time (s)	Calls	Time (s)	Calls	Time (s)
AIME	3.2	1.44	2.9	1.31	3.5	1.58
MATH	2.6	1.17	2.5	1.13	2.9	1.31
GSM8K	2.5	1.13	2.1	0.95	2.2	0.99
LCB	2.9	1.22	3.0	1.26	3.1	1.30
HumanEval	3.1	1.30	2.6	1.09	2.5	1.05
DROP	2.1	0.99	2.2	1.03	2.2	1.03
GPQA-D	2.4	1.13	2.5	1.18	3.1	1.46
ARC-C	2.0	0.94	2.0	0.94	2.1	0.99
Avg.	2.60	1.16	2.48	1.11	2.70	1.21

Table 10: Mean Controller calls and cumulative local request time per executed training task across the ten optimization steps.

	Mathematics		Coding		General reasoning	
Setting	Calls	Time (s)	Calls	Time (s)	Calls	Time (s)
Eight-model pool	2.74	1.23	2.56	1.08	2.21	1.04
Larger pool L	2.44	1.10	2.68	1.13	2.23	1.05
Larger pool M	2.77	1.25	2.68	1.13	2.33	1.10

and emits a structured action for the next stage. Observed generation throughput ranged from 64 to 297 tokens/s. Input prefill took 0.06–0.10 s per request and is included in the measured request durations.

Test-time Controller use. Table 9 reports the mean number of local Controller calls per frozen-policy test query. Controller request time is calculated as the mean call count multiplied by the mean request duration for the corresponding task family: 0.45 s for mathematics, 0.42 s for coding, and 0.47 s for general reasoning. These times account for local Controller requests and are separate from the Agent API costs in the main tables.

Controller use during optimization. Table 10 summarizes local Controller calls per executed training-task rollout across the ten optimization steps. Request time uses the same task-family durations as Table 9. Thus, this table reports Controller time per training-task execution, while the one-time optimization costs in the main tables retain their stated USD accounting.

The Controller runs on local hardware and therefore adds no external Controller API charge. Its request times are reported as a separate local resource measure. They are not included in the test-time Agent API cost column.

I Reproducibility Details

The experimental records specify split definitions, candidate models, tool permissions, decoding settings, price ledgers, the optimization protocol, trajectory costs, Intern activation, and the blinded generation protocol. Tables 11–12 summarize these settings.

J Execution Prompt Construction and Policy Artifact Example

An execution call keeps the evolving Agent instruction in the system message: the runtime places `agent_policy` beneath an `Agent execution policy` heading and appends the required

Table 11: Experimental configuration.

Field	Final value
Data splits	Each benchmark uses disjoint optimization and held-out test splits. Training and validation data guide candidate generation and selection; final test data are used only after the policy is frozen.
Execution-model pool and exact checkpoints	The complete API identifiers are listed in Table 6; each rollout records the bound model string, token counts, context, and policy hash.
Anonymous alias protocol	Aliases are stable within one run and task-local Profile, hidden from the optimizer, frozen before held-out execution, and disclosed only after freeze. The reported independent run seeds are 42, 43, and 44; no claim relies on cross-seed alias permutation.
Budget regimes and contracts	Extra Low/Low/Medium/High/Extra High correspond to mean Agent-cost anchors \$0.003/\$0.006/\$0.012/\$0.024/\$0.048. Each candidate separately declares maximum calls and rounds; early stopping is permitted, and Router cost is reported outside the Agent cap.
Decoding and tool settings	Temperature, maximum tokens, and retry limits are frozen per candidate rather than globally shared. Tool permissions are given in Appendix B.1; undeclared tools and model calls are forbidden.
Price accounting	The execution-time standard model-price table produces the optimization objective; per-call ledgers separately retain estimated standard cost, observed billing when available, Router cost, candidate-generation cost, failed attempts, and recovery. A single consolidated snapshot date was not retained, so no claim depends on a later price comparison.
Seeds and uncertainty	The reported runs use seeds 42, 43, and 44. The tables report observed frozen-policy point estimates.

