EVOORCH: EVOLVING ORCHESTRATION FOR DYNAMIC MULTI-AGENT SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model-based multi-agent systems (MASs) solve complex, long-horizon tasks by decomposing them into subtasks and assigning these subtasks to specialized sub-agents. However, existing systems often rely on predefined or ad hoc orchestration structures, limiting their adaptability at both the intra-task and cross-task levels: 1) within a task, they struggle to dynamically reorganize sub-agent structures and correct execution failures in a timely manner, and 2) across tasks, they make limited use of previous trajectories to improve future orchestration. To address these limitations, we propose **EvoOrch**, a multi-agent orchestration framework that supports both intra-task and cross-task evolution. EvoOrch consists of a persistent Meta Agent and dynamically instantiated sub-agents. At the intra-task level, EvoOrch performs *structural evolution* by dynamically constructing and extending task-specific sub-agent structures, and *configuration evolution* by revising ineffective active sub-agents during execution. At the cross-task level, EvoOrch follows a two-stage pipeline that first hierarchically organizes completed multi-agent trajectories into structured trajectories, and then performs trajectory-driven analysis to distill reusable orchestration and verification insights, which are stored in an experience base and retrieved to guide future tasks without parameter updates. Experiments on diverse settings show that EvoOrch surpasses the state-of-the-art automated orchestration method by 7.9 points on average, while achieving a better Pareto trade-off between performance and cost.

1 INTRODUCTION

Large language model (LLM)-based agents have shown strong potential in solving diverse tasks (Yao et al., 2023; Wang et al., 2024; Yang et al., 2024). To handle increasingly long-horizon tasks, recent studies (Wu et al., 2023; Hong et al., 2024) extend single-agent system to multi-agent system (MAS), enabling collaboration among specialized sub-agents.

Despite this progress, constructing an effective MAS remains challenging, because manually designing agent workflows and configurations requires substantial effort and often generalizes poorly across tasks Hu et al. (2025); Lin et al. (2026a). This motivates a shift from manually crafted, static MASs toward evolving MASs Hong et al. (2024); Zhang et al. (2025); Lin et al. (2026b), which can adapt their organization during execution and improve future orchestration through accumulated cross-task experience, thereby reducing the need for manual system design.

Recent studies Lyu et al. (2026) pursue this goal through automated orchestration, which can be categorized into intra-task and cross-task levels, depending on whether orchestration evolves within the current task or across tasks. At the intra-task level, AOrchestra Ruan et al. (2026) dynamically instantiates sub-agents according to the current execution state, but primarily organizes them through sequential delegation. Modern coding agent such as Claude Code CLI Anthropic (2025), Nanobot Ren (2026) also support runtime delegation to sub-agents and lightweight multi-agent coordination during execution. Compared with predefined MASs, these approaches provide greater flexibility, but offer limited support for explicitly constructing task-specific dependency structures among sub-agents. This limitation becomes important when a task requires multiple directions to be explored in parallel and subsequently aggregated or verified. Moreover, existing runtime orchestration mechanisms provide limited support for revising an active sub-agent once its current configuration becomes ineffective. These limitations are particularly important for long-horizon tasks, where

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.

both the appropriate agent structure and effective sub-agent configurations may change as execution unfolds. At the cross-task level, existing methods leverage previous executions to improve future orchestration. Some Ruan et al. (2026); Dang et al. (2025) fine-tune policies from collected trajectories, while others evolve or retain high-performing workflows and reusable skills based on execution statistics Zhang et al. (2025); Lin et al. (2026b). However, execution statistics mainly indicate which workflows or strategies are effective, while raw multi-agent trajectories contain substantial task-specific details and redundant execution histories, making it challenging to distill reusable orchestration insights. In particular, unsuccessful trajectories may still reveal reusable failure patterns, while successful trajectories contain orchestration and verification strategies that can be abstracted beyond task-specific execution details. A key challenge for cross-task evolution is therefore to extract such process-level orchestration insights while filtering out task-specific execution details.

To address these limitations, we propose **EvoOrch**, an evolving orchestration framework for LLM-based MASs that supports both intra-task and cross-task evolution. EvoOrch consists of a persistent Meta Agent and dynamically instantiated sub-agents, and formulates multi-agent task solving as a dynamic graph transition process. At the intra-task level, EvoOrch performs two complementary forms of evolution. *Structural evolution* dynamically constructs and extends task-specific sub-agent structures according to the current execution state, while *configuration evolution* monitors active sub-agents and revises their configurations when their current execution becomes ineffective. These agentic behaviors are coordinated through a multi-agent operation protocol, which forms a closed-loop execution process. At the cross-task level, EvoOrch follows a two-stage evolution pipeline. First, *hierarchical trajectory organization* preserves the Meta Agent's orchestration trace while summarizing sub-agent execution trajectories, producing a structured trajectory that connects orchestration decisions with their execution outcomes. Then, *trajectory-driven analysis* reviews the structured trajectory to distill reusable orchestration insights, including effective coordination patterns and diagnosed failure patterns. These distilled insights are stored in an experience base and retrieved to guide future orchestration.

We evaluate EvoOrch against 11 baselines across multiple benchmarks and backbones settings. The results show that EvoOrch surpasses the state-of-the-art automated orchestration method AOrchestra by 7.9 points on average, while achieving a better Pareto trade-off between performance and execution cost than existing methods. We provide in-depth analyses of both intra-task and cross-task evolution, examining how structural adaptation, configuration intervention, and backbone sensitivity contribute to intra-task execution, as well as how historical orchestration insights enable cross-task transfer. The main contributions are summarized as follows:

- We propose **EvoOrch**, a training-free evolving orchestration framework that dynamically evolves both sub-agent structures and configurations according to the execution state.
- We further introduce a two-stage pipeline that hierarchically organizes completed trajectories and performs analysis to distill reusable orchestration insights for future tasks.
- Experiments across diverse settings show that EvoOrch outperforms strong automated orchestration baselines by an average of 7.9 points while achieving a better performance-cost trade-off among MASs.

2 METHODOLOGY

In this section, we first formalize dynamic multi-agent execution and agent configurations, and then present EvoOrch, including its overall framework, intra-task and cross-task evolution mechanisms.

2.1 PROBLEM FORMULATION

2.1.1 GRAPH-STATE MULTI-AGENT EXECUTION

To characterize how the multi-agent system changes throughout task execution, we represent the evolving MAS as a sequence of graph states. Given a task x, a MAS solves it through a structured execution process. At each time t, the MAS maintains an agent graph state:

$$G(t) = \langle \mathcal{V}(t),\ \mathcal{E}(t),\ \mathcal{S}(t) \rangle. \tag{1}$$

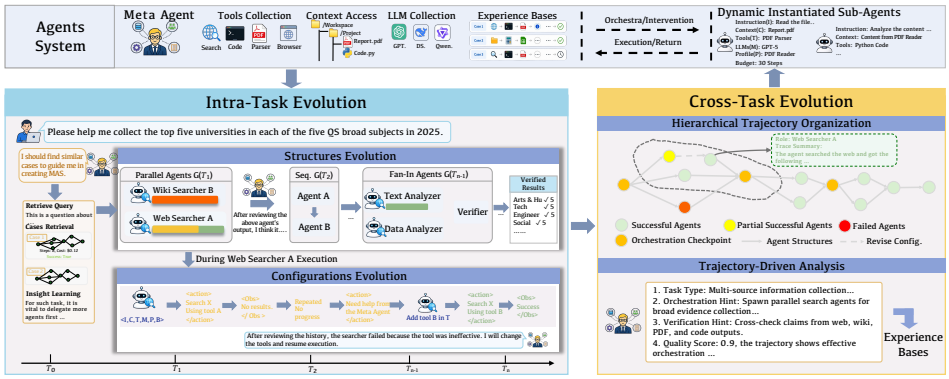


Figure 1: Demonstration of the proposed EvoOrch.

Here, $\mathcal{V}(t) = \{v_1(t), v_2(t), \ldots, v_n(t)\}$ is the set of agent nodes at time t, $\mathcal{E}(t) \subseteq \mathcal{V}(t) \times \mathcal{V}(t)$ represents the directed dependencies between agents, and $\mathcal{S}(t)$ denotes the information state. Similar to React Yao et al. (2023), the MAS execution can then be formulated as a graph-level trajectory:

$$\Gamma(x) = \big(G(0), a(0), o(0), G(1), a(1), \ldots, G(T)\big), \tag{2}$$

where $a(t)$ denotes the system-level action taken at step t. Such actions may execute tools to update $\mathcal{S}(t)$, or modify the agent organization by updating $\mathcal{V}(t)$ and $\mathcal{E}(t)$. $o(t)$ denotes the observation produced by the action. The graph state then evolves according to

$$G(t+1) = \Phi\big(G(t), a(t), o(t), x\big), \tag{3}$$

where Φ denotes the graph-state transition induced by the selected action and its observation. The execution continues until $\mathcal{S}(t)$ contains sufficient evidence to answer the task.

2.1.2 Agent Configuration

AOrchestra Ruan et al. (2026) propose a unified agent abstraction using a four-tuple $\langle \mathcal{I}, \mathcal{C}, \mathcal{T}, \mathcal{M} \rangle$, where $\mathcal{I}$, $\mathcal{C}$, $\mathcal{T}$, and $\mathcal{M}$ denote the instruction, context, tool set, and model. These characterize the key elements of an agent and can accommodate diverse agent forms, such as ReAct Yao et al. (2023), SWE-Agent Yang et al. (2024). For MASs, we extend this abstraction with a functional profile $\mathcal{P}$ and an execution budget $\mathcal{B}$. The profile specifies an agent's functional role in the multi-agent graph, while the budget specifies the execution resources allocated to the agent, such as its maximum number of execution steps. Formally, each agent node v_i at time t is represented as:

$$v_i(t) = \langle \mathcal{I}_i(t), \mathcal{C}_i(t), \mathcal{T}_i(t), \mathcal{M}_i(t), \mathcal{P}_i(t), \mathcal{B}_i(t) \rangle. \tag{4}$$

Unlike prior formulations in which an agent configuration remains fixed after creation, the time-indexed representation allows these components to be revised during execution, enabling fine-grained adaptation of an active agent without reconstructing the entire graph.

2.2 Overview

As shown in Figure 1, EvoOrch consists of a persistent Meta Agent $\mathcal{A}_M$ and dynamically instantiated sub-agents $\mathcal{V}_S(t)$. The Meta Agent maintains the global execution state and coordinates task-specific sub-agents, while each sub-agent executes its assigned task within an isolated context and local execution budget. EvoOrch evolves the multi-agent system at two temporal levels. At the *intra-task* level, *structural evolution* dynamically adapts the participating agents and their dependency structure, while *configuration evolution* revises active sub-agent configurations during execution. Both are realized through the Multi-Agent Operation Protocol (MAOP), forming a closed-loop execution process. At the *cross-task* level, EvoOrch hierarchically organizes completed multi-agent trajectories and performs trajectory-driven analysis to distill reusable orchestration and verification insights, which are stored in an experience base to guide future tasks.

2.3 INTRA-TASK EVOLUTION

EvoOrch performs intra-task evolution from two aspects: *structural evolution* creates the agents and their dependency structure, while *configuration evolution* revises active sub-agent configurations during execution. We instantiate them through MAOP, where ORCHESTRATION supports structural evolution, INTERVENTION supports configuration evolution, and EXECUTION and RETRIEVAL update the information state through task interaction and cross-task insight retrieval, respectively.

Structural Evolution Structural evolution is realized through ORCHESTRATION operations, which allow $\mathcal{A}_M$ to dynamically extend the agent graph by instantiating new sub-agents and specifying their dependency structures according to the current execution state. Specifically, $\mathcal{A}_M$ can instantiate a single sub-agent, multiple parallel sub-agents, or structured sub-agent groups with explicit internal dependencies. These operations determine which agents participate in the current execution and how information is propagated among them, thereby updating $\mathcal{V}(t)$ and $\mathcal{E}(t)$:

$$\langle \mathcal{V}(t+1), \mathcal{E}(t+1) \rangle = \Phi_{\text{orch}}\big(G(t), a_{\text{orch}}(t)\big). \tag{5}$$

Here, $a_{\text{orch}}(t)$ denotes an orchestration action such as DELEGATE-PARALLEL, or DELEGATE-GROUP defined in the MAOP. The instantiated sub-agents execute according to the resulting dependency graph and return their outputs to the Meta Agent. This allows EvoOrch to adapt both the scale and topology of the agent organization as new subproblems emerge during execution.

Configuration Evolution Configuration evolution is realized through INTERVENTION operations, which allow $\mathcal{A}_M$ to regulate sub-agents during their executions. An intervention can be triggered by either *hard* or *soft* conditions. Hard conditions are triggered by defined execution rules, such as repeated tool failures (e.g., three consecutive failures) or fixed monitoring intervals (e.g., every 10 sub-agent steps). For soft conditions, a sub-agent can explicitly request assistance from the Meta Agent when it cannot make further progress. Once triggered, the Meta Agent performs a two-stage diagnosis. First, $\mathcal{A}_M$ determines whether the sub-agent is making effective progress, faces a recoverable configuration bottleneck, or is unlikely to recover, corresponding to CONTINUE, CHANGE, and STOP. The diagnosis is based on its execution history, including actions, observations, and prior outcomes. CONTINUE preserves the current configuration when useful progress is still being made, whereas STOP terminates an unproductive sub-agent. If CHANGE is selected, $\mathcal{A}_M$ further diagnoses the primary failure source and revises one configuration dimension, including the tool set, instruction, context, backbone, or budget:

$$v_i(t+1) = \Phi_{\text{int}}\big(v_i(t), G(t), a_{\text{int}}(t)\big). \tag{6}$$

This enables EvoOrch to revise an active sub-agent in place, rather than reconstructing the entire agent graph. Detailed diagnosis criteria and prompt templates are provided in Appendix A.6.3.

Execution and Retrieval. EXECUTION provides the interface for interacting with the task environment. The Meta Agent directly invokes atomic tools, while sub-agents use their assigned tool sets $\mathcal{T}_i$ to execute delegated subtasks. The observations are incorporated into the execution history and global information state $\mathcal{S}(t)$, providing new evidence for subsequent system execution. RETRIEVAL connects cross-task insights to the current execution by retrieving distilled orchestration insights from similar historical tasks. The retrieved insights serve as high-level guidance for future orchestration, allowing $\mathcal{A}_M$ to adapt or ignore them according to the current state.

2.3.1 EVOORCH EXECUTION PROCESS

Within a task, EvoOrch forms a closed-loop execution process through MAOP operations. Given a task x, EvoOrch initializes a minimal graph $G(0) = \langle \{\mathcal{A}_M\}, \emptyset, \mathcal{S}(0) \rangle$ and allows $\mathcal{A}_M$ to make at most B_M checkpoint decisions. At each checkpoint, $\mathcal{A}_M$ observes the current state and selects RETRIEVAL, EXECUTION, or ORCHESTRATION, or terminates with a final answer. RETRIEVAL retrieves relevant historical insights and incorporates them into the current context; EXECUTION directly invokes an atomic tool; and ORCHESTRATION performs structural evolution by creating task-specific sub-agents $\mathcal{V}_S(t)$ and constructing their dependency graph $\mathcal{E}(t)$. Then sub-agents execute according to the dependency graph, each with an isolated environment and a step budget B_i. Each sub-agent is activated only after all predecessors are available, while independent sub-agents can be scheduled concurrently.

