PROACTIVE AGENTIC ORCHESTRATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic orchestration delegates tasks to sub-agents with independent contexts, providing a key path to extending the effective execution horizon of large language model agents. However, existing orchestration methods face two limitations: in execution, current methods fix sub-agents' goals and harnesses at creation time, limiting their adaptability to dynamic real-world environments; in optimization, sparse and delayed feedback constrains effective learning of the orchestration policy. To address these limitations, we introduce PORCHESTRA, an on-the-fly proactive agentic orchestration framework. It allows sub-agents to proactively communicate with the orchestrator at any logical point to request harness adjustments while maintaining continuity of local execution. This mechanism not only enables sub-agents to adapt seamlessly to dynamic environments but also naturally inspires *On-Policy Orchestration Distillation* (OPOD) based on hindsight feedback. OPOD attributes sub-agents' proactive revision requests back to preceding orchestration decisions to derive targeted improvement signals, and leverages these signals as process supervision alongside the final reward to optimize the orchestration policy. We evaluate PORCHESTRA on three long-horizon agent benchmarks spanning deep research, dynamic digital assistance, and software engineering. PORCHESTRA improves task success by an average of 9.1% relative to the strongest baseline; combined with OPOD, it improves small-orchestrator success by an average of 10.0% relative to existing training baselines. PORCHESTRA also achieves a more favorable cost–performance trade-off. Our code is publicly available at https://anonymous.4open.science/r/POrchestra-23F3.

1 INTRODUCTION

To tackle complex, long-horizon tasks, humans form organizations to orchestrate individual efforts and achieve what no one could accomplish alone (March & Simon, 1993; Galbraith, 1974). Inspired by this, agentic orchestration (Ruan et al., 2026; Zhang et al., 2026; Wang et al., 2026) has emerged as a key means of enabling large language models (LLMs) (Qwen Team, 2026; Grattafiori et al., 2024) to tackle long-horizon tasks across deep research, digital assistance, and software engineering (Mialon et al., 2024; Froger et al., 2026; Jimenez et al., 2024). In these settings, the orchestrator delegates subtasks to sub-agents with execution harnesses (Wang et al., 2024) and independent contexts, thereby mitigating context rot (Du et al., 2025) and enabling longer-horizon execution.

For real-world long-horizon tasks, agentic orchestration has progressed from static design toward runtime adaptation. **(i) Manual configuration.** Early frameworks, such as MetaGPT and AgentVerse (Hong et al., 2024; Chen et al., 2024b), rely on humans to predefine agent roles and collaboration structures. **(ii) Offline orchestration.** Subsequent approaches use an orchestrator to automatically generate agent harnesses and collaboration workflows for each task (Zhang et al., 2025; Wang et al., 2025). However, interactions between agents and real-world environments are dynamic and error-prone (e.g., tool failures and shifts in task goals as the environment changes), and such unexpected disruptions can cause predefined harnesses to break down (*harness mismatch*) (Cemri et al., 2025). **(iii) Re-orchestration.** To enable runtime adaptation, recent work integrates harness orchestration into task execution. For example, AORCHESTRA (Ruan et al., 2026) re-creates subsequent sub-agents after preceding sub-agents finish execution, whereas MAS^2 (Wang et al., 2026) triggers re-orchestration when specific conditions are met, such as code execution errors.

Despite this progress, current re-orchestration methods face two limitations: ❶ *Delayed Adaptation in Execution:* harness mismatches often emerge naturally during sub-agent execution, yet the