Table 12: Optimization and Intern parameters.

Component	Final parameterization
Forecast-guided policy evolution	10 optimization steps; rotating training minibatches; one fixed validation set; minibatch screening followed by full-validation admission; budget-feasible quality first, then cost on quality ties; full lineage and frontier records retained.
Forecast allocation	M is a proposal-pool parameter. The reported selector study uses 16 candidates per pool and Top-2; all reported experiments use $K = 2$. Selection is frozen before rollout, and unselected audit controls cannot enter evolution.
Calibration	Chronological quality, cost, branch, and call residuals are stored in `forecast_calibrations.json`; only earlier residuals may affect a later forecast. The reported selector evidence concerns Top-2 utility and savings, not accurate total ranking.
Model / Prompt Intern	Training-only Model Intern comparisons hold task, role, prompt, and visible upstream state fixed. Prompt Intern comparisons hold task, role, model, and upstream state fixed while changing the candidate prompt. Evidence enters the Profile only after a measured comparison; Intern cost is excluded from deployment cost.
Policy contracts	Only configured anonymous aliases are legal.
Blinded generation study	Websites, visual applications, slides, games, SVGs, and documents; aggregation over 14 task categories shared by all methods; GPT-5.6 Luna, Gemini-2.5 Flash and DeepSeek V4 Flash judge requirement following, correctness/completeness, technical validity, visual hierarchy, UX/accessibility, and finish quality. Current tables report descriptive means.

output contract. The user message contains the original task, only directly selected dependencies, the assigned role, and the local objective. Figure 8 shows dataset-neutral excerpts of saved candidate system messages. The following archived candidate illustrates how role instructions and Controller-visible conditions support different collaboration paths. Its task input and output contract are omitted.

Conditional code collaboration policy

Name: `s01_tool_first_test_and_repair` **Candidate:** `policy: 75977e0cd42f04f6b711`

Used in: `three_frozen_tool_rounds_isolated_v2_20260827/coding`.
Inputs visible to the Agent: assigned role; original task; declared output format; direct dependency artifacts; the `run_code` interface for the Critic; and, for reconstruction, the original artifact plus the Critic's defect report.
Information hidden from the Agent: held-out answers and evaluator outcomes; public model identity behind anonymous slots; unrelated Agent outputs; optimization selection records; and Controller-only routing, model-selection, budget, and stopping decisions.

Agent-policy excerpt.
Follow the assigned role exactly and produce only the required artifact.

- **independent_solver:** Generate a complete solution in the declared output format (for example, boxed code) with no extra commentary.
- **critic:** Receive the candidate artifact and original task. Execute the artifact using `run_code`. Return `VALID` if all checks pass, or `DEFECT: <brief description>` when execution, output, interface, or edge-case checking fails.
- **reconstructor:** Receive the task, original artifact, and Critic defect report. Produce a corrected solution in the same format as the independent Solver. Do not make model-selection, routing, or stopping decisions; those belong to the Controller.

Conditional paths in the candidate policy (condensed).

- **Routine completion:** One Solver sees the task and produces a solution. With no risk cue or failed check, the Controller can stop without launching a Critic.
- **Semantic risk:** Two independent Solvers receive the task. If their solutions disagree, an artifact is invalid, or ambiguity remains, the Controller gives both solutions to a Critic for comparison.
- **Execution risk:** A Solver produces code and a Critic checks it with `run_code`. A `VALID` report permits stopping; a concrete failure passes the code and diagnostic report to a Reconstructor for one targeted repair.
- **Medium coding task:** A failed check, complex-algorithm cue, or disagreement can activate an alternate Solver and a Critic after the initial solution, supplying another candidate answer for review.

Boundary: The Controller selects a path from observed task and feedback signals, with at most three Agent calls and one repair wave. These are recorded policy branches, not a claim that every branch ran in the cited optimization rollout.