During sub-agent execution, INTERVENTION may be triggered following the defined hard or soft conditions. The Meta Agent $\mathcal{A}_M$ then selects CONTINUE, CHANGE, or STOP, where $\mathcal{A}_M$ performs configuration evolution by revising the diagnosed dimension of the active sub-agent. After a sub-agent completes, it summarizes its local trajectory and incorporates the summary into the global state $\mathcal{S}(t)$. The $\mathcal{A}_M$ then determines whether the state is sufficient to answer the task. If not, execution proceeds to the next checkpoint; otherwise, $\mathcal{A}_M$ generates the final answer and stores the complete multi-agent trajectory for subsequent cross-task evolution. The complete execution procedure and prompts are provided in Algorithm 1 and Appendix A.6.3.

2.4 CROSS-TASK EVOLUTION

Cross-task evolution follows a two-stage pipeline consisting of Hierarchical Trajectory Organization and Trajectory-driven Analysis. The former converts completed multi-agent executions into compact structured trajectories, while the latter reviews these trajectories and distills reusable orchestration and verification insights for future tasks.

Hierarchical Trajectory Organization. After each task execution, EvoOrch stores the completed multi-agent trajectory, which contains the task, the Meta Agent's orchestration trace, the execution trajectories of instantiated sub-agents, and the final result. Directly retaining all raw sub-agent trajectories would introduce substantial task-specific details and redundant execution histories. We therefore organize the trajectory hierarchically by preserving the Meta Agent trajectory as the global orchestration trace while summarizing each sub-agent trajectory into a compact execution summary. Each summary captures the assigned subtask, key execution behavior, obtained evidence or output, and execution status, and is inserted into the corresponding orchestration step of the Meta Agent trajectory. The resulting structured trajectory preserves the relationship between orchestration decisions and their execution outcomes while removing unnecessary low-level execution details.

Trajectory-driven Analysis. Given the structured trajectory, EvoOrch performs trajectory-driven review to distill a structured insight representation centered on four primary components: (1) the *task type*, which characterizes the general task pattern, such as complex reasoning or broad information collection; (2) the *orchestration hint*, which captures reusable agent structures and coordination patterns from successful trajectories, as well as diagnosed failure causes and patterns to avoid from unsuccessful ones; (3) the *verification hint*, which specifies how intermediate evidence or final results should be cross-checked; and (4) a *quality score* that reflects the reliability of the distilled insights. We consider two review settings. *Golden-Review* leverages both outcome supervision and process signals to identify effective or ineffective orchestration decisions, whereas *Self-Review* relies only on the execution process and does not require gold supervision. In both settings, the review abstracts reusable orchestration and verification strategies. Finally, the task representation and the distilled insights are stored in an experience base and retrieved for similar future tasks, providing strategy-level guidance for subsequent orchestration without parameter updates.

3 EXPERIMENT

3.1 SETUPS

Benchmarks. We mainly evaluate EvoOrch on three agentic benchmarks: GAIA Mialon et al. (2023), BrowseComp-Plus Chen et al. (2025b), and Travel Planning Zhang et al. (2026b). To further assess generality, we evaluate BrowseComp-Zh Zhou et al. (2025), WideSearch Wong et al. (2025), and Terminal-Bench 2.0 Merrill et al. (2026). Each case is executed three times. We report Avg@3, Pass@3, and cost for the applicable benchmarks, while Travel Planning follows its official commonsense score (C.S.) and personalized score (P.S.) metrics.

Baselines. We compare EvoOrch against three categories of baselines. The first category consists of single-agent methods, including ReAct Yao et al. (2023), ReAct+Skills, and an evolving method HGM Wang et al. (2025). The second category comprises MAS workflows, including OWL Hu et al. (2025), smolagents Roucher et al. (2025), and SMAS Lin et al. (2026a). The third category includes agent-orchestration methods, including GPTSwarm Zhuge et al. (2024), MASFactory Liu et al. (2026), EvoMAS Hu et al. (2026), AOrchestra Ruan et al. (2026), and Nanobot Ren (2026).

Table 1: Performance comparison across benchmarks under different LLM backbones. Best and second-best results within each backbone are highlighted in **bold** and underline, respectively.

Method	GAIA			BrowseComp-Plus			Travel Planning		
	Avg@3 (%) ↑	Pass@3 (%) ↑	Cost ($) ↓	Avg@3 (%) ↑	Pass@3 (%) ↑	Cost ($) ↓	C.S. Avg@3 (%) ↑	P.S. Avg@3 (%) ↑	Cost ($) ↓
DeepSeek-V3.2 Backbone									
Single-Agent Baselines									
ReAct Yao et al. (2023)	42.1(0.69)	60.1	**0.07**	40.2(1.04)	58.5	**0.28**	30.4(0.36)	23.3(0.47)	**0.09**
ReAct+Skills Yao et al. (2023)	43.1(0.60)	61.2	0.10	40.8(1.26)	57.5	0.32	32.5(0.54)	24.1(0.58)	0.13
HGM Wang et al. (2025)	47.3(1.21)	63.6	0.15	42.3(0.87)	59.0	0.42	36.1(0.66)	35.3(0.81)	0.18
Predefined Multi-Agent Workflows									
OWL Hu et al. (2025)	57.5(1.05)	75.6	0.24	40.8(0.87)	54.0	0.60	30.7(0.84)	18.6(0.93)	0.93
SmolAgents Roucher et al. (2025)	63.6(0.60)	78.7	0.33	42.0(0.50)	59.0	1.13	34.5(0.96)	21.7(1.04)	0.51
SMAS Lin et al. (2026a)	58.8(0.35)	76.4	0.30	42.6(0.57)	60.5	1.06	39.4(1.02)	29.2(0.70)	0.52
Automated Agent Orchestration									
GPTSwarm Zhuge et al. (2024)	38.8(0.70)	53.9	0.36	31.2(1.61)	51.5	0.60	30.4(1.08)	12.5(1.39)	0.32
MASFactory Liu et al. (2026)	41.2(0.92)	55.1	0.14	36.5(0.35)	53.0	0.49	33.6(1.20)	40.0(1.16)	0.23
Nanobot Ren (2026)	53.3(1.52)	70.3	0.16	36.0(1.75)	45.5	0.30	34.2(1.32)	27.4(1.28)	0.57
EvoMAS Hu et al. (2026)	58.2(1.26)	75.2	0.14	37.2(1.53)	52.5	0.38	30.8(1.69)	45.8(1.63)	0.36
AOrchestra Ruan et al. (2026)	54.5(0.92)	64.6	0.11	36.0(1.32)	52.0	0.32	35.5(1.57)	47.2(1.51)	0.39
EvoOrch	**65.5(1.05)**	**80.6**	0.10	**44.5(1.04)**	**63.5**	0.31	**40.3(0.87)**	**51.7(1.09)**	0.35
GPT-5 Backbone									
Single-Agent Baselines									
ReAct Yao et al. (2023)	47.8(1.60)	63.0	**0.31**	45.7(0.76)	61.5	**0.86**	37.2(1.62)	14.2(0.22)	**1.01**
ReAct+Skills Yao et al. (2023)	48.5(1.52)	64.2	0.36	46.3(1.04)	61.5	1.01	36.7(0.42)	17.4(0.19)	1.10
HGM Wang et al. (2025)	51.3(0.35)	66.0	0.43	47.2(1.04)	62.0	1.28	40.4(0.72)	25.2(0.62)	1.43
Predefined Multi-Agent Workflows									
OWL Hu et al. (2025)	62.8(0.70)	75.2	1.23	64.9(1.26)	77.5	2.31	38.1(0.78)	25.3(0.76)	3.92
SmolAgents Roucher et al. (2025)	62.8(1.26)	**79.4**	1.68	61.9(1.53)	79.5	2.16	38.2(0.60)	23.7(0.84)	3.04
SMAS Lin et al. (2026a)	58.2(1.05)	77.0	1.47	55.9(1.44)	72.0	1.96	39.1(0.96)	30.7(0.93)	2.79
Automated Agent Orchestration									
GPTSwarm Zhuge et al. (2024)	41.2(1.21)	52.7	1.42	35.2(1.53)	54.0	2.80	32.1(0.90)	35.7(1.03)	2.88
MASFactory Liu et al. (2026)	54.8(1.26)	67.9	0.90	55.9(0.35)	71.0	1.91	36.4(1.14)	39.2(1.14)	3.01
Nanobot Ren (2026)	44.0(1.40)	66.7	0.72	41.2(1.75)	49.5	1.87	37.1(1.26)	26.3(1.21)	2.61
EvoMAS Hu et al. (2026)	56.8(1.52)	69.1	1.01	60.7(0.76)	77.0	1.93	42.1(1.44)	48.3(0.38)	2.39
AOrchestra Ruan et al. (2026)	58.7(0.60)	69.1	0.81	60.0(1.32)	79.5	2.01	44.7(0.63)	46.3(1.65)	2.91
EvoOrch	**64.2(1.26)**	**79.4**	0.56	**71.3(1.05)**	**82.5**	1.84	**52.3(1.32)**	**55.9(0.71)**	2.32

Settings. Following Ruan et al. (2026), we adopt the *Homogeneous* setting in the main experiments, where the Meta Agent and sub-agents use the same backbone, as most methods do not support backbone switching. In the configuration evolution analysis, we consider the *Heterogeneous* setting, where the Meta Agent can assign different backbones to sub-agents. We use DeepSeek-V3.2 DeepSeek-AI (2025) and GPT-5 as the primary backbones, with the same task tools provided to all methods. For EvoOrch, the Meta-Agent checkpoint budget is 10, the sub-agent budget is 30 steps, and hard intervention is checked every 10 steps. For cross-task evolution, we construct a fixed experience base from trajectories collected on 400 TaskCraft cases Shi et al. (2025), which are disjoint from benchmarks, using Golden-Review for distillation and BM25 for retrieval. To avoid test-set contamination, we freeze the base before evaluation and disable test-time updates.

3.2 RESULTS

Main Results The main results are shown in Table 1. Across different settings, EvoOrch achieves the strongest overall performance while maintaining relatively low execution cost. In particular, it obtains the best Avg@3 on GAIA and BrowseComp-Plus under both backbones, and consistently achieves the best C.S. and P.S. on Travel Planning. The relatively small standard deviations across repeated runs also indicate stable performance. First, single-agent methods remain limited on long-horizon tasks that require decomposition and coordinated information gathering. Although they generally incur lower execution costs, their performance is substantially below multi-agent methods. Across the evaluated settings, EvoOrch exceeds the single-agent baselines by 18.7 points on average, showing the benefit of dynamically coordinating specialized sub-agents.

Second, predefined MAS workflows improve task performance through multi-agent collaboration, but their fixed roles and interaction structures can introduce substantial coordination and context overhead. EvoOrch improves the main task scores over SmolAgents and SMAS by 12.1 and 11.5 points on average, respectively, while reducing the average execution cost from $1.35 for SMAS to $0.89. This suggests that adapting the participating sub-agents and their organization to the evolving execution state provides a more efficient alternative to fixed workflows.

Third, compared with automated agent-orchestration methods, EvoOrch improves the scores by an average of 7.9 points over AOrchestra and 8.2 points over EvoMAS. GPTSwarm and MASFactory construct multi-agent structures before execution and therefore have limited ability to adapt to newly emerging subproblems. Nanobot can dynamically generate sub-agents but provides limited support for constructing richer task-specific dependency structures. AOrchestra can introduce new sub-agents during execution, but they cannot correct active sub-agents once they become ineffective. In contrast, EvoOrch supports both dynamic graph-level orchestration and runtime configuration revision, enabling the agent system to adapt its organization and execution as the task evolves. Additional generality, robustness, and case-study results are provided in Appendix A.2.2, Appendix A.2.9, and Appendix A.5, respectively.

Table 2: Ablation of the three evolution mechanisms in EvoOrch under DeepSeek-V3.2.

Method	GAIA			BrowseComp-Plus		
	Avg@3 ↑	Pass@3 ↑	Cost ↓	Avg@3 ↑	Pass@3 ↑	Cost ↓
EvoOrch	**65.5**	**80.6**	0.10	**44.5**	**63.5**	0.31
w/o Stru. Evo.	$53.9^{\downarrow 11.6}$	$71.8^{\downarrow 8.8}$	$\mathbf{0.08}^{\downarrow 0.02}$	$37.6^{\downarrow 6.9}$	$55.5^{\downarrow 8.0}$	$\mathbf{0.25}^{\downarrow 0.06}$
w/o Config. Evo.	$60.2^{\downarrow 5.3}$	$75.8^{\downarrow 4.8}$	$0.15^{\uparrow 0.05}$	$42.3^{\downarrow 2.2}$	$59.5^{\downarrow 4.0}$	$0.38^{\uparrow 0.07}$
w/o Cross. Evo.	$62.4^{\downarrow 3.1}$	$77.6^{\downarrow 3.0}$	$0.12^{\uparrow 0.02}$	$41.3^{\downarrow 3.2}$	$60.5^{\downarrow 3.0}$	$0.36^{\uparrow 0.05}$

Ablation Study We assess the contribution of EvoOrch's evolution mechanisms through three variants in Table 2: (1) removing *structural evolution* and restricting the Meta Agent to serial sub-agent delegation, denoted as *w/o Stru. Evo.*; (2) removing *configuration evolution* by disabling intervention over sub-agents, denoted as *w/o Config. Evo.*; and (3) removing *cross-task evolution* by disabling the retrieval of insights, denoted as *w/o Cross. Evo.*. Removing any of the three mechanisms reduces Avg@3 and Pass@3 on both benchmarks. Removing structural evolution causes the largest degradation, reducing Avg@3 by 11.6 points on GAIA and 6.9 points on BrowseComp-Plus. Although serial delegation decreases cost, it cannot dynamically construct task-specific agents and their dependency structures, demonstrating the importance of evolving the agent structures during execution. Removing configuration evolution also consistently degrades performance and increases cost. Although intervention introduces additional Meta-Agent decisions, timely revision or termination of ineffective sub-agents prevents substantially more unproductive agent computation. Finally, removing cross-task evolution reduces Avg@3 by 3.15 points on average, confirming that reusable insights provide useful guidance for future orchestration.

Token Efficiency Analysis We further compare the token efficiency of representative methods in Table 3, with counts reported in thousands. We have three observations. Compared with SMAS, EvoOrch reduces input-token consumption by 73.6% on GAIA and 75.2% on BrowseComp-Plus, indicating substantially lower context overhead. Compared with AOrchestra, EvoOrch achieves higher performance at similar cost while using fewer input and output tokens on both benchmarks. Notably, lower token consumption alone does not imply better efficiency: ReAct uses fewer tokens but performs substantially worse, while MASFactory consumes more tokens yet still underperforms EvoOrch. Overall, EvoOrch achieves a stronger performance–computation trade-off. We also conduct a wall-clock analysis in Appendix 9.

Table 3: Performance and token efficiency comparison under DeepSeek-V3.2.