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

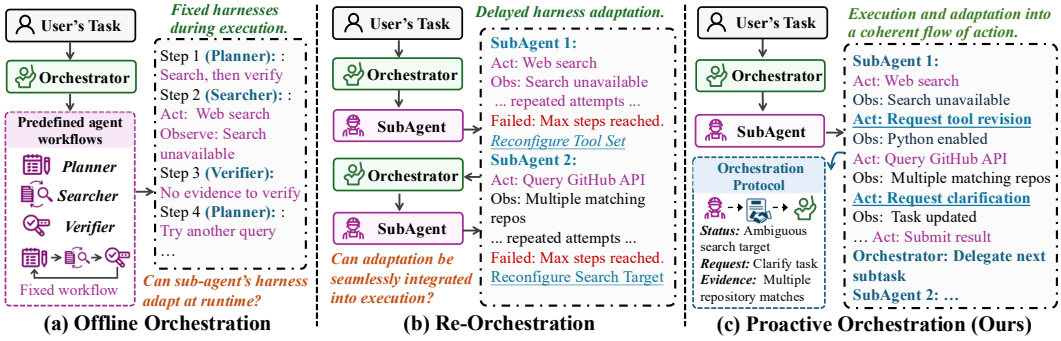


Figure 1: Comparison of orchestration paradigms. **(a) Offline orchestration** fixes harnesses and workflows. **(b) Re-orchestration** adapts harnesses upon completion or predefined triggers. **(c) PORCHESTRA** integrates execution and harness adaptation via proactive sub-agent communication.

orchestrator cannot readily detect or address them until execution ends or predefined conditions are triggered (Figure 1(b)). Consequently, sub-agents may continue operating under mismatched harnesses or pursuing outdated subgoals, wasting execution steps, incurring additional costs, and propagating local errors that disrupt the task. ❷ *Sparse Supervision in Optimization:* the reward used to optimize the orchestrator is typically available only after multiple rounds of orchestration and the sub-agent executions end. Even then, this reward is difficult to attribute to individual orchestration decisions and offers little guidance for improving them. In light of these limitations, a central question arises: Can we **seamlessly integrate dynamic adjustment** of sub-agent harnesses into execution while providing **denser, richer supervision** for orchestrator policy optimization?

To answer this question, we propose PORCHESTRA, a proactive agentic orchestration paradigm. Our motivation is that, as task executors, sub-agents can recognize harness mismatches from local execution difficulties and request revisions. Accordingly, PORCHESTRA makes harness adjustment a collaborative process between sub-agents and the orchestrator. A sub-agent can interact with the environment or proactively pause execution at any logical point to send structured feedback containing revision suggestions and supporting evidence from its local execution perspective. The orchestrator updates the harness from a global task perspective. The update is recorded as an observation in the sub-agent's local trajectory, forming a continuous action stream interleaving harness adaptation with task execution. In this way, the orchestrator is no longer limited to creating sub-agents with fixed harnesses; it can revise their harnesses throughout their lifecycles based on feedback.

Furthermore, proactive sub-agent feedback reveals obstacles encountered during execution and provides corresponding suggestions for improvement. This feedback mechanism not only enables sub-agents to adapt seamlessly to dynamic environments during execution but also naturally motivates *On-Policy Orchestration Distillation* (OPOD), which optimizes the orchestrator using hindsight feedback. OPOD first attributes execution feedback to prior decisions and distills it into natural-language improvement gradients. It then appends these gradients to the original decision context and uses the same model to construct a feedback-augmented self-teacher, providing process-level supervision for orchestration decisions through self-distillation. Meanwhile, OPOD retains task-level outcome supervision via reward-weighted regression, combining sparse final rewards with targeted process feedback to optimize the orchestration policy. Our contributions are as follows:

❶ **Execution Perspective:** We introduce PORCHESTRA, an agentic orchestration paradigm that enables sub-agents to proactively participate in their own orchestration, allowing their harnesses to adapt seamlessly as dynamic, long-horizon tasks unfold in real-world environments.

❷ **Optimization Perspective:** We introduce OPOD, an on-policy orchestration self-distillation method that converts proactive feedback from sub-agents in PORCHESTRA into process-level supervision, enabling more targeted optimization of the orchestrator policy.

❸ **Empirical Evaluation:** We evaluate PORCHESTRA on three long-horizon benchmarks spanning deep research, dynamic digital assistance, and software engineering. Results demonstrate **(i) superior orchestration**, improving task success by an average of 9.1% relative to the strongest baseline; **(ii) enhanced learnability**, with PORCHESTRA and OPOD improving small-orchestrator success by an average of 10.0% relative to training baselines; and **(iii) Pareto-optimal efficiency**, achieving the highest performance along the cost–performance frontier.