J.1 Representative Optimization and Runtime Prompts

The following are short, source-backed excerpts of four distinct model instructions. Omitted schema fields and task-specific data are supplied by the runtime. They are not execution-Agent prompts: the first three operate only on optimization data, while the Controller instruction is used to choose runtime actions. Figure 8 instead shows execution-Agent prompt candidates.

Forecast: estimate candidates before execution

System excerpt: "You are a concise listwise topology simulator. Anonymous model IDs encode no capability, price, or ordering. For every candidate estimate expected task quality mean, epistemic standard deviation, and each conditional call's trigger probability given that its dependencies were reached. ... Do not rank, select, rewrite, or repair candidate topology. Do not estimate monetary cost; the program computes it from branch probabilities and Profile whole-call costs."

Reflection: diagnose the responsible component

System excerpt: "You are a component-level analyst for a dynamic multi-agent router. Diagnose failures without assuming that the selected model is globally weak. ... Separate: Meta Policy/topology, PolicyGuide/model assignment, model capability for this slot, model-specific Adapter, role/tool assignment, context, environment/tool, and evaluator error. Return JSON only."

Proposal: turn diagnoses into testable policy edits

Abridged, terminology-normalized system instruction: "You are an inventive but evidence-disciplined policy architect for a dynamic multi-agent router. Translate causal hypotheses into compact, testable context-conditioned policy mechanisms. Never solve benchmark tasks or memorize a sample's answer. Produce a concrete reusable recipe for the task family under optimization." The response schema asks for a diagnosis, mutation strategy, selected blueprint, tested hypothesis, expected behavioral change, and component updates; prompt-only edits keep the parent topology and model bindings fixed.

Controller: act on the observed state

System excerpt: "You are the runtime stage controller and dynamic topology constructor. Do not solve the task and do not write a final answer. Follow a Guide-conditioned loop on every call: OBSERVE the task analysis, current state, sources, artifact and action history; REFLECT on what the Guide requires in this state and what evidence is still missing; then ACT by finishing from an existing source or constructing the smallest useful next topology. ... The next Controller call re-observes reality and chooses the matching situation again. Return JSON format."

These excerpts correspond to the forecast simulator in `metarouter_v3/evolution_forecast.py`, the policy proposer in `metarouter_v3/evolution_design.py`, and the reflection and runtime Controller templates in `router_prompts_optimized.py`. Ellipses mark omitted clauses. The term "role/tool" above follows the paper's terminology for the corresponding source field. The full versioned templates and candidate policy JSON remain the executable record.

K Integrating New Models into Frozen Policies

We study the ability to incorporate newly available models into existing EchoRouter policies across six benchmark tasks. When a newly arrived model demonstrates favorable performance on a given task, the router may select it for use while keeping the original incumbent model available for other tasks.

Table 13: New-model access cost, optimization cost under optimization-from-zero baseline, and final task-specific bindings, in USD. D, L, and Q denote DeepSeek V4.1 Flash, Llama 3.1 70B, and Qwen3.5 27B. All runs adopt the same question counts and search spaces. Opt. From Zero means we optimize from seed policy; otherwise we fix the policy and evaluate new models via model comparison.

Task	D	L	Q	Access	Opt. from Zero	Ratio	Selected
AIME	0.215	0.085	0.180	0.480	0.972	49.4%	Q
DeepSearchQA	0.515	0.185	0.400	1.100	8.943	12.3%	D, Q
GPQA-D	0.155	0.075	0.110	0.340	1.913	17.8%	D, Q
HLE	0.325	0.130	0.280	0.735	7.356	10.0%	D
LiveCodeBench	0.725	0.260	0.550	1.535	2.959	51.9%	D
Generation	1.000	0.435	0.825	2.260	14.570	15.5%	D, Q
Total	2.935	1.170	2.345	6.450	36.713	17.57%	—

Qwen3.5 27B in four tasks and DeepSeek V4.1 Flash in five. Llama 3.1 70B was not selected for any task, and LiveCodeBench maintains its original incumbent model. The overall access cost across all six tasks totals $6.45, covering evaluation of all screened candidate models. The optimization cost under the optimization-from-zero baseline is listed for each task. The results demonstrate that our method can rapidly onboard new models via model comparison at low access cost. It achieves substantial cost savings compared with optimization from zero, especially for complex long-horizon tasks.