Method	GAIA			BrowseComp-Plus		
	Avg@3	Cost	Tokens (in/out)	Avg@3	Cost	Tokens (in/out)
ReAct	42.1	0.07	197.1 / 16.1	40.2	0.28	1015.1 / 31.1
SMAS	58.8	0.30	1131.2 / 15.9	42.6	1.06	3945.7 / 73.8
MASFactory	41.2	0.14	396.0 / 45.3	36.5	0.49	1867.2 / 27.3
AOrchestra	54.5	0.11	309.2 / 29.1	36.0	0.32	1127.0 / 43.1
EvoOrch	65.5	0.10	298.7 / 26.3	44.5	0.31	979.0 / 42.0

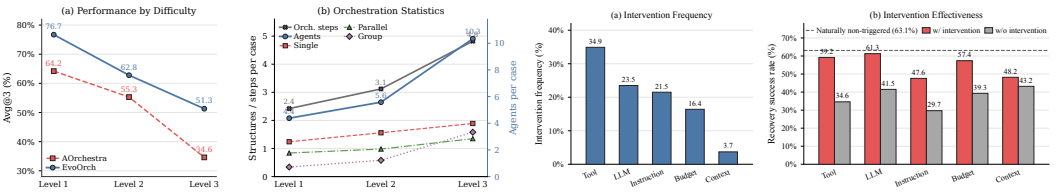
4 Analysis

We further analyze EvoOrch from two perspectives: (1) how intra-task evolution enables the system to adapt its organization and active sub-agents during execution; and (2) how historical insights support cross-task evolution and future orchestration.

4.1 Intra-Task Evolution Analysis

We analyze intra-task evolution from three aspects: whether structural evolution adapts the agent organization to the evolving task state, whether configuration evolution can recover ineffective active sub-agents, and how backbone capability should be allocated between meta-agent evolution and localized sub-agent execution.

Structure Evolution Analysis We first examine whether structural evolution adapts the agent organization to task difficulty and execution stage. As shown in Figure 2a, harder tasks induce increasingly complex orchestration patterns. From Level 1 to Level 3, the average number of orchestration checkpoints increases from 2.03 to 4.91, while the average number of spawned agents rises from 4.23 to 9.69. The frequencies of single-agent, parallel, and group-based structures also increase, with parallel and group structures becoming more prevalent on harder tasks. We further observe that the orchestration topology changes over the course of execution. At earlier checkpoints, the Meta Agent tends to instantiate parallel sub-agents to explore multiple independent directions, whereas later checkpoints more frequently introduce fan-in or other group-based structures to aggregate evi-



(a) Structural evolution across task difficulties. (b) Effectiveness of configuration interventions.

Figure 2: Analysis of intra-task evolution in EvoOrch.

dence and perform verification. These observations suggest that structural evolution does not merely increase the number of sub-agents for harder tasks, but adapts both the scale and dependency topology of the multi-agent organization according to task difficulty and the evolving execution state.

Configuration Evolution Analysis We analyze configuration evolution in a heterogeneous setting, using Qwen3.5-397B-A17B as the Meta Agent backbone and Qwen3.5-397B-A17B, Qwen3.5-27B, and Qwen3.5-35B-A3B as candidate sub-agent backbones. We consider five intervention types: changing Tools, Model, Instruction, Budget, and Context. For each intervention, we conduct a same-state comparison between the selected intervention and the original configuration. Figure 2b reports the frequency of each type and the success rates of these two configurations. Across intervention types, the weighted success rate increases from 36.3% to 56.5% (+20.2 points). Tool changes are the most frequent intervention and improve recovery from 34.6% to 59.2%, while LLM changes achieve the highest post-intervention success rate of 61.3%. Revising instructions and budgets also produces substantial gains, while context changes are less frequent and provide a smaller improvement. The same-state comparison isolates the effect of intervention and shows that targeted configuration changes substantially improve the recovery rate without reconstructing the agent graph.

Backbone Sensitivity of Evolution and Execution EvoOrch separates two distinct types of agent behavior: the Meta Agent is responsible for evolution decisions, including both structures and configurations, and sub-agents primarily perform execution. We further investigate how backbone capability should be allocated between evolution and execution on GAIA. We consider the Gemini-3 family and the Qwen3.5 family as shown in Table 4. With Qwen3.5-397B-A17B as the Meta Agent, replacing its sub-agents with 35B-A3B reduces Avg@3 by only 1.6 points while lowering the cost from 0.13 to 0.09. Conversely, using the weaker model as the Meta Agent and the stronger model as sub-agents reaches only 55.6 Avg@3, 11.6 points below the reverse allocation. A similar trend is observed in the Gemini family. These results highlight the importance of Meta-Agent capability for structural and configuration evolution, while localized execution can be flexibly delegated to cheaper sub-agent backbones. Additional comparisons with pre-planned and runtime-adaptive orchestration and budget analysis are provided in Appendix A.2.5 and Appendix A.2.3

Table 4: Backbone analysis between evolution and execution.

Meta Agent	Sub-agents	Avg@3	Pass@3	Cost
Gemini-3	Gemini-3	71.9	84.8	0.46
Gemini-3	Gemini-3-Flash	69.3	84.8	0.36
Gemini-3-Flash	Gemini-3-Flash	65.5	78.5	0.24
Gemini-3-Flash	Gemini-3	67.2	79.4	0.27
Qwen3.5-397B-A17B	397B-A17B	67.9	83.0	0.13
Qwen3.5-397B-A17B	35B-A3B	66.3	81.2	**0.09**
Qwen3.5-35B-A3B	35B-A3B	53.7	69.9	0.08
Qwen3.5-35B-A3B	397B-A17B	55.6	72.1	0.11

4.2 Cross-Task Evolution Analysis

We analyze three factors for distilling trajectories into reusable cross-task insights: (1) review signal (Golden-Review vs. Self-Review), (2) trajectory source (out-of-domain vs. in-domain), and (3) trajectory representation (raw vs. hierarchically organized). We evaluate these factors on BrowseComp-Plus using DeepSeek-V3.2 in Table 5, with 400 disjoint BrowseComp-Plus cases as the in-domain source and 400 TaskCraft cases as the out-of-domain source. First, historical insights provide useful guidance under both review settings. Compared with no cross-task insights, Self-Review improves Avg@3 by 2.0 points on average. Golden-Review further increases the gain to 3.2 points,

Table 5: Effect of different cross-task settings.

Review Set	Form	Avg@3 ↑	Δ	Pass@3 ↑	Cost ↓
No Cross-Task Insights		41.3	–	60.5	0.36
Self-Review					
OOD	Raw	42.0	+0.7	60.5	0.38
OOD	Hierarchical	43.3	**+2.0**	61.5	0.34
ID	Raw	42.8	+1.5	61.0	0.35
ID	Hierarchical	45.2	**+3.9**	62.0	0.31
Golden-Review					
OOD	Raw	42.3	+1.0	63.5	0.36
OOD	Hierarchical	44.5	**+3.2**	63.5	0.31
ID	Raw	44.8	+3.5	64.0	0.34
ID	Hierarchical	**48.5**	**+7.2**	**64.5**	**0.29**

indicating that outcome supervision can improve the quality of distilled insight, but is not necessary for cross-task transfer to be effective. Second, the trajectory source affects the strength of cross-task transfer. In-domain trajectories outperform out-of-domain trajectories by 2.3 points on average, while the gains from TaskCraft still demonstrate transfer across task distributions. Third, hierarchical trajectory organization improves distillation, it better preserves the correspondence between orchestration decisions and execution outcomes, leading to more transferable orchestration insights.

Table 6: Effects of retrieval corpus size and strategy.

Corpus Size				Retriever			
Setting	**Avg@3**	**Pass@3**	**Cost**	**Setting**	**Avg@3**	**Pass@3**	**Cost**
No Traj.	41.3	60.5	0.36	Random	41.6	62.0	0.33
25%	41.9	61.0	0.34	BM25	44.5	63.5	**0.31**
50%	44.2	62.5	**0.31**	FAISS	44.8	63.0	0.32
100%	**44.5**	**63.5**	**0.31**	Agentic	**45.3**	**64.0**	0.35

Retrieval Analysis. Table 6 examines how the number of historical trajectories and the retrieval strategy affect cross-task evolution. Increasing the trajectory corpus size generally improves performance: Avg@3 rises from 41.3 without cross-task evolution to 44.5 with the full corpus. Most of the gain is already obtained with 50% of the trajectories, suggesting diminishing returns as more historical trajectories become available. We then examine whether the gains depend on retrieving relevant insights. Random retrieval yields only a marginal improvement, whereas BM25 retrieval increases Avg@3 to 44.5, indicating that cross-task transfer depends on insights relevant to the task rather than simply introducing additional information. Among relevance-based retrieval methods, the choice of retriever has a relatively small effect. Compared with BM25, Agentic Search further improves Avg@3 by 0.8 points, while incuring a higher execution cost. These results suggest that the relevance of retrieved insights is more important than the choice of retriever. Additional analyses on retrieved-insight controls, offline evolution are provided in Appendices 4.2, A.2.6.

5 Related Work

Multi-Agent Systems LLM-based MASs have been widely studied for long-horizon tasks Li et al. (2023); Wu et al. (2023). Early systems Hu et al. (2025); Roucher et al. (2025); Zhao et al. (2025) rely on manually specified agent roles and workflows, which require substantial design effort. Recent work therefore explores automatic orchestration Hong et al. (2024); Yang et al. (2025); Yu (2026). Some methods Zhuge et al. (2024); Liu et al. (2026); Zhang et al. (2026a) construct multi-agent workflows before execution through task decomposition or workflow optimization, while AOrchestra and runtime agent frameworks Ruan et al. (2026); Ren (2026); Anthropic (2025) dynamically instantiate sub-agents during execution. However, existing approaches provide limited support for jointly adapting task-specific dependency structures and revising active sub-agent configurations. EvoOrch addresses this gap through structural and configuration evolution during execution.

Evolving Agents Existing approaches to agent evolution can be broadly divided into training-based and training-free methods. Training-based methods improve agent policies through RL Dang et al. (2025); Feng et al. (2026); Yuan et al. (2026); Zhang (2026) or SFT Ruan et al. (2026). Training-free methods instead improve future behavior by reusing or refining previous executions. Some retain or optimize high-performing workflows Zhang et al. (2025); Wang et al. (2025), while others evolve prompts through iterative feedback Agrawal et al. (2026) or distill trajectories into reusable skills and meta-level experience Lin et al. (2026b); Saeidi et al. (2026); Wang et al. (2026). Recent work further explores evolving multi-agent configurations, orchestration, and agent behaviors from feedback Hu et al. (2026); Li & Ramakrishnan (2026); Sampath & Baskaran (2026). Most existing methods focus on a particular form of agent evolution, such as workflow, prompt optimization, or trajectory-based experience reuse EvoOrch instead combines runtime evolution of agent structures and configurations with cross-task distillation of reusable orchestration insights.

6 Conclusion

In this paper, we propose **EvoOrch**, an evolving orchestration framework for LLM-based multi-agent systems. Within a task, EvoOrch dynamically evolves both task-specific agent structures and active sub-agent configurations through a persistent Meta Agent and MAOP. Across tasks, it hierarchically organizes completed trajectories and distills reusable orchestration insights to guide future tasks. Experiments show consistent improvements over the current automated orchestration methods, while maintaining a favorable trade-off between performance and cost.

AI USE STATEMENT

Generative AI tools were used to assist with language polishing, code debugging, and the presentation and formatting of parts of the manuscript. They were not used to generate experimental data, determine evaluation results, or make final scientific conclusions. All AI-assisted text and code were reviewed by the authors, and the experimental implementations, reported results, and claims were independently checked for correctness and consistency with the underlying experiments. The authors take full responsibility for the final content of this work, including any text, code, or other artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We provide detailed information to support the reproducibility of EvoOrch. The main paper describes the framework, execution protocol, experimental settings, and evaluation procedure. The appendix further provides the complete execution algorithm, benchmark and tool configurations, model pricing and implementation settings, the full MAOP operation set, evaluation metric definitions, representative case traces, and the prompt templates used for the Meta Agent, sub-agents, intervention, and trajectory review. We also report the model backbones, execution budgets, retrieval settings, and cost calculation procedure used in our experiments.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning, 2026. URL https://arxiv.org/abs/2507.19457.

Anthropic. Claude code. https://www.anthropic.com/, 2025. Agentic coding system.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Jiangning Liu, Wenwei Zhang, Kai Chen, and Feng Zhao. Mindsearch: Mimicking human minds elicits deep ai searcher, 2025a. URL https://arxiv.org/abs/2407.20183.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent. *arXiv preprint arXiv:2508.06600*, 2025b.

Yufan Dang, Chen Qian, Xueheng Luo, Jingru Fan, Zihao Xie, Ruijie Shi, Weize Chen, Cheng Yang, Xiaoyin Che, Ye Tian, Xuantang Xiong, Lei Han, Zhiyuan Liu, and Maosong Sun. Multi-agent collaboration via evolving orchestration, 2025. URL https://arxiv.org/abs/2505.19591.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *CoRR*, abs/2501.12948, 2025. doi: 10.48550/ARXIV.2501.12948. URL https://doi.org/10.48550/arXiv.2501.12948.

Yichao Feng, Haoran Luo, Zhenghong Lin, Yiqun Sun, Pengfei Wei, Lawrence B. Hsieh, and Anh Tuan Luu. Orchmas: Orchestrated reasoning with multi collaborative heterogeneous scientific expert structured agents, 2026. URL https://arxiv.org/abs/2603.03005.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Mengkang Hu, Yuhang Zhou, Wendong Fan, Yuzhou Nie, Bowei Xia, Tao Sun, Ziyu Ye, Zhaoxuan Jin, Yingru Li, Qiguang Chen, Zeyu Zhang, Yifeng Wang, Qianshuo Ye, Bernard Ghanem, Ping

Luo, and Guohao Li. Owl: Optimized workforce learning for general multi-agent assistance in real-world task automation, 2025. URL https://arxiv.org/abs/2505.23885.

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

Lisheng Huang, Yichen Liu, Jinhao Jiang, Rongxiang Zhang, Jiahao Yan, Junyi Li, and Wayne Xin Zhao. Manusearch: Democratizing deep search in large language models with a transparent and open multi-agent framework, 2025. URL https://arxiv.org/abs/2505.18105.

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. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Sha Li and Naren Ramakrishnan. Experience as a compass: Multi-agent rag with evolving orchestration and agent prompts, 2026. URL https://arxiv.org/abs/2604.00901.

Fulin Lin, Shaowen Chen, Ruishan Fang, Hongwei Wang, and Tao Lin. Stop wasting your tokens: Towards efficient runtime multi-agent systems. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=pzFhtpkabh.

Hehai Lin, Qi Yang, and Chengwei Qin. Skill-mas: Evolving meta-skill for automatic multi-agent systems, 2026b. URL https://arxiv.org/abs/2606.18837.

Yang Liu, Jinxuan Cai, Yishen Li, Qi Meng, Zedi Liu, Xin Li, Chen Qian, Chuan Shi, and Cheng Yang. Masfactory: A graph-centric framework for orchestrating llm-based multi-agent systems with vibe graphing. *arXiv preprint arXiv:2603.06007*, 2026. doi: 10.48550/arXiv.2603.06007. URL https://arxiv.org/abs/2603.06007.