2 Related Work

Agentic Orchestration Systems. Agentic orchestration decomposes complex tasks by creating sub-agents and coordinating their work. In early **manually configured agent systems**, such as AutoGen, MetaGPT, and AgentVerse, developers predefine sub-agent composition and collaboration (Wu et al., 2024; Hong et al., 2024; Chen et al., 2024b); **offline orchestration** automatically generates agent systems before execution: G-Designer learns communication topologies, while ScoreFlow searches workflows using task-level feedback (Zhang et al., 2025; Wang et al., 2025). Both paradigms fix agent configurations before task execution. **Re-orchestration** further enables runtime adaptation: MAS^2 revises generated workflows when specific errors occur (Wang et al., 2026); methods such as Orchestra-o1 and AOrchestra use sub-agents as the unit of orchestration, re-creating subsequent sub-agents after preceding sub-agents finish execution (Zhang et al., 2026; Ruan et al., 2026). This sub-agent-based paradigm has demonstrated practical value in real-world applications (OpenAI, 2026; Anthropic, 2026). In POrchestra, we follow the same paradigm but introduce proactive sub-agents to seamlessly integrate orchestration into their execution.

Orchestration Policy Optimization. Learning from execution experience can improve the orchestrator's delegation decisions. Existing approaches fall into two categories: **Learning from sparse supervision.** AOrchestra (Ruan et al., 2026) adopts supervised fine-tuning (SFT), while MAS-Orchestra (Ke et al., 2026) applies reinforcement learning with sparse and delayed task-level rewards. **Process-level reward shaping.** Uno-Orchestra (Cui et al., 2026) introduces format-based rewards at each step, whereas DA-GRPO (Zhang et al., 2026) uses LLM-based rubrics to score candidate decisions. However, local rewards primarily assess output compliance or decision quality, and therefore provide limited decision attribution or directional revision signals that are grounded in the actual obstacles encountered during execution. By contrast, OPOD turns proactive sub-agent feedback into execution-grounded attribution and revision guidance for prior orchestration decisions.

3 Methodology

We first formulate the existing re-orchestration process (Section 3.1), then introduce POrchestra (Figure 2), which interleaves sub-agent execution and harness adaptation via proactive communication (Section 3.2). Finally, we use this sub-agent feedback as process supervision to optimize the orchestrator with OPOD via on-policy self-distillation (Section 3.3).

3.1 Problem Formulation

In sub-agent-based re-orchestration, task execution alternates between global control by the orchestrator and local execution by sub-agents. Specifically, given a user goal G and an orchestration horizon T_{orch}, at round $0 \le t < T_{\text{orch}}$, the orchestrator either creates a sub-agent to execute a delegated task or submits the final answer:

$$a_t^{\text{orch}} \sim \pi_\theta(\cdot \mid G, s_t^{\text{orch}}), \qquad a_t^{\text{orch}} \in \{\texttt{Delegate}(\Phi_t), \texttt{Finish}(y)\}. \tag{1}$$

π_θ denotes the orchestration policy parameterized by θ. For delegation, the orchestrator generates a harness $\Phi_t = (g_t, C_t, \Omega_t, B_t)$ for a new sub-agent, specifying the delegated subgoal, curated working context, assigned tool set, and execution budget. y denotes the final answer, and the orchestration state s_t^{orch} summarizes the orchestrator's decision history and the local results returned by completed sub-agents. Each Φ_t instantiates a sub-agent with action space $\mathcal{A}_t^{\text{exe}} = \Omega_t \cup \{\texttt{Submit}\}$. Tools in Ω_t support environment interaction, whereas `Submit` ends the sub-agent's lifecycle and returns a result o_t^{orch} to the orchestrator. This repeats until the orchestrator terminates at round t^*:

$$\tau^{\text{orch}} = (s_0^{\text{orch}}, \texttt{Delegate}(\Phi_0), o_0^{\text{orch}}, \ldots, s_{t^*}^{\text{orch}}, \texttt{Finish}(y), R_\tau). \tag{2}$$

Here, R_τ is the terminal reward assigned by the task evaluator. This formulation reveals two limitations: **(1) During execution,** existing re-orchestration methods typically keep each sub-agent's harness fixed throughout its lifetime. Adapting the harness requires waiting for the current sub-agent to invoke `Submit` or time out before creating a new one. This not only delays adaptation but also causes the loss of the sub-agent's ongoing execution state. **(2) During optimization,** sub-agent feedback o_t^{orch} becomes available only at the end of each sub-agent lifecycle, often after dozens of execution steps, whereas the terminal reward R_τ is delayed until the entire task concludes after multiple such lifecycles. Consequently, supervision for orchestration decisions is sparse and delayed.