L Additional Open-Ended Generation Examples

The examples below complement the aggregate evaluation in Figure 4. Figure 12 compares archived outputs at increasing spend within three artifact types. Figure 13 follows successive revisions of one slide deck. Figure 14 shows six output formats. These qualitative examples are not additional preference scores or evidence that cost alone causes improvement.

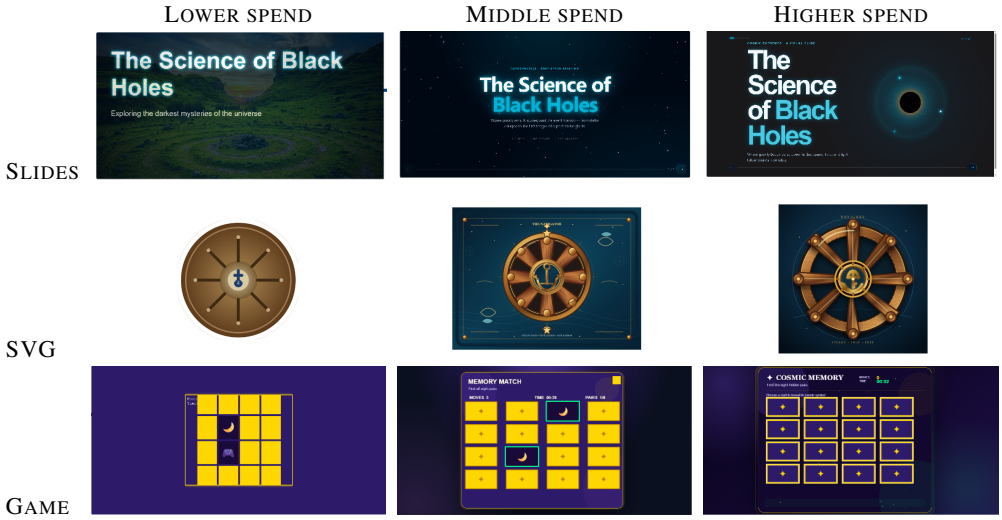

Figure 12: Representative generation outputs across increasing per-artifact spend. Each row keeps the task type fixed: a black-hole slide deck, a vector steering-wheel illustration, or an interactive memory game. For slides and games, each cell isolates one view from the saved three-view artifact record. The SVG cells show the complete image. Higher-spend examples show changes in visual hierarchy and interface treatment, while Figure 4 supplies the aggregate evaluation.

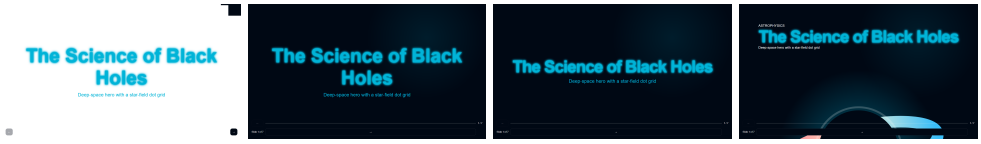

Figure 13: Selected stages from a seven-version black-hole slide-deck lineage, shown as one cover-slide view per displayed version. V2–V4 are omitted for legibility. The archived trace records one artifact ID and a parent-hash chain from V1 through V7. These are successive revisions, not independent samples or measured per-version quality scores.