Yougang Lyu, Xi Zhang, Xinhao Yi, Yuyue Zhao, Shuyu Guo, Wenxiang Hu, Jan Piotrowski, Jakub Kaliski, Jacopo Urbani, Zaiqiao Meng, Lun Zhou, and Xiaohui Yan. Evoscientist: Towards multi-agent evolving ai scientists for end-to-end scientific discovery, 2026. URL https://arxiv.org/abs/2603.08127.

Mike A Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Jenia Jitsev, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants, 2023. URL https://arxiv.org/abs/2311.12983.

Xubin Ren. nanobot: The ultra-lightweight personal ai agent. https://github.com/HKUDS/nanobot, 2026. Accessed: 2026-05-12.

Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. 'smolagents': a smol library to build great agentic systems. https://github.com/huggingface/smolagents, 2025.

Jianhao Ruan, Zhihao Xu, Yiran Peng, Fashen Ren, Zhaoyang Yu, Xinbing Liang, Jinyu Xiang, Yongru Chen, Bang Liu, Chenglin Wu, Yuyu Luo, and Jiayi Zhang. Aorchestra: Automating sub-agent creation for agentic orchestration, 2026. URL https://arxiv.org/abs/2602.03786.

Amir Saeidi, Venkatesh Mishra, Souradeep Mukhopadhyay, Gaowen Liu, Ali Payani, Jayanth Srinivasa, and Chitta Baral. Fama: Failure-aware meta-agentic framework for open-source llms in interactive tool use environments, 2026. URL https://arxiv.org/abs/2604.25135.

Sathish Sampath and Anuradha Baskaran. Adaptive orchestration: Scalable self-evolving multi-agent systems, 2026. URL https://arxiv.org/abs/2601.09742.

Dingfeng Shi, Jingyi Cao, Qianben Chen, Weichen Sun, Weizhen Li, Hongxuan Lu, Fangchen Dong, Tianrui Qin, King Zhu, Minghao Yang, Jian Yang, Ge Zhang, Jiaheng Liu, Changwang Zhang, Jun Wang, Yuchen Eleanor Jiang, and Wangchunshu Zhou. Taskcraft: Automated generation of agentic tasks, 2025. URL https://arxiv.org/abs/2506.10055.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), March 2024. ISSN 2095-2236. doi: 10.1007/s11704-024-40231-1. URL http://dx.doi.org/10.1007/s11704-024-40231-1.

Wenyi Wang, Piotr Piękos, Li Nanbo, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley-gödel machine: Human-level coding agent development by an approximation of the optimal self-improving machine, 2025. URL https://arxiv.org/abs/2510.21614.

Xiaoxing Wang, Ning Liao, Shikun Wei, Chen Tang, and Feiyu Xiong. Autoagent: Evolving cognition and elastic memory orchestration for adaptive agents, 2026. URL https://arxiv.org/abs/2603.09716.

Ryan Wong, Jiawei Wang, Junjie Zhao, Li Chen, Yan Gao, Long Zhang, Xuan Zhou, Zuo Wang, Kai Xiang, Ge Zhang, Wenhao Huang, Yang Wang, and Ke Wang. Widesearch: Benchmarking agentic broad info-seeking, 2025. URL https://arxiv.org/abs/2508.07999.

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.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

Yingxuan Yang, Huacan Chai, Shuai Shao, Yuanyi Song, Siyuan Qi, Renting Rui, and Weinan Zhang. Agentnet: Decentralized evolutionary coordination for llm-based multi-agent systems, 2025. URL https://arxiv.org/abs/2504.00587.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

Geunbin Yu. Adaptorch: Task-adaptive multi-agent orchestration in the era of llm performance convergence, 2026. URL https://arxiv.org/abs/2602.16873.

Wenzhen Yuan, Wutao Xiong, Fanchen Yu, Shengji Tang, Ting Liu, Tao Chen, Peng Ye, Yuzhuo Fu, Wanli Ouyang, and Lei Bai. Small model as master orchestrator: Learning unified agent-tool orchestration with parallel subtask decomposition, 2026. URL https://arxiv.org/abs/2604.17009.

Chenchen Zhang. Reinforcement learning for llm-based multi-agent systems through orchestration traces, 2026. URL https://arxiv.org/abs/2605.02801.

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, 2025. URL https://arxiv.org/abs/2410.10762.

Xing Zhang, Yanwei Cui, Guanghui Wang, Wei Qiu, Ziyuan Li, Fangwei Han, Yajing Huang, Hengzhi Qiu, Bing Zhu, and Peiyang He. Verified multi-agent orchestration: A plan-execute-verify-replan framework for complex query resolution, 2026a. URL https://arxiv.org/abs/2603.11445.

Yinger Zhang, Shutong Jiang, Renhao Li, Jianhong Tu, Yang Su, Lianghao Deng, Xudong Guo, Chenxu Lv, and Junyang Lin. Deepplanning: Benchmarking long-horizon agentic planning with verifiable constraints, 2026b. URL https://arxiv.org/abs/2601.18137.

Di Zhao, Longhui Ma, Siwei Wang, Miao Wang, and Zhao Lv. Cola: A scalable multi-agent framework for windows ui task automation, 2025. URL https://arxiv.org/abs/2503.09263.

Peilin Zhou, Bruce Leon, Xiang Ying, Can Zhang, Yifan Shao, Qichen Ye, Dading Chong, Zhiling Jin, Chenxuan Xie, Meng Cao, et al. Browsecomp-zh: Benchmarking web browsing ability of large language models in chinese. *arXiv preprint arXiv:2504.19314*, 2025.

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

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

A Appendix

A.1 Limitations

Our current study has several limitations: First, due to the increasing complexity of environment setup, our evaluation mainly focuses on deep-research and planning tasks. We have not yet extensively evaluated EvoOrch on a broader range of agentic tasks, such as open-ended Claw-style tasks. Future work will investigate how efficient multi-agent orchestration can be adapted to these settings.

Second, our current cross-task evolution mainly operates at the level of trajectory distillation and skill reuse. Although historical trajectories can improve future orchestration through reusable insights, they do not directly modify the underlying agent harness. This limits the framework's ability to move toward more general recursive self-improvement. In future work, we plan to extend cross-task experience from strategy-level guidance to executable harness-level updates, including code-level modifications to the agent system itself.

Third, our experiments primarily use large LLMs as the Meta Agent and sub-agent backbones. The behavior of EvoOrch with smaller models remains less explored, especially because orchestration and long-horizon execution may impose different capability requirements. Future work will study more efficient deployment with small LLMs and explore post-training methods for improving both their orchestration and execution capabilities.

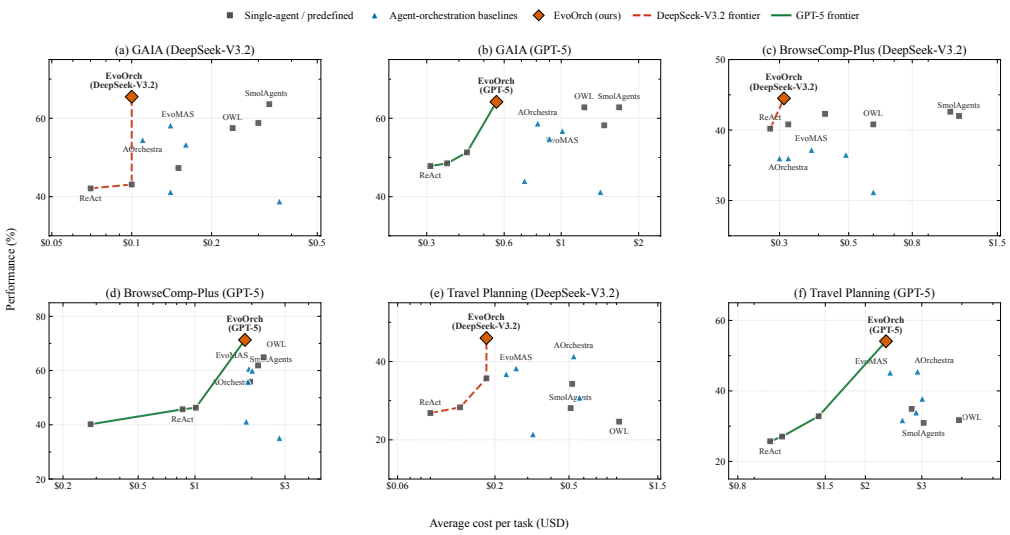


Figure 3: Pareto frontier analysis of the main experiment across three benchmarks and two backbones. For Travel Planning, the performance score is the average of C.S. and P.S.

A.2 Additional Experiments

A.2.1 Pareto Analysis

To better characterize the performance-cost trade-off, we plot the main performance score against the average execution cost in Figure 3. The six panels show that EvoOrch lies on the Pareto frontier for every benchmark-backbone setting. On GAIA, EvoOrch reaches the highest score with only \$0.10 under DeepSeek-V3.2 and \$0.56 under GPT-5, respectively. This places it above higher-cost multi-agent baselines such as SmolAgents, SMAS, GPTSwarm, and EvoMAS, and also above the previous automated orchestration method AOrchestra. Although ReAct-based methods are cheaper in several panels, their performance is substantially lower, so they represent low-budget but low-accuracy operating points rather than high-performance alternatives.

The same pattern is more pronounced on BrowseComp-Plus and Travel Planning. On BrowseComp-Plus, EvoOrch achieves the best score under both backbones while costing less than most multi-agent and orchestration baselines; under GPT-5, for example, it improves over AOrchestra while reducing the cost from \$2.01 to \$1.84. On Travel Planning, where the plotted score averages C.S. and P.S., EvoOrch also occupies the high-performance frontier: it improves over AOrchestra from 41.35 to 46.00 at much lower cost under DeepSeek-V3.2, and from 45.50 to 54.10 while reducing cost under GPT-5. These results are consistent with the main-table observations: EvoOrch's gains are not obtained by simply spending more tokens, but by dynamically allocating computation, avoiding fixed workflow overhead, and intervening when sub-agents become ineffective. The Pareto analysis therefore provides complementary evidence that protocol-guided dynamic orchestration improves both task effectiveness and cost utilization.

A.2.2 Generality Experiment

We further evaluate the generality of EvoOrch on three additional benchmarks, including WideSearch, BrowseComp-Zh, and Terminal-Bench 2.0, with the results shown in Table 7. These benchmarks cover complementary task settings, ranging from broad information seeking and Chinese web browsing to long-horizon execution in interactive terminal environments. For WideSearch, we use Item-F1 as the task-level metric and report its average over three runs (Avg@3), following the benchmark evaluation protocol. EvoOrch consistently achieves the best task performance across all three benchmarks. On WideSearch, EvoOrch reaches an Avg@3 of 46.2, outperforming the strongest baseline, SmolAgents, by 6.7 points. On BrowseComp-Zh, EvoOrch achieves 49.5 Avg@3, improving over SMAS by 3.5 points while remaining substantially more cost-efficient than predefined multi-agent workflows. On Terminal-Bench 2.0, EvoOrch obtains 32.8 Avg@3, exceed-

Table 7: Generality results based on DeepSeek-V3.2.

Method	WideSearch		BrowseComp-Zh		Terminal-Bench 2.0	
	Avg@3 ↑	Cost ↓	Avg@3 ↑	Cost ↓	Avg@3 ↑	Cost ↓
Single-Agent						
ReAct	35.0	0.20	34.3	0.21	13.5	0.13
ReAct+Skills	36.2	0.23	35.3	0.19	13.1	0.15
Predefined MAS						
SmolAgents	39.5	0.87	45.7	1.21	27.7	0.36
SMAS	36.0	0.72	46.0	1.01	26.1	0.34
Automated Orchestration						
MASFactory	31.5	0.59	44.6	0.45	22.8	0.31
AOrchestra	37.5	0.17	41.9	0.16	29.7	0.25
Nanobot	33.4	0.45	–	–	–	–
EvoOrch	**46.2**	0.19	**49.5**	0.20	**32.8**	0.31

ing the strongest baseline, AOrchestra, by 3.1 points. While its execution cost is slightly higher than AOrchestra on this benchmark, it remains lower than the costs of the predefined multi-agent systems. Overall, these results demonstrate that EvoOrch generalizes beyond the main English web-based benchmarks. Its gains consistently extend across broad search, multilingual browsing, and interactive terminal tasks, suggesting that dynamic orchestration can adapt effectively to substantially different task environments and execution requirements.

Table 8: Effect of execution budgets on GAIA using DeepSeek-V3.2.

Meta Budget B_M	Sub-agent Budget B_S	Avg@3 ↑	Pass@3 ↑	Cost ↓
8	20	64.1	78.1	0.08
8	30	65.0	80.0	0.10
8	40	64.4	79.4	0.11
10	20	65.3	79.4	0.10
10	**30**	**65.5**	**80.6**	**0.10**
10	40	65.7	80.6	0.11
12	20	65.5	80.0	0.11
12	30	65.7	80.6	0.11
12	40	65.7	80.6	0.13

A.2.3 Execution Budget Analysis

We analyze the effect of execution budgets on GAIA using DeepSeek-V3.2. Specifically, we vary the Meta-Agent orchestration budget $B_M \in \{8, 10, 12\}$ and the sub-agent execution budget $B_S \in \{20, 30, 40\}$. As shown in Table 8, increasing the execution budgets generally improves performance, but the gains gradually saturate as more computation is allocated. Increasing B_M from 8 to 10 yields a consistent improvement across different sub-agent budgets, while further increasing it to 12 provides only marginal gains. Similarly, increasing B_S from 20 to 30 is generally beneficial, whereas increasing it further to 40 brings little additional improvement and increases execution cost. Increasing either budget beyond $(B_M = 10, B_S = 30)$ improves Avg@3 by at most 0.2 points and does not improve Pass@3, while increasing execution cost. We therefore use $(B_M = 10, B_S = 30)$ as the default setting in the main experiments. This also indicates that EvoOrch's gains are not sensitive to a narrowly tuned execution budget.

A.2.4 Wall-clock Analysis

We further compare the wall-clock execution time of SmolAgents, AOrchestra, and EvoOrch on GAIA using DeepSeek-V3.2. Table 9 reports the average wall-clock time across different GAIA difficulty levels. On Level 1, where tasks generally require less complex coordination, EvoOrch completes each case in 701.94

Table 9: Average wall-clock execution time (seconds) on GAIA using DeepSeek-V3.2.

Level	EvoOrch	AOrchestra	SmolAgents
Level 1	701.94	798.24	754.86
Level 2	1,194.32	1,076.91	1,152.80
Level 3	1,493.04	1,246.17	1,523.43

Table 10: Intra-task comparison on GAIA.

Method	Avg@3 ↑	Pass@3 ↑	Cost ↓
MindSearch	41.6	54.7	**0.25**
ManuSearch	59.4	76.4	1.09
Claude Code CLI	40.0	50.3	–
Nanobot	44.0	66.7	0.72
EvoOrch	**64.2**	**79.4**	0.56

Table 11: Comparison with offline workflow and prompt evolution on a held-out GAIA split using GPT-5.