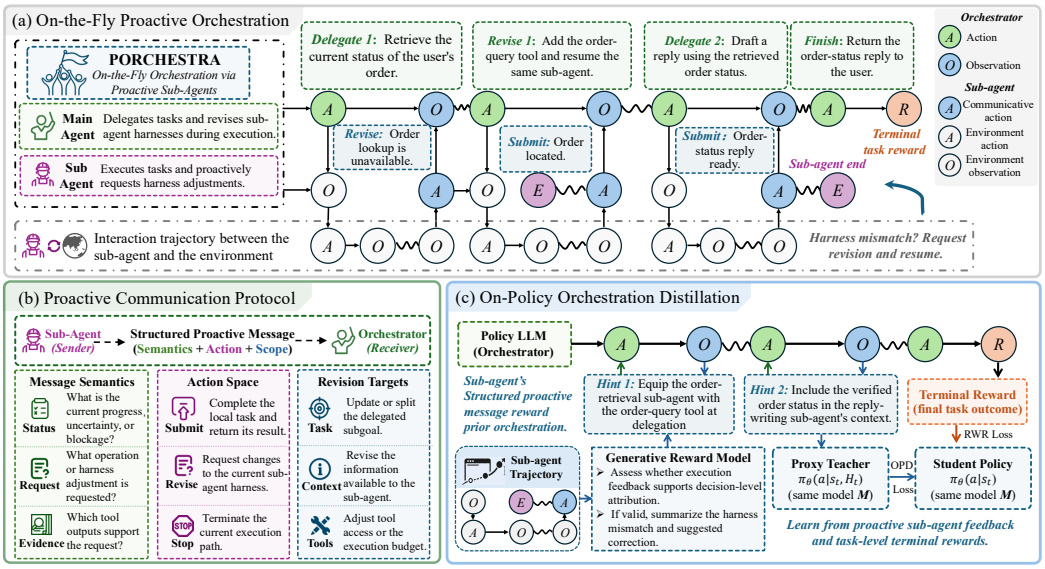


Figure 2: Overview of PORCHESTRA and OPOD. **(a)** Sub-agent execution and harness adaptation form a coherent workflow. **(b)** Structured proactive communication guides harness revisions. **(c)** OPOD combines feedback-based self-distillation with terminal rewards to optimize the orchestrator.

3.2 ON-THE-FLY AGENTIC ORCHESTRATION VIA PROACTIVE SUB-AGENTS

To address delayed adaptation in agentic orchestration, PORCHESTRA gives sub-agents control over initiating orchestration adjustments during execution. A sub-agent can pause at any point and request harness revisions from the orchestrator. We formalize this process through three components: *(i) Sub-Agent Execution*, *(ii) Proactive Communication Protocol*, and *(iii) Orchestrator Coordination*.

Sub-Agent Execution. Let M_{sub} denote the sub-agent model. For a sub-agent instantiated by Φ_i, at local step $\ell \in \{0, \dots, B_i - 1\}$, the prompt template I_{sub} combines the subgoal g_i, orchestrator-assigned context C_i, tool set Ω_i, internal memory $h_{i,\ell}$ consisting of the accumulated local trajectory, and communication action set $\mathcal{A}_i^{\text{comm}}$ into a sub-agent prompt. The prompt guides M_{sub} to consider proactive communication as a planning option when it encounters obstacles that cannot be resolved through local actions, such as task ambiguity or unavailable essential tools:

$$a_{i,\ell}^{\text{sub}} = M_{\text{sub}}(I_{\text{sub}}(g_i, C_i, \Omega_i, h_{i,\ell}, \mathcal{A}_i^{\text{comm}})), \qquad a_{i,\ell}^{\text{sub}} \in \mathcal{A}_i^{\text{sub}} = \Omega_i \cup \mathcal{A}_i^{\text{comm}}. \tag{3}$$

When M_{sub} invokes a tool from Ω_i, the environment returns an observation, which is appended to the sub-agent's local trajectory. When M_{sub} selects a communication action, execution pauses and control passes to the orchestrator. The orchestrator can either revise the current sub-agent's harness or terminate it. In the former case, the sub-agent's harness is updated to $\Phi_i^{\ell+1}$. The update record is added to the sub-agent's internal memory as a communication observation. This mechanism allows the sub-agent to recognize harness updates alongside its prior execution history and continue task execution coherently. In the latter case, the sub-agent's lifecycle ends immediately.