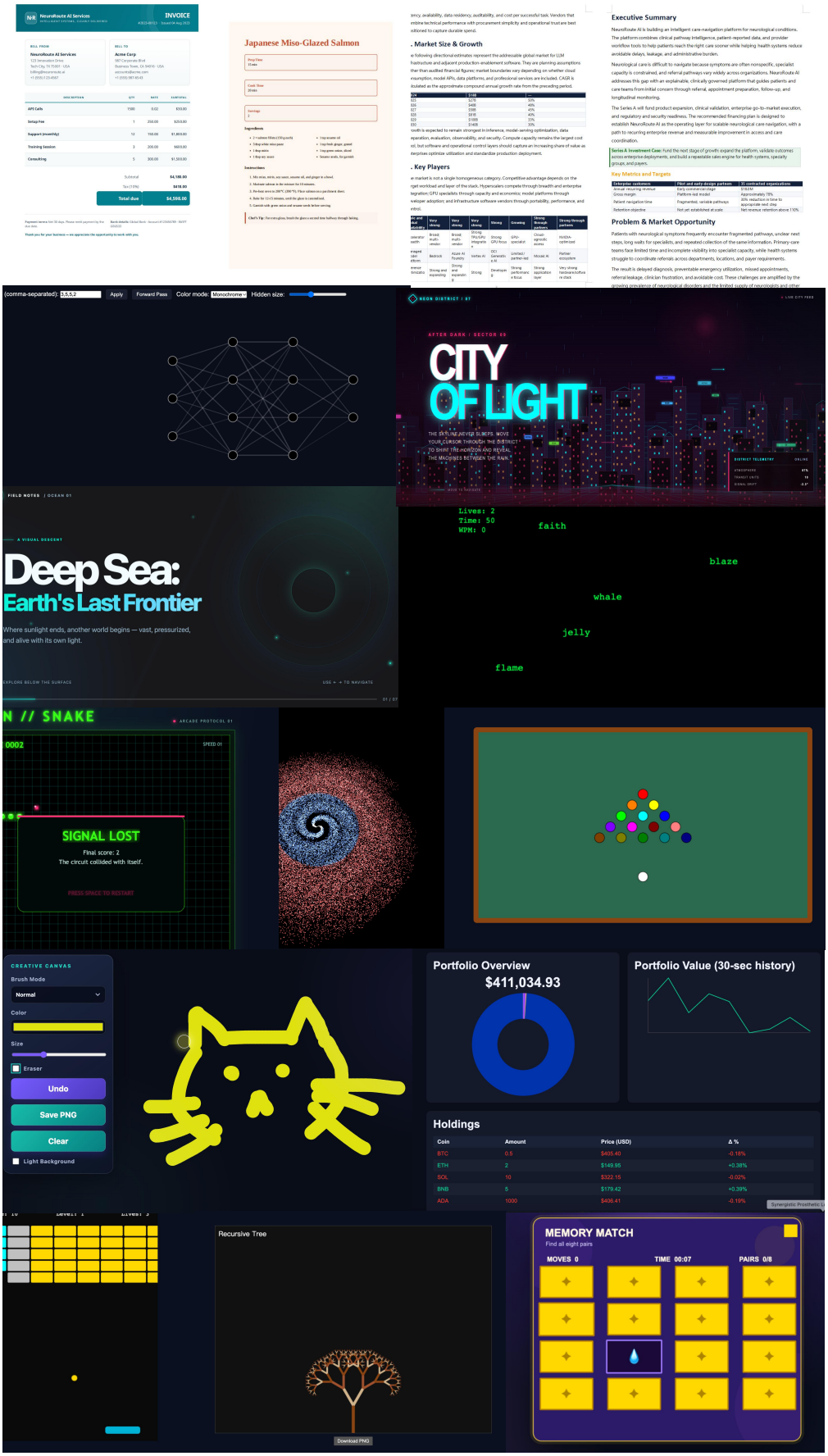

Figure 14: Format diversity among archived ECHOROUTER outputs.