Method	Avg@3 ↑	Pass@3 ↑	Cost ↓
AFlow	43.6	56.0	0.62
GEPA	47.6	62.9	0.55
EvoOrch	**65.6**	**80.0**	**0.53**

seconds on average, compared with 798.24 seconds for AOrchestra and 754.86 seconds for SmolAgents. This suggests that when the solution path is relatively straightforward, EvoOrch can efficiently allocate a small number of task-relevant sub-agents without introducing substantial coordination overhead. As task difficulty increases, EvoOrch incurs additional wall-clock latency. The additional latency mainly arises from configuration evolution, which introduces short Meta Agent–sub-agent interactions for monitoring, diagnosis, and runtime revision. These additional synchronization and interaction steps can increase end-to-end latency even when the overall token consumption and inference cost remain low. Notably, on Level 3, EvoOrch still achieves lower wall-clock time than SmolAgents (1,493.04 vs. 1,523.43 seconds). This reflects a performance–latency tradeoff: EvoOrch introduces additional runtime supervision and configuration adaptation to identify and revise ineffective sub-agent executions, at the cost of additional interaction and synchronization overhead.

A.2.5 Comparison with Pre-Planned and Runtime-Adaptive Orchestration

We first compare EvoOrch with representative pre-planned and runtime-adaptive orchestration methods. Specifically, we consider pre-planned deep-search systems, MindSearch Chen et al. (2025a) and ManuSearch Huang et al. (2025), which construct structured search or multi-agent workflows before execution, and runtime-adaptive systems, Nanobot Ren (2026) and Claude Code Anthropic (2025), which can dynamically delegate work during execution. In contrast, EvoOrch jointly supports structural evolution of task-specific agent dependencies and configuration evolution of active sub-agents. As shown in Table 10, stronger pre-planned workflows can already provide substantial gains: ManuSearch reaches 59.4 Avg@3 and 76.4 Pass@3. However, EvoOrch further improves these scores to 64.2 and 79.4, respectively, while reducing the execution cost from $1.09 to $0.56. Runtime-adaptive systems such as Nanobot provide additional flexibility during execution, but remain substantially below EvoOrch, with 44.0 Avg@3 and 66.7 Pass@3. These results highlight the difference between partial runtime adaptation and full intra-task evolution. Pre-planned methods provide structured coordination but cannot adapt the agent organization as execution unfolds, while lightweight runtime delegation provides limited support for jointly evolving dependency structures and revising active sub-agent configurations. EvoOrch combines structural and configuration evolution throughout execution.

A.2.6 Comparison with Offline Workflow and Prompt Evolution

We further compare EvoOrch with AFlow Zhang et al. (2025) and GEPA Agrawal et al. (2026), which improve agent behavior through offline workflow or prompt evolution before evaluation. For this experiment, we split the 165 GAIA validation cases into 65 cases for evolution and 100 held-out cases for evaluation. Therefore, the results are intended for within-table comparison and are not directly comparable to the main GAIA results. As shown in Table 11, both offline evolution methods remain below EvoOrch. AFlow and GEPA achieve 43.6 and 47.6 Avg@3, respectively, whereas EvoOrch reaches 65.6 Avg@3 and 80.0 Pass@3 at a comparable execution cost. The key

distinction is that these methods optimize reusable workflows or prompts before evaluation, while EvoOrch can additionally continue adapting its agent structure and active sub-agent configurations during test-time execution. These results complement the main analysis by showing that offline cross-task evolution and intra-task evolution provide complementary forms of adaptation.

A.2.7 Controls for Retrieved Insights

Finally, we examine whether EvoOrch's gains can be explained solely by access to retrieved historical insights. We construct two controls using the same retrieved insights: *ReAct + Retrieved Insights* converts them into task-execution hints for a single ReAct agent, while *AOrchestra + Role Insights* converts the orchestration guidance into reusable role-level skills for AOrchestra. As shown in Table 12, retrieved insights improve both underlying systems. Adding retrieved insights to ReAct increases Avg@3 from 47.8 to 48.5, while providing role-level insights to AOrchestra improves Avg@3 from 58.7 to 60.4 and Pass@3 from 69.1 to 71.7. This confirms that the distilled cross-task insights themselves provides useful guidance. Nevertheless, both enhanced baselines remain below EvoOrch, which achieves 64.2 Avg@3 and 79.4 Pass@3. This suggests that retrieved insights are most effective when translated into runtime evolution decisions, including structural evolution through task-specific graph construction and configuration evolution through active sub-agent revision, rather than being used only as static execution guidance.

Table 12: Controls for retrieved insights on GAIA using GPT-5.

Method	**Avg@3** ↑	**Pass@3** ↑	**Cost** ↓
ReAct	47.8	63.0	**0.31**
ReAct + Retrieved Insights	48.5	64.2	0.33
AOrchestra	58.7	69.1	0.81
AOrchestra + Role Insights	60.4	71.7	0.77
EvoOrch	**64.2**	**79.4**	0.56

A.2.8 Cross-Task Analysis of GAIA

We further examine whether trajectory-based evolution transfers beyond BrowseComp-Plus. Since GAIA contains only 165 validation examples, we use TaskCraft as an out-of-domain trajectory source and compare Self-Review and Golden-Review using raw trajectories and distilled insights. As shown in Table 13, distilled insights outperform raw trajectories under both review settings. This trend is consistent with the BrowseComp-Plus cross-task analysis in the main paper and supports the claim that compact orchestration insights are more reusable than raw execution traces.

Table 13: Out-of-domain cross-task transfer on GAIA using TaskCraft trajectories and DeepSeek-V3.2.

Review Signal	**Representation**	**Avg@3** ↑	**Pass@3** ↑	**Cost** ↓
Self-Review	Raw Trajectory	62.8	77.6	0.14
Self-Review	Distilled Insight	64.2	79.4	0.11
Golden-Review	Raw Trajectory	63.8	79.4	0.12
Golden-Review	Distilled Insight	**65.5**	**80.6**	**0.10**

A.2.9 Robustness Analysis

Each result is averaged over three independent runs per case. As shown in Table 14, EvoOrch consistently improves Pass@k across different k. In paired item-level comparisons with AOrchestra, EvoOrch wins on 62.1% of GAIA cases and 56.5% of BrowseComp-Plus cases, suggesting that the gains are distributed across examples rather than driven by a small subset.

Table 14: Pass@k stability under DeepSeek-V3.2.

Benchmark	Method	Pass@1	Pass@2	Pass@3
GAIA	ReAct	42.1	55.2	60.1
	AOrchestra	54.5	60.0	64.6
	EvoOrch	**65.5**	**75.8**	**80.6**
BrowseComp-Plus	ReAct	40.2	53.2	58.5
	AOrchestra	36.0	46.5	52.0
	EvoOrch	**44.5**	**57.6**	**63.5**

Algorithm 1: Execution loop of EvoOrch.

Require: Task x, tool pool $\mathbb{T}$, experience base $\mathcal{B}$, maximum checkpoints K, maximum sub-agent steps S
Ensure: Final answer $\hat{y}$

Initialize $G(0) \leftarrow \langle \{\mathcal{A}_M\}, \emptyset, \mathcal{S}(0) \rangle$ and trajectory $\Gamma(x)$
for $t = 1, \ldots, K$ **do**
  $\mathcal{A}_M$ observes $G(t-1)$ and selects $a(t) \in \{\text{RETRIEVAL}, \text{EXECUTION}, \text{ORCHESTRATION}, \text{ANSWER}\}$
  if $a(t) = \text{RETRIEVAL}$ **then**
    Retrieve relevant orchestration insights from $\mathcal{B}$ and update $\mathcal{S}(t)$
  else if $a(t) = \text{EXECUTION}$ **then**
    $\mathcal{A}_M$ executes an atomic tool operation and updates $\mathcal{S}(t)$
  else if $a(t) = \text{ORCHESTRATION}$ **then**
    Instantiate task-specific sub-agents and construct their dependency graph
    Execute ready sub-agents according to dependencies, allowing independent agents to run in parallel
    for all active sub-agents v_i **do**
      Execute v_i within its local budget S
      if intervention is triggered **then**
        $\mathcal{A}_M$ selects CONTINUE, CHANGE, or STOP
        if CHANGE **then**
          Revise the diagnosed configuration of v_i
        else if STOP **then**
          Terminate v_i
        end if
      end if
    end for
    Summarize sub-agent outputs and update $\mathcal{S}(t)$
  else if $a(t) = \text{ANSWER}$ **then**
    $\hat{y} \leftarrow \text{ANSWER}(\mathcal{A}_M, \mathcal{S}(t-1))$
    Store $\Gamma(x)$ for cross-task evolution
    return $\hat{y}$
  end if
  Update graph state $G(t)$ and trajectory $\Gamma(x)$
end for
$\hat{y} \leftarrow \text{ANSWER}(\mathcal{A}_M, \mathcal{S}(K))$
Store $\Gamma(x)$ for cross-task evolution
return $\hat{y}$

A.3 METHOD DETAILS

Execution Algorithm Algorithm 1 summarizes the execution loop of EvoOrch. Given a task x, EvoOrch initializes the graph with the Meta Agent $\mathcal{A}_M$ and an empty sub-agent set. At each checkpoint, the Meta Agent selects RETRIEVAL, EXECUTION, or ORCHESTRATION, or directly produces the final answer. Orchestrated sub-agents are executed according to their dependency graph, with INTERVENTION triggered during execution when necessary. Their outputs are incorporated into the global information state, and the process continues until the task is completed or the checkpoint budget is exhausted. The resulting trajectory is stored for subsequent cross-task evolution, while the experience base remains fixed during evaluation.

A.4 Experimental Settings

A.4.1 Evaluation Settings

This section summarizes the evaluation protocol used in our experiments, including the datasets, compared baselines, cost calculation, answer cleansing procedure, and implementation details.

Dataset We evaluate EvoOrch on a diverse set of agentic benchmarks covering general reasoning, deep research, long-horizon planning, broad information seeking, and multilingual web browsing.

- **GAIA** Mialon et al. (2023) evaluates general assistant capabilities including reasoning, web browsing, multimodal understanding, and tool use. We use its public validation split of 165 questions across three difficulty levels.
- **BrowseComp-Plus** Chen et al. (2025b) is a deep-research benchmark over a fixed curated corpus. We sample 200 test cases for evaluation and an additional 400 disjoint cases for in-domain trajectory retrieval.
- **Travel Planning** Zhang et al. (2026b) evaluates long-horizon itinerary generation under constraints on schedules, budgets, transportation, accommodation, and activities.
- **WideSearch** Wong et al. (2025) evaluates broad information collection and verification across diverse domains. We use its 100 English tasks for evaluation.
- **BrowseComp-Zh** Zhou et al. (2025) contains 289 Chinese web-browsing questions across 11 domains and evaluates multi-hop retrieval and reasoning.
- **Terminal-Bench 2.0** Merrill et al. (2026) contains 89 challenging terminal-based tasks inspired by real-world workflows and evaluates agents' long-horizon tool use and execution capabilities in computer environments.
- **TaskCraft** Shi et al. (2025) provides 400 external tasks for constructing out-of-domain historical trajectories, which are distilled into orchestration insights for cross-task evolution.

Baselines We compare EvoOrch with several representative baselines, including single-agent reasoning, human-predefined multi-agent systems, and automated agent-orchestration systems.

- **OWL** Hu et al. (2025) uses a hierarchical planner–coordinator–worker multi-agent workflow.
- **Smolagents** Roucher et al. (2025) is a lightweight framework for tool-using agents.
- **SMAS** Lin et al. (2026a) introduces a meta-level supervisor to monitor and intervene in multi-agent execution.
- **GPTSwarm** Zhuge et al. represents multi-agent workflows as optimizable graphs.
- **MASFactory** Liu et al. (2026) generates executable multi-agent graphs from natural-language workflow specifications.
- **EvoMAS** Hu et al. (2026) evolves multi-agent configurations through selection, mutation, and crossover.
- **AOrchestra** Ruan et al. (2026) dynamically creates specialized sub-agents at runtime using a unified agent abstraction.
- **Nanobot** Ren (2026) is a lightweight agent framework with a compact runtime loop and practical memory/tool support.
- **MindSearch** Chen et al. (2025a) and **ManuSearch** Huang et al. (2025) are deep-search frameworks that organize search and evidence collection through structured multi-agent workflows.
- **AFlow** Zhang et al. (2025), **GEPA** Agrawal et al. (2026), and **HGM** Wang et al. (2025) represent workflow optimization, prompt evolution, and evolving-agent approaches, respectively.
- **Claude Code CLI** is included as a modern general-purpose agent baseline, with benchmark tools exposed through skills for GAIA.

Table 15: Input and output costs of the evaluated model backbones, reported in USD per 1M tokens.

Model	Input Cost	Output Cost
DeepSeek-V3.2	0.25	1.00
GPT-5	1.25	10.00
GPT-5-mini	0.25	2.00
Gemini-3-Flash	0.50	3.00
Gemini-3-Pro	2.00	4.00
Qwen3.5-397B-A17B	0.30	2.40
Qwen3.5-27B	0.20	1.80
Qwen3.5-35B-A3B	0.15	1.20

Tool Use Following AOrchestra (Ruan et al., 2026), we equip all methods with the same benchmark-specific tool set for fair comparison. The full benchmark tool configurations are deferred to Table A.6.1.

Cost Calculation All token usage statistics in our experiments are collected through the corresponding model APIs. The inference cost is calculated based on the official API pricing of each model, using the recorded input and output token counts. Table 15 lists the pricing used in our experiments, reported in USD per 1M tokens. Following Ruan et al. (2026), we restrict candidate backbones within the same model family and provide model price tags as a simple proxy for capability in prompts to reduce selection bias.

Answer Cleansing Different agent systems may generate final answers in different formats. We first apply rule-based matching to judge whether the generated answer matches the gold answer. For answers that cannot be matched by rules, following AOrchestra Ruan et al. (2026), BrowseComp-Plus Chen et al. (2025b) and Travel Planning Zhang et al. (2026b), we use Gemini-3 to extract the prediction from the model output and check whether the normalized prediction is semantically consistent with the gold answer. This procedure is applied to all methods under the same setting to ensure fairness.

Implementations For baselines with official implementations, we follow their released configurations whenever available and rerun the benchmarks. When official configurations are unavailable, we reproduce them using the same backbone model, tool access, maximum execution budget, and evaluation protocol as EvoOrch. For each method, we run each test case three times with temperature set to 1.0 when the temperature parameter is supported. All reported costs are computed from input and output token usage under the same pricing table.

A.5 Case Study

We present a case study from BrowseComp-Plus to illustrate how EvoOrch solves a complex web reasoning task. The task requires identifying a venue whose first performance occurred in the 1930s with an attendance between 3,700 and 4,500, whose capacity was expanded in the 1980s to between 5,500 and 6,500, and whose surrounding area was established in the same year as a play written by an author born in the 1870s. The expected answer is **6187**.

This case is challenging because the clues are distributed across different domains and no single clue directly reveals the answer. EvoOrch first retrieves a prior orchestration pattern that suggests broad parallel exploration followed by chain-style verification. Although the retrieved case does not contain the answer, it provides a useful planning prior. Following this pattern, the Meta Agent decomposes the task into independent clue families: one branch identifies Alfred Jarry's *Ubu roi* and its year, while another investigates the venue history and capacity timeline of the Greek Theatre in Los Angeles. The Meta Agent then links these partial findings through Griffith Park's establishment year and verifies the final capacity after the 1983 expansion.

During execution, an intermediate verifier makes little progress. EvoOrch recovers by issuing a stricter cross-verification task with known entities and explicit constraints, rather than restarting

the entire process. This leads to the verified answer **6187**. The case shows that retrieval in EvoOrch serves as an orchestration prior rather than an answer source: it guides the system toward the strategy of "parallel search first, chain verification later", enabling more reliable evidence aggregation for complex multi-hop tasks.

A.5.1 DETAILED CASE TRACE

Case Metadata (JSON)


```
{
  "task_id": "browsecomp_plus_681",
  "dataset": "BrowseComp-Plus",
  "question": "There is a venue and after its establishment, when the first performance took place in
  the 1930s, the crowd attendance ranged approximately between 3700 to 4500 (exclusive). In the 1980s
  the capacity was increased and ranged between 5500-6500 (exclusive). The area where the venue is
  located, was established in the 1890s. During the same time in the 1890s, the exact year when that
  area was created, a play was produced by a writer who was born in the 1870s. The play was published
  around variety of titles. The main character in this play was seen as a gluttonous and rapacious
  person. Can you tell me the total number of capacity of that venue, after it got raised?",
  "expected_answer": "6187",
  "main_model": "Deepseek-V3.2",
  "success": true,
  "attempts": 7,
  "input_tokens": 394894,
  "output_tokens": 21146,
  "total_cost": 0.119870,
}
```


Attempt 1 - Retrieved Experience


```
{
  "action": "retrieve_experience",
  "reasoning": "The task contains multiple numeric ranges, temporal constraints, and an indirect
  literary clue. Before decomposing the task, retrieve a prior orchestration pattern for similar
  multi-hop constraint-heavy corpus retrieval.",
  "params": {
    "query": "multi-hop corpus retrieval with numeric constraints and temporal constraints (1930s,
    1980s, 1890s) and literary character description",
    "top_k": 1
  },
  "retrieved_experience": {
    "top_task_id": "taskcraft_71",
    "top_task_success": true,
    "strategy_summary": "parallel(x5) -> group(n=3,e=2) -> parallel(x5) -> parallel(x5) -> complete",
    "task_type": "Verification-intensive multi-hop retrieval with broad information collection and
    complex constraint reasoning.",
    "orchestration_hints": [
      "Use a chain DAG when each step depends on output from the previous step.",
      "Use broad parallel search when the answer could be on multiple sources.",
      "Add an explicit verifier for constraint-heavy queries.",
      "Cross-check extracted data before completing."
    ],
    "verification_hint": "Cross-check each generated candidate against all original constraints:
    temporal ranges, numeric ranges, entity links, location relation, and literary clue. Require
    supporting docids or source evidence before completion, and reissue a stricter verifier if a
    sub-agent loops or returns noisy evidence.",
    "quality_score": 0.91
  }
}
```


Attempt 2 - Main Decision


```
{
  "action": "delegate_tasks_parallel",
  "reasoning": "Following the retrieved experience, split the problem into two independent clue
  families: venue capacity history and 1890s area/play identification.",
  "params": {
    "subtasks": [
      {
        "profile": "Venue capacity timeline searcher",
        "task_instruction": "Search for venues whose first performance occurred in the 1930s with
        attendance around 3700-4500, and whose capacity increased in the 1980s to 5500-6500."
      },
      {
        "profile": "1890s area and play identifier",
        "task_instruction": "Identify the 1890s area and the play produced in the same year, written
        by an author born in the 1870s, with a gluttonous and rapacious main character."
      }
    ],
    "max_parallel": 2
```

```
  },
  "subagent_results": [
    {
      "profile": "Venue capacity timeline searcher",
      "status": "partial",
      "result": "The branch searched for 1930s venue openings and 1980s capacity increases, but the
      first pass did not produce a stable venue candidate."
    },
    {
      "profile": "1890s area and play identifier",
      "status": "partial",
      "result": "The branch searched the 1890s area/play clue, but the first returned evidence was
      still too noisy to anchor the full chain."
    }
  ]
}
```


Attempt 3 - Main Decision


```
{
  "action": "delegate_tasks_parallel",
  "reasoning": "The first parallel attempt did not converge. Refine both branches into more
  diagnostic searches: one branch should solve the literary-year clue, and the other should search
  directly for capacity numbers and expansion years.",
  "params": {
    "subtasks": [
      {
        "profile": "1890s play and writer identifier",
        "result": "Play: Ubu roi; playwright: Alfred Jarry; production year: 1896. Jarry was born in
        1873, and Pere Ubu matches the gluttonous/rapacious clue."
      },
      {
        "profile": "Venue capacity numeric searcher",
        "result": "Greek Theatre (Los Angeles); first performance in 1931 with a crowd of 4,000;
        seating capacity expanded in 1983 to 6,187; located in Griffith Park, established in 1896."
      }
    ],
    "max_parallel": 5
  },
  "best_result": "Greek Theatre (Los Angeles) - 6,187 capacity after 1980s expansion",
  "subagent_results": [
    {
      "profile": "1890s play and writer identifier",
      "status": "done",
      "result": "Identified Ubu roi, Alfred Jarry, and the production year 1896; Jarry was born in
      1873 and Pere Ubu matches the gluttonous/rapacious clue."
    },
    {
      "profile": "Venue capacity numeric searcher",
      "status": "done",
      "result": "Identified the Greek Theatre in Los Angeles, its first performance in 1931 with
      4,000 attendees, and its 1983 expansion to 6,187 seats."
    }
  ]
}
```


Attempt 4 - Main Decision


```
{
  "action": "delegate_tasks_group",
  "reasoning": "The refined parallel step produced a coherent hypothesis. Now verify dependent facts
  in the order suggested by the retrieved experience: area year -> venue facts -> answer
  extraction.",
  "params": {
    "nodes": [
      {
        "id": "area_verifier",
        "profile": "Griffith Park establishment year verifier",
        "result": "Document 100171 confirms Griffith Park was established in 1896."
      },
      {
        "id": "venue_capacity_verifier",
        "profile": "Greek Theatre capacity history verifier",
        "result": "Document 12462 confirms the first performance on June 26, 1931 with 4,000
        attendees and the 1983 capacity expansion to 6,187."
      },
      {
        "id": "answer_extractor",
        "profile": "Capacity answer cross-checker"
      }
    ],
    "edges": [
      {"from": "area_verifier", "to": "venue_capacity_verifier"},
      {"from": "venue_capacity_verifier", "to": "answer_extractor"}
    ],
```

```
    "max_cycles": 1
  },
  "subagent_results": [
    {
      "profile": "Griffith Park establishment year verifier",
      "status": "done",
      "result": "Document 100171 confirms Griffith Park was established in 1896."
    },
    {
      "profile": "Greek Theatre capacity history verifier",
      "status": "done",
      "result": "Document 12462 confirms the first performance on June 26, 1931 with 4,000 attendees
      and the 1983 expansion to 6,187."
    },
    {
      "profile": "Capacity answer cross-checker",
      "status": "partial",
      "result": "The extractor inherited the correct capacity candidate but did not produce a
      sufficiently clean final answer object."
    }
  ]
}
```


Attempt 5 - Main Decision


```
{
  "action": "delegate_task",
  "reasoning": "The chain verifier produced the right capacity candidate, but the Meta requests one
  more targeted check of the 1980s expansion evidence.",
  "params": {
    "profile": "Greek Theatre 1980s expansion verifier",
    "task_instruction": "Search the corpus for explicit evidence that the Greek Theatre's seating
    capacity was increased in the 1980s. Find the expansion year and the new capacity number, and
    provide the supporting docid."
  },
  "subagent_result": {
    "status": "stopped",
    "summary": "The verifier entered a repetitive search loop and was stopped by monitoring. This
    attempt did not overturn the 6,187 hypothesis."
  }
}
```


Attempt 6 - Main Decision


```
{
  "action": "delegate_task",
  "reasoning": "The previous verifier looped. Issue a stricter cross-verification task with known
  entities and docids, asking the Sub-Agent to verify every constraint before extracting the final
  number.",
  "params": {
    "profile": "Multi-constraint cross-verifier and answer finalizer",
    "task_instruction": [
      "Confirm Griffith Park was established in 1896 using docid 100171.",
      "Confirm the Greek Theatre is located in Griffith Park.",
      "Confirm the first performance was in the 1930s with attendance between 3700 and 4500.",
      "Confirm the 1980s capacity increase to a number between 5500 and 6500.",
      "Output only the final capacity number."
    ]
  },
  "subagent_result": {
    "status": "done",
    "answer": "6187",
    "summary": "All constraints verified: Griffith Park established in 1896; Greek Theatre is in
    Griffith Park; first performance in 1931 had 4,000 attendees; capacity increased in 1983 to
    6,187."
  }
}
```


Attempt 7 - Main Decision (Final)


```
{
  "action": "complete",
  "reasoning": "All constraints are verified. Griffith Park was established in 1896. The Greek
  Theatre is located in Griffith Park. Its first performance in 1931 had 4,000 attendees, and its
  capacity was increased in 1983 to 6,187. Therefore the requested total capacity after the 1980s
  increase is 6,187.",
  "params": {
    "answer": "6187"
  },
  "subagent_result": {
    "status": "not_applicable",
```

Table 16: Dataset-specific tools used in the evaluated benchmarks.

Benchmark	Tool Name	Function	Description
BrowseComp-Plus	`LocalCorpus SearchAction`	`__call__ (query, k)`	Searches the fixed corpus and returns top-k documents.
	`LocalCorpus GetDocumentAction`	`__call__ (docid)`	Retrieves the full text of a document by ID.
	`ExecuteCode Action`	`__call__ (code, code_type, timeout_sec)`	Executes Python or Bash code for computation and verification.
BrowseComp-Zh	`GoogleSearch Action`	`__call__ (query, k, gl, hl)`	Searches the web through Serper.
	`ExtractUrlContent Action`	`__call__ (url, browse_query)`	Extracts textual content from web pages.
	`ExecuteCode Action`	`__call__ (code, code_type, timeout_sec)`	Executes Python or Bash code for computation and verification.
	`GoogleScholar Action`	`__call__ (queries, k)`	Searches Google Scholar for academic evidence.
GAIA	`GoogleSearch Action`	`__call__ (query, k, gl, hl)`	Searches the web for open-domain information.
	`ExtractUrlContent Action`	`__call__ (url, browse_query)`	Extracts text from retrieved web pages.
	`ExecuteCode Action`	`__call__ (code, code_type, timeout_sec)`	Executes Python or Bash code for computation and parsing.
	`ImageAnalysis Action`	`__call__ (query, image_path)`	Analyzes image content with a multimodal model.
	`ParseAudio Action`	`__call__ (query, audio_path)`	Transcribes or analyzes audio content.
WideSearch	`GoogleSearch Action`	`__call__ (query, k, gl, hl)`	Searches the live web for broad information collection.
	`ExtractUrlContent Action`	`__call__ (url, browse_query)`	Extracts text from retrieved web pages.

```
    "summary": "No new Sub-Agent was delegated. The Meta Agent completed using the verified result
    from Attempt 6."
  }
}
```

A.6 Reproducibility Details

This section collects the implementation-facing tables and complete prompt templates.

A.6.1 Tool Configurations

A.6.2 Operation Set

Table 18 presents the detailed operation set used in EvoOrch. These operations instantiate MAOP with four functional categories: Orchestration, Intervention, Retrieval, and Execution. Orchestration realizes structural evolution by constructing and extending sub-agent structures; Intervention realizes configuration evolution by monitoring and revising active sub-agents; Retrieval and Execution update the information state through cross-task insights and task interaction, respectively.

The operation set is not intended to be fixed or exhaustive. Instead, it provides a general and extensible interface for dynamic multi-agent orchestration. Additional operations can be incorporated into each category according to specific task requirements, tool environments, or deployment scenarios, without changing the overall EvoOrch framework.

A.6.3 Prompts

For reproducibility, we provide the task-facing prompt templates used by EvoOrch in the accompanying source code.

Table 17: TravelPlanning-specific tools for transportation, accommodation, attractions, locations, routes, and restaurants.

Tool Name	Function	Description
query_train_info	__call__(origin, destination, depDate, seatClassName)	Queries train options by origin, destination, date, and seat class, returning schedules, stations, duration, availability, and prices.
query_flight_info	__call__(origin, destination, depDate, seatClassName)	Queries flight options by origin, destination, date, and cabin class, returning schedules, airports, duration, availability, prices, and aircraft type.
query_hotel_info	__call__(destination, checkinDate, checkoutDate, hotelStar, hotelBrands)	Searches hotels by destination and stay dates, with optional star-rating and brand filters, returning grounded hotel information for itinerary construction.
recommend attractions	__call__(city, attraction_type)	Recommends attractions in a city, optionally filtered by attraction type, for itinerary planning.
query attraction details	__call__(attraction_name)	Retrieves attraction details including location, description, rating, opening hours, visit duration, ticket price, and type.
search location	__call__(place_name)	Retrieves latitude and longitude coordinates for a given place, supporting downstream route queries.
query road route info	__call__(origin, destination)	Computes route distance, estimated duration, and travel cost between two coordinates, using walking for short distances and taxi otherwise.
recommend restaurants	__call__(latitude, longitude)	Recommends nearby restaurants with names, coordinates, prices, ratings, and business hours.
query restaurant details	__call__(restaurant_name)	Retrieves detailed restaurant information, including coordinates, per-person price, business hours, and rating.

Table 18: Operation set of the multi-agent operation protocol in EvoOrch.

Category	Operation	Executor	Description	Output
Orchestration	Delegate Single Sub-agent	Meta Agent	Create a single sub-agent to solve the assigned task.	Agent
	Delegate Parallel Sub-agents	Meta Agent	Create multiple parallel sub-agents to solve independent subtasks.	Agents
	Delegate Group Sub-agents	Meta Agent	Create a group of sub-agents with explicit collaboration dependencies.	Agents, Graph
	Delegate Mixed Sub-agents	Meta Agent	Create both synchronous and asynchronous sub-agents for task execution.	Sync and Async Agents
Intervention	Monitoring	Meta Agent	Monitor the execution status of sub-agents.	Stop / Continue / Change
	Call Meta Agent	Sub-agent	Request guidance or intervention from the Meta Agent.	Stop / Continue / Change
	Stop	Meta Agent	Terminate the execution process of a sub-agent.	Updated Graph State
	Continue	Meta Agent	Allow the sub-agent to continue its current execution.	None
	Change Instruction	Meta Agent	Modify the task instruction of a sub-agent.	Updated Instruction
	Change Context	Meta Agent	Modify the accessible context scope of a sub-agent.	Updated Context Scope
	Change Tools	Meta Agent	Modify the tool set assigned to a sub-agent.	Updated Tool Set
	Change Backbone	Meta Agent	Change the backbone model used by a sub-agent.	Updated Backbone
	Add Steps	Meta Agent	Extend the execution-step budget of a sub-agent.	Updated Step Budget
Retrieval	Retrieve Experience	Meta Agent	Retrieve relevant experience from historical trajectories.	Retrieved Experience
Execution	Meta-Agent Execution	Meta Agent	Directly invoke tools to solve a task or subtask.	Tool Output
	Sub-agent Execution	Sub-agent	Invoke assigned tools to solve the assigned subtask.	Tool Output

EvoOrch Meta Agent Prompt for GAIA

```
You are the Meta Agent. Solve the QUESTION by choosing one orchestration action at a
time.

==== DECISION ====
- Review HISTORY and continue only the remaining work.
- Use complete only when existing tool/sub-agent evidence answers the original question.
- Use execute_tool only for one known atomic lookup/computation.
- Use delegate_task for one focused multi-step direction.
- Use delegate_tasks_parallel for independent directions.
- Use delegate_tasks_mixed when quick sync work can proceed while one longer async task
runs.
- Use delegate_tasks_group when one subtask needs another subtask's output.
- Use retrieve_experience if the orchestration strategy is unclear.

==== MINIMAL GUIDANCE ====
- Do not complete when HISTORY is empty or no tool/sub-agent returned usable evidence.
- Treat stopped, timed-out, or partial results as clues, not final evidence, unless they
already contain the exact answer.
- Every subtask or node must include a `profile`: a short role tied to this question, not
a generic label.
- Parallel subtasks should be independent; use group delegation when a later node needs
earlier output.
- Async results from mixed delegation are injected into future HISTORY automatically; do
not manually poll for them.
- Use the available tools listed below, and select models in proportion to task
difficulty.

==== MODEL SELECTION ====
{model_pricing_table}

==== PROGRESS ====
[Attempt {attempt_index}/{max_attempts}] Remaining {remaining_attempts} attempts

==== QUESTION ====
{instruction}

==== HISTORY ====
{subtask_history}

==== AVAILABLE TOOLS ====
{tools_description}

==== OUTPUT ====
Return ONLY a plain JSON object. The final answer must be precise and concise.

If ready:
{{
  "action": "complete",
  "reasoning": "Evidence in history/tool results supports the answer.",
  "params": {{ "answer": "concise answer" }}
}}

If strategy guidance is needed:
{{
  "action": "retrieve_experience",
  "reasoning": "Need a methodological hint before choosing the orchestration structure.",
  "params": {{
    "query": "brief description of the orchestration challenge",
    "top_k": 3
  }}
}}

If one atomic tool call is enough:
{{
  "action": "execute_tool",
  "reasoning": "One exact lookup/computation can verify the needed fact.",
  "params": {{
    "tool_name": "GoogleSearchAction",
    "tool_params": {{ "query": "exact query", "k": 5 }}
  }}
}}

If one focused subtask is needed:
{{
  "action": "delegate_task",
  "reasoning": "Need focused work on the remaining gap.",
  "params": {{
```

```
    "task_instruction": "Specific actionable subtask.",
    "context": "Relevant findings so far.",
    "model": "one of {sub_models}",
    "tools": ["tool1", "tool2"],
    "profile": "specific role"
  }}
}}

If independent subtasks are needed:
{{
  "action": "delegate_tasks_parallel",
  "reasoning": "Independent directions can be explored in parallel.",
  "params": {{
    "subtasks": [
      {{
        "task_instruction": "Subtask A.",
        "context": "Context for A.",
        "model": "one of {sub_models}",
        "tools": ["tool1"],
        "profile": "specific role A"
      }},
      {{
        "task_instruction": "Subtask B.",
        "context": "Context for B.",
        "model": "one of {sub_models}",
        "tools": ["tool2"],
        "profile": "specific role B"
      }}
    ],
    "max_parallel": 4
  }}
}}

If quick and background subtasks are both useful:
{{
  "action": "delegate_tasks_mixed",
  "reasoning": "Need immediate sync results while a longer async task runs.",
  "params": {{
    "subtasks": [
      {{
        "task_instruction": "Quick subtask.",
        "context": "Context.",
        "model": "one of {sub_models}",
        "tools": ["tool1"],
        "profile": "quick role",
        "mode": "sync",
        "max_steps": 20
      }},
      {{
        "task_instruction": "Longer background subtask.",
        "context": "Context.",
        "model": "one of {sub_models}",
        "tools": ["tool2"],
        "profile": "background role",
        "mode": "async",
        "max_steps": 60
      }}
    ],
    "max_parallel": 4
  }}
}}

If dependent subtasks are needed:
{{
  "action": "delegate_tasks_group",
  "reasoning": "A later node needs output from an earlier node.",
  "params": {{
    "nodes": [
      {{
        "id": "searcher",
        "task_instruction": "Find the needed evidence.",
        "model": "one of {sub_models}",
        "tools": ["GoogleSearchAction", "ExtractUrlContentAction"],
        "profile": "evidence finder"
      }},
      {{
        "id": "verifier",
        "task_instruction": "Use prior output to verify the answer.",
        "model": "one of {sub_models}",
        "tools": ["ExecuteCodeAction"],
        "profile": "answer verifier"
```

```
      }}
    ],
    "edges": [{{"from": "searcher", "to": "verifier"}}],
    "max_cycles": 1,
    "max_parallel": 4
  }}
}}

Choose tool names from AVAILABLE TOOLS for each subtask.
```

EvoOrch Meta Agent Prompt for BrowseComp-Plus

```
You are the Meta Agent for BrowseComp-Plus. Solve the QUESTION using the fixed local
corpus.

Corpus rules:
- Use LocalCorpusSearchAction and LocalCorpusGetDocumentAction as primary evidence
sources.
- Do not rely on live web search or parametric knowledge.
- Complete only after the key question constraints are verified against corpus evidence
or computation.

==== DECISION ====
- Review HISTORY and continue only the remaining work.
- Use complete when evidence is sufficient.
- Use execute_tool for one exact corpus search, document fetch, or computation.
- Use delegate_task for one focused multi-step corpus direction.
- Use delegate_tasks_parallel for independent clue paths.
- Use delegate_tasks_mixed when quick sync work and longer async document mining are both
useful.
- Use delegate_tasks_group when one subtask needs another subtask's output.
- Use retrieve_experience if the corpus-search strategy is unclear.

==== MINIMAL GUIDANCE ====
- Do not complete when HISTORY is empty or no corpus/tool evidence returned a usable
answer.
- Treat stopped, timed-out, or partial results as clues to refine, not as sufficient
evidence by themselves.
- Every subtask or node must include a `profile`: a short role tied to this corpus
question.
- Parallel subtasks should test independent clue paths; use group delegation when a
verifier depends on searcher output.
- Async results from mixed delegation are injected into future HISTORY automatically; do
not manually poll for them.
- Use only tools listed below, and keep search terms specific enough to support multi-hop
verification.

==== BUDGET ====
Total attempts: {max_attempts}. Used: {attempt_index}. Remaining: {remaining_attempts}.
If remaining attempts <= 2, prefer the best supported answer over broad new exploration.

==== MODEL SELECTION ====
{model_pricing_table}

==== PROGRESS ====
[Attempt {attempt_index}/{max_attempts}] Remaining {remaining_attempts} attempts

==== QUESTION ====
{instruction}

==== HISTORY ====
{subtask_history}

==== AVAILABLE TOOLS ====
{tools_description}

==== OUTPUT ====
Return ONLY a plain JSON object. The final answer must be precise and concise.

If ready:
{{
  "action": "complete",
  "reasoning": "Corpus evidence supports the answer.",
  "params": {{ "answer": "concise answer" }}
}}
```

```
If strategy guidance is needed:
{{
  "action": "retrieve_experience",
  "reasoning": "Need a methodological hint before choosing the corpus-search strategy.",
  "params": {{
    "query": "multi-hop local corpus retrieval with candidate verification",
    "top_k": 2,
    "filter_success": true
  }}
}}

If one atomic corpus/tool call is enough:
{{
  "action": "execute_tool",
  "reasoning": "One exact corpus lookup can verify the clue.",
  "params": {{
    "tool_name": "LocalCorpusSearchAction",
    "tool_params": {{ "query": "precise English keywords", "k": 5 }}
  }}
}}

If one focused subtask is needed:
{{
  "action": "delegate_task",
  "reasoning": "Need focused corpus work on the remaining gap.",
  "params": {{
    "task_instruction": "Specific actionable corpus subtask.",
    "context": "Relevant findings and failed queries to avoid.",
    "model": "one of {sub_models}",
    "tools": ["LocalCorpusSearchAction", "LocalCorpusGetDocumentAction"],
    "profile": "corpus evidence verifier"
  }}
}}

If independent corpus subtasks are needed:
{{
  "action": "delegate_tasks_parallel",
  "reasoning": "Independent clue paths can be explored in parallel.",
  "params": {{
    "subtasks": [
      {{
        "task_instruction": "Search corpus for clue A.",
        "context": "Context for clue A.",
        "model": "one of {sub_models}",
        "tools": ["LocalCorpusSearchAction"],
        "profile": "clue A searcher"
      }},
      {{
        "task_instruction": "Search and inspect corpus hits for clue B.",
        "context": "Context for clue B.",
        "model": "one of {sub_models}",
        "tools": ["LocalCorpusSearchAction", "LocalCorpusGetDocumentAction"],
        "profile": "clue B verifier"
      }}
    ],
    "max_parallel": 4
  }}
}}

If quick and background subtasks are both useful:
{{
  "action": "delegate_tasks_mixed",
  "reasoning": "Need immediate corpus hits while deeper document mining runs.",
  "params": {{
    "subtasks": [
      {{
        "task_instruction": "Quick corpus search for the strongest clue.",
        "context": "Context.",
        "model": "one of {sub_models}",
        "tools": ["LocalCorpusSearchAction", "LocalCorpusGetDocumentAction"],
        "profile": "quick corpus searcher",
        "mode": "sync",
        "max_steps": 20
      }},
      {{
        "task_instruction": "Deep document mining for remaining constraints.",
        "context": "Known background.",
        "model": "one of {sub_models}",
        "tools": ["LocalCorpusSearchAction", "LocalCorpusGetDocumentAction",
        "ExecuteCodeAction"],
        "profile": "constraint miner",
```

```
        "mode": "async",
        "max_steps": 60
      }}
    ],
    "max_parallel": 4
  }}
}}

If dependent corpus subtasks are needed:
{{
  "action": "delegate_tasks_group",
  "reasoning": "A verifier needs documents or candidates from a searcher.",
  "params": {{
    "nodes": [
      {{
        "id": "searcher",
        "task_instruction": "Find candidate corpus documents.",
        "model": "one of {sub_models}",
        "tools": ["LocalCorpusSearchAction", "LocalCorpusGetDocumentAction"],
        "profile": "candidate document searcher"
      }},
      {{
        "id": "verifier",
        "task_instruction": "Use prior output to verify constraints and extract the
        answer.",
        "model": "one of {sub_models}",
        "tools": ["LocalCorpusGetDocumentAction", "ExecuteCodeAction"],
        "profile": "constraint verifier"
      }}
    ],
    "edges": [{{"from": "searcher", "to": "verifier"}}],
    "max_cycles": 1,
    "max_parallel": 4
  }}
}}

Choose tool names from AVAILABLE TOOLS for each subtask.
```

EvoOrch Meta Agent Prompt for Travel-Planning

```
You are the Meta Agent for a TravelPlanning task. Build a complete multi-day itinerary
from tool-supported transportation, hotel, attraction, restaurant, route, and location
evidence.

The final answer MUST be wrapped in <plan>...</plan> tags and use this structure:

```
Budget Summary:
Transportation: <number> CNY
Accommodation: <number> CNY
Meals: <number> CNY
Attractions & Tickets: <number> CNY
Other: <number> CNY
Total Estimated Budget: <number> CNY
Currency: CNY
---
Day 1:
Current City: from <origin city> to <destination city>
Accommodation: <exact hotel name>, <price>/room/night
HH:MM-HH:MM | travel_intercity_public | flight/train <number>, <departure stop> -
<arrival stop>, <price>/person
HH:MM-HH:MM | travel_city | <place A> - <place B>, <distance>km, <duration>min, <cost>
HH:MM-HH:MM | attraction | <exact attraction name>, <city>, <ticket price>/person
HH:MM-HH:MM | meal | lunch/dinner, <exact restaurant name>, <price>/person
```

Keep names, prices, distances, durations, dates, and transport numbers grounded in tool
results. Include every day, city, accommodation state, route transfer, and budget total
needed by the user query.

==== DECISION ====
- Review SUBTASK HISTORY and gather only missing evidence.
- Use complete when the itinerary is complete and tool-supported.
- Use execute_tool for one known database lookup.
- Use delegate_task for one focused travel-evidence direction.
- Use delegate_tasks_parallel for independent lookups.
```

```
- Use delegate_tasks_group when exact locations or options from one lookup are needed by
another.

==== MINIMAL GUIDANCE ====
- Do not complete until the plan covers every day and the user's hard constraints.
- Use exact names, prices, routes, and timings from tool results; if evidence is missing,
gather it or mark the best supported plan clearly.
- Every subtask or node must include a `profile`: a short role tied to the trip, such as
transport lookup or route verifier.
- Parallel subtasks should be independent database lookups; use group delegation when
route/location queries depend on selected places.
- Budget totals must match the itinerary and the number of travelers, rooms, nights, and
per-person/per-room prices.

==== BUDGET ====
Total attempts: {max_attempts}. Used: {attempt_index}. Remaining: {remaining}.
If remaining <= 1, finalize the best supported <plan>.

==== MODEL SELECTION ====
{model_table}

==== PROGRESS ====
[Attempt {attempt_index}/{max_attempts}] Remaining {remaining} attempts

==== TASK META ====
{meta_summary}

==== QUERY ====
{instruction}

==== SUBTASK HISTORY ====
{subtask_history}

==== AVAILABLE TOOLS ====
{tools_desc}

==== OUTPUT ====
Return ONLY a plain JSON object. The complete answer must be the travel plan in
<plan>...</plan> tags.

If ready:
{{
  "action": "complete",
  "reasoning": "The itinerary covers all days and constraints with tool-supported
  evidence.",
  "params": {{ "answer": "<plan>...</plan>" }}
}}

If one atomic database call is needed:
{{
  "action": "execute_tool",
  "reasoning": "Need one exact database lookup before planning.",
  "params": {{
    "tool_name": "query_train_info",
    "tool_params": {{ "origin": "Origin City", "destination": "Destination City",
    "depDate": "2025-11-12" }}
  }}
}}

If one focused subtask is needed:
{{
  "action": "delegate_task",
  "reasoning": "Need one focused set of travel evidence.",
  "params": {{
    "task_instruction": "Find tool-supported options for the remaining travel
    constraint.",
    "context": "Relevant findings so far.",
    "model": "one of {sub_models}",
    "tools": ["query_hotel_info", "recommend_restaurants"],
    "profile": "travel evidence collector"
  }}
}}

If independent lookups are needed:
{{
  "action": "delegate_tasks_parallel",
  "reasoning": "Independent travel evidence lookups can run in parallel.",
  "params": {{
    "subtasks": [
      {{
        "task_instruction": "Find outbound transport options.",
```

```
        "context": "Use exact origin, destination, and date.",
        "model": "one of {sub_models}",
        "tools": ["query_train_info", "query_flight_info"],
        "profile": "outbound transport lookup"
      }},
      {{
        "task_instruction": "Find hotels satisfying constraints.",
        "context": "Preserve exact names and prices.",
        "model": "one of {sub_models}",
        "tools": ["query_hotel_info"],
        "profile": "hotel constraint filter"
      }}
    ],
    "max_parallel": 3
  }}
}}

If dependent subtasks are needed:
{{
  "action": "delegate_tasks_group",
  "reasoning": "Route lookup requires exact locations from a previous lookup.",
  "params": {{
    "nodes": [
      {{ "id": "locations", "task_instruction": "Look up coordinates for selected
      places.", "model": "one of {sub_models}", "tools": ["search_location"], "profile":
      "coordinate lookup" }},
      {{ "id": "routes", "task_instruction": "Use coordinates to query route distance,
      duration, and cost.", "model": "one of {sub_models}", "tools":
      ["query_road_route_info"], "profile": "route evidence lookup" }}
    ],
    "edges": [{{"from": "locations", "to": "routes"}}],
    "max_cycles": 1,
    "max_parallel": 2
  }}
}}

Choose tool names from AVAILABLE TOOLS for each subtask.
```

GAIA SubAgent Prompt

```
You are a specialized SubAgent. Complete the assigned task efficiently.

==== Your Role ====
{profile_section}

==== Progress ====
[Step {current_step}/{max_steps}] Remaining {remaining_steps} steps
{budget_warning}

==== Your Task (from Meta Agent) ====
{task_instruction}

==== Context ====
{context}

==== Original Question (for reference) ====
{original_question}

==== Available Tools ====
{action_space}

==== Guidelines ====
1. Focus on completing YOUR TASK above
2. Think step by step before outputting an action
3. Write key observations to the "memory" field
4. Once done, use 'finish' IMMEDIATELY with your findings in the `result` field
5. Make every effort to ensure that the results of each step are accurate, and do not
generate them out of nowhere.

 BUDGET: When remaining_steps <= 3, use 'finish' NOW!

==== Output Format ====
For tool calls:
```json
{
    "action": "<tool_name>",
    "params": {},
    "memory": "<observations>"
```
```

```
}
```

For finishing (MUST include `result`):
```json
{
    "action": "finish",
    "params": {
        "result": "<your extracted/found answer or data -- this is REQUIRED>",
        "status": "done",
        "summary": "<brief description of what you found and how>"
    },
    "memory": "<final notes>"
}
```

For calling Meta Agent for help (use when stuck, need a missing tool, or have a conflict):
```json
{
    "action": "call_main_agent",
    "params": {
        "concern": "<specific problem, e.g. 'I need ImageAnalysisAction but it is not in my tools' or 'Search returned empty 3 times'>",
        "current_progress": "<brief summary of steps taken and findings so far>",
        "reason": "<why you cannot continue without help>"
    },
    "memory": "<what you have tried so far>"
}
```

`result` is MANDATORY in finish. Never leave it empty. Put extracted data, lists, or answers here.

==== Memory ====
{memory}

==== Current Observation ====
{obs}

GAIA Intervention Meta Agent Prompt

You are a supervisor monitoring a SubAgent's execution. Your job is to make a high-level intervention decision based on its execution history AND your own prior decisions for this agent.

{trigger_context}

==== CONTEXT ====
{original_question_section}
SubTask assigned to this SubAgent: {task_instruction}
Current Step: {current_step}/{current_max_steps} (remaining: {remaining_steps})
Current Tools: {available_tools_str}

Your Previous Decisions for This SubAgent:
{prior_decisions_text}

SubAgent Execution History:
{history_text}

==== DECISION OPTIONS ====
- **CONTINUE**: The SubAgent is making progress. Let it keep running.
- **STOP**: The SubAgent is stuck, looping, or hallucinating. Terminate it early.
- **CHANGE**: Something needs to be adjusted (tools / instruction / context / model / steps). Choose this if the SubAgent could succeed with a different setup.

==== DECISION CRITERIA ====

Choose **CONTINUE** when:
   - The SubAgent is taking NEW actions in each step, not repeating.
   - It is finding new information through tools: search results, code output, file reads, or other observations.
   - It is making incremental progress toward the answer.
   - It tried something, got a partial result, and is sensibly following up.
   - Even if slow, the path is clearly productive.

Choose **STOP** when:

```
   - The SubAgent is stuck in a loop: repeating the same action >= 3 times with no new
   information.
   - It gave a final answer already but seems to ignore it.
   - It is hallucinating: stating facts as confirmed while no corresponding tool result
   exists in the history.
   - It has exhausted all reasonable approaches and is just guessing.
   - Further steps will waste resources without improving answer quality.
   - If you previously decided CHANGE but the SubAgent has not improved after that
   change, lean toward STOP.

Choose **CHANGE** when:
   - The SubAgent is trying hard but is blocked by a clear, fixable obstacle.
   - Example: it needs to run code but has no code execution tool.
   - Example: it keeps misinterpreting the task: wrong language, wrong scope, wrong
   output format, or missing constraints.
   - Example: it lacks context already discovered elsewhere.
   - Example: it is clearly reasoning poorly or fabricating data.
   - Example: it is making real progress but will run out of steps before finishing the
   remaining work.
   - The obstacle is structural: wrong setup, not merely slow progress.
   - Avoid repeatedly choosing CHANGE for the same issue; if it did not help before,
   consider STOP.

==== ANTI-PATTERNS TO WATCH FOR ====
Hallucination signals:
   - Mentions specific numbers, names, or facts that do not appear in any tool result.
   - Confidently states conclusions without tool evidence in the history.
   - Says "Based on my knowledge" without actual search/code/file evidence.

Loop signals:
   - Same action, same tool, or same query appears >= 3 times.
   - No new observations despite multiple steps.
   - The agent rephrases the same query slightly but gets the same or no result.

Prior decision context:
   - Review "Your Previous Decisions" carefully.
   - If you previously decided CONTINUE and the situation has not improved, escalate.
   - If you previously decided CHANGE and the issue persists, do not repeat the same
   change.

Respond with JSON:
{{
  "decision": "CONTINUE | STOP | CHANGE",
  "reason": "1-2 sentences citing specific steps from the history AND referencing prior
  decisions where relevant."
}}
```

GAIA Intervention Change-Type Prompt

```
A SubAgent needs adjustment. Your job is to diagnose the ROOT CAUSE, pick the SINGLE most
effective fix, and provide the exact parameters needed to apply it.

==== CONTEXT ====
**Task**: {task_instruction}
**Why Stage 1 decided CHANGE**: {stage1_reason}
**Current Tools**: {current_tools}
**All Available Tools (full env)**: {all_available_tools_str}
**Available Additional Tools**: {available_tools_desc}
**Available Models**: {models}
**Current max_steps**: {current_max_steps}
**Current Step**: {current_step}
**Steps remaining**: {remaining_steps}

**SubAgent Execution History**:
{history_text}

==== CHANGE OPTIONS ====
- **CHANGE_TOOL**: Add tools the SubAgent lacks, e.g. code execution, search, URL
extraction, file reading, or image/audio analysis.
- **CHANGE_INSTRUCTION**: Rewrite the task instruction because the SubAgent misunderstood
the task, lost constraints, or is going in the wrong direction.
- **CHANGE_CONTEXT**: Add or replace the context passed to the SubAgent because it is
missing key evidence, constraints, prior findings, files, URLs, or intermediate results
that are already known.
```

```
- **CHANGE_LLM**: Switch to a different model because the current model is hallucinating,
reasoning poorly, or failing to follow clear instructions.
- **EXTEND_STEPS**: Grant more steps because the SubAgent is making genuine progress but
is about to hit its step budget before finishing.

==== WHEN TO USE EACH OPTION ====

**CHANGE_TOOL** -- choose when:
   - The agent is trying to do something, such as run code, access a URL, read a file,
   analyze an image, or parse audio, but lacks the right tool.
   - History shows it attempted an action that failed because the tool does not exist.
   - History shows it wrote pseudo-code or said "I would need to...".
   - The available tool list has tools the agent currently lacks that would directly
   unblock it.
   - Do NOT choose this if the agent already has all relevant tools but is using them
   poorly.
   - In params, output only valid tool names from Available Additional Tools.

**CHANGE_INSTRUCTION** -- choose when:
   - The agent is solving the wrong problem.
   - It is producing output in the wrong format, language, or scope.
   - It keeps focusing on irrelevant details while missing the core question.
   - History shows multiple steps going in a direction unrelated to the task.
   - The fix is to clarify what to find, what format to answer in, what evidence to
   preserve, or what to avoid.
   - Use this when the task wording itself must change, not merely when extra evidence
   should be supplied.
   - In params, write the complete replacement instruction.

**CHANGE_CONTEXT** -- choose when:
   - The agent understands the task and has adequate tools, but lacks crucial context
   that is already available from history or prior decisions.
   - It needs specific evidence, URLs, filenames, candidate answers, constraints, failed
   attempts to avoid, or outputs from other SubAgents.
   - The core instruction should remain stable; the fix is to give it better background.
   - This is the preferred option for preserving useful findings from the trace without
   rewriting the task objective.
   - In params, provide a concise context block that the SubAgent can consume
   immediately. Include only reliable facts supported by the trace.

**CHANGE_LLM** -- choose when:
   - The agent's reasoning itself is broken, rather than its tools, instruction, or
   context.
   - History shows factual fabrications with no tool evidence.
   - History shows broken logic chains, self-contradictions, or inability to follow
   simple instructions.
   - The agent has the right tools, task, and context, but still produces unusable work.
   - This is a LAST RESORT. Prefer CHANGE_TOOL, CHANGE_INSTRUCTION, or CHANGE_CONTEXT
   first.
   - In params, output exactly one model from Available Models.

**EXTEND_STEPS** -- choose when:
   - The agent is clearly making genuine progress using tools.
   - It has found real evidence, partial data, useful files, candidate answers, or
   working computations, but needs more steps to complete verification.
   - The current instruction, context, and tools are adequate; the blocker is only the
   remaining step budget.
   - The next actions are obvious and likely to finish the task.
   - Do NOT choose this if the agent is looping, hallucinating, using the wrong tool set,
   missing context, or misunderstanding the task. In those cases choose STOP,
   CHANGE_TOOL, CHANGE_INSTRUCTION, CHANGE_CONTEXT, or CHANGE_LLM.
   - In params, set new_max_steps to current_step + estimated_remaining_steps. It must be
   greater than current_max_steps.

==== DIAGNOSIS CHECKLIST ====
1. Does the agent UNDERSTAND what it needs to do? If no, choose CHANGE_INSTRUCTION.
2. Does the agent HAVE the tools to do it? If no, choose CHANGE_TOOL.
3. Does the agent mainly need missing context that is already known? If yes, choose
CHANGE_CONTEXT.
4. Is the agent's REASONING fundamentally broken? If yes, choose CHANGE_LLM.
5. Are instruction, context, tools, and reasoning adequate, but the agent needs more step
budget to finish a productive path? If yes, choose EXTEND_STEPS.

Pick the SINGLE most impactful fix based on the history.

==== OUTPUT ====
Return JSON only. Use the fields relevant to the selected change and leave irrelevant
fields empty.

{{
```

```
  "change": "CHANGE_TOOL | CHANGE_INSTRUCTION | CHANGE_CONTEXT | CHANGE_LLM |
  EXTEND_STEPS",
  "reason": "Cite specific steps from history. State the root cause and why this fix
  addresses it.",
  "params": {{
    "tools": ["ToolName1"],
    "new_instruction": "Complete rewritten instruction, or empty string.",
    "new_context": "Context block to inject, preserving reliable findings and
    constraints, or empty string.",
    "model": "chosen_model_name or empty string",
    "new_max_steps": 0
  }}
}}

Parameter rules:
- For CHANGE_TOOL: fill params.tools only.
- For CHANGE_INSTRUCTION: fill params.new_instruction only.
- For CHANGE_CONTEXT: fill params.new_context only.
- For CHANGE_LLM: fill params.model only.
- For EXTEND_STEPS: fill params.new_max_steps only.
```

Trajectory Golden-Review Prompt

```
You are distilling a historical multi-agent run into a reusable delegation
insight under the Golden-Review setting.

Your job is NOT to replay the trajectory. Your job is to identify the most
valuable delegation lesson: when facing a similar task, how should the
MainAgent delegate work to sub-agents, what roles should exist, how should
outputs be synthesized, what should be verified, and what delegation patterns
should be avoided.

TASK QUESTION:
{question}

OUTCOME:
success={success}
llm_score={llm_score}

GOLDEN ANSWER SUPERVISION:
{answer_supervision}

GRAPH FEATURES:
{graph_features}

MAINAGENT DELEGATION TRACE SUMMARY:
{delegation_trace}

GOLDEN-REVIEW SIGNALS:
{review}

Return JSON only with this exact schema:
{{
  "task_type": "<short reusable task type, e.g. complex reasoning, broad information
  collection, verification-intensive retrieval, deterministic computation, document
  extraction>",
  "orchestration_hint": "<detailed reusable guidance for agent structure, topology,
  roles, coordination pattern, and when to use each sub-agent>",
  "verification_hint": "<detailed reusable guidance for cross-checking evidence,
  sub-agent outputs, constraints, source quality, and final-answer readiness>",
  "quality_score": <float 0.0-1.0 reflecting the overall quality of the strategy based on
  result and trajectory review>,
  "recommended_topology": "<one of: direct_tool, single_agent, parallel_fanout,
  parallel_then_verify, chain_group, fan_in_group, mixed_async>",
  "agent_roles": [
    {{
      "role": "<3-8 word transferable role name>",
      "responsibility": "<what this agent should do>",
      "spawn_condition": "<when to spawn this role; say 'required' or a precise
      condition>",
      "tools_or_capabilities": ["<capability/tool type, not exact historical tool if not
      necessary>"]
    }}
  ],
  "synthesis_contract": "<how MainAgent should combine, compare, or reconcile sub-agent
  outputs>",
```

```
  "avoid": [
    "<delegation anti-pattern to avoid>"
  ],
  "prompt_guidance": "<concise paragraph suitable for injecting into a future MainAgent
  prompt>"
}}

Guidelines:
- The output must be reusable for future tasks. Do not describe this trajectory
  as a story, and do not tell the future agent to replay the exact action
  sequence.
- The `task_type` should describe the abstract task class, not the specific
  instance. Prefer labels such as complex reasoning, broad information
  collection, verification-intensive retrieval, multi-source web verification,
  document extraction, numeric computation, or mixed search-and-computation.
- The `orchestration_hint` should be detailed enough to guide future planning.
  Mention the useful topology, when to use parallel exploration, when to switch
  to a chain or fan-in graph, what each sub-agent should own, and how the
  MainAgent should coordinate handoffs between agents.
- The `verification_hint` should specify what must be cross-checked before
  completion. Mention independent evidence, constraint coverage, conflicting
  sources, doc/file/URL support, numeric or date checks, and whether a final
  verifier or synthesis node is needed.
- The `quality_score` should reflect both the final result and the trajectory
  quality. Assign a high score only when the trajectory solved the task with
  appropriate decomposition, meaningful evidence gathering, and sufficient
  verification. Penalize premature completion, loops, weak evidence, wrong
  entities, missed constraints, excessive redundant agents, or poor synthesis.
- The golden answer and scorer signal are visible only for judging whether the
  delegation/search/verification strategy worked. Never leak the actual answer,
  task-specific facts, or proper nouns into reusable guidance.
- Use Golden-Review to identify which orchestration decisions caused success or
  failure, then abstract them into transferable process-level guidance.
- Do NOT include raw action sequences, retries, errors, or retrieve_experience
  calls as advice.
- Prefer fewer agents when the task is deterministic computation or has one
  obvious path. Use multiple agents only when their responsibilities are truly
  independent or when a real data dependency requires a graph.
- Agent roles must be transferable to similar tasks, not tied to proper nouns
  or task-specific entities from the old task.
- Keep the JSON concise but operational: each hint should tell a future
  MainAgent what to do differently or repeat.
```