Proactive Communication Protocol. Proactive sub-agent communication provides the orchestrator with an intervention point, but runtime adaptation also requires sufficient execution information. Inspired by organizational information-processing theory (Galbraith, 1974), we instantiate $\mathcal{A}_i^{\text{comm}}$ as a standardized proactive orchestration communication protocol. Each communication action carries a structured message whose semantics are organized into three complementary signals:

- *Status signal* summarizes current progress, uncertainty, and blockage from the sub-agent's perspective, helping the orchestrator understand why communication was initiated.
- *Request signal* specifies the adjustment target and intended operation.
- *Evidence signal* preserves key tool outputs that substantiate the request.

$\mathcal{A}_i^{\text{comm}}$ supports three control actions: `submit` completes the local task and returns its result; `revise` requests a modification to the sub-agent configuration; and `stop` terminates the execution

path. When the selected action is `revise`, the request may specify changes along two complementary dimensions. *Task Scope* adjusts the subgoal g_i, for example by splitting or updating the task. *Resource* adjusts the working context C_i, tool set Ω_i, or execution budget B_i.

Orchestrator Coordination. From the orchestrator's perspective, PORCHESTRA extends re-orchestration from decisions made only after a sub-agent returns to continuous coordination throughout its lifecycle. For task G, the orchestrator begins with $\texttt{Delegate}(\Phi_i^0)$. When structured feedback arrives, the orchestrator updates the global orchestration state s_t^{orch} and selects $a_t^{\text{orch}} \in \{\texttt{Delegate}(\Phi_j^0), \texttt{Revise}(\Phi_i^{\ell+1}), \texttt{Stop}(i), \texttt{Finish}(y)\}$. Here, superscripts distinguish configuration versions within one delegation. PORCHESTRA extends the orchestration trajectory into an iterative process in which proactive sub-agent feedback and orchestration decisions interleave:

$$\tau_{\text{PORCH}}^{\text{orch}} = \langle s_0^{\text{orch}}, \texttt{Delegate}(\Phi_0^0), o_0^{\text{orch}}, s_1^{\text{orch}}, \texttt{Revise}(\Phi_0^1), o_1^{\text{orch}}, \ldots, s_{t^*}^{\text{orch}}, \texttt{Finish}(y), R_\tau \rangle. \tag{4}$$

Consequently, between $\texttt{Delegate}(\Phi_i)$ and the sub-agent's return, PORCHESTRA uses more timely and informative proactive feedback to adjust the subgoal and harness in Φ_i as tasks progress and environmental needs change. In contrast, existing sub-agent-based re-orchestration methods adjust only after sub-agent completion, creating tension between coherent execution and timely adaptation:

Proposition 1 (Re-Orchestration After Sub-Agent Completion Incurs Adaptation Delay)**.** *Suppose a sub-agent requires at least $H_{\min}$ steps to maintain coherent execution, while re-orchestration is permitted only after its lifecycle ends. If a mismatch occurs uniformly over horizon H, then the probability of intervention within a recovery window δ satisfies* $\Pr(D_H \le \delta) \le \min\left\{1, \frac{\delta}{H_{\min}}\right\}$.

When $H_{\min} \gg \delta$, this bound scales inversely with $H_{\min}$: the longer the horizon required for coherence, the lower the upper bound on timely intervention. PORCHESTRA allows sub-agents to trigger adjustments during execution, so intervention need not wait until the H-step execution ends. We summarize this interleaved inference process in Appendix A (Algorithm 1).

3.3 ON-POLICY ORCHESTRATION DISTILLATION

Compared with eliciting orchestration through prompting, optimizing the orchestrator from execution experience offers an attractive direction. However, supervision for orchestration decisions is sparse and expensive to obtain, making it essential to provide more timely and informative feedback and convert it into effective supervision. PORCHESTRA's interleaved process provides a natural basis for this objective: proactive sub-agent feedback not only guides runtime adaptation but can also reveal how preceding orchestration decisions could be improved. Motivated by this, we introduce On-Policy Orchestration Distillation (OPOD), which converts sub-agent feedback into natural-language gradients, constructs a feedback-augmented self-teacher using the same model, and distills its decision distribution into the orchestration policy on trajectories generated by the current policy.

Natural-Language Gradient Generation. To turn structured feedback o_t^{orch} into supervision for improving prior orchestration decisions, we first assess whether the issues it reveals can be attributed to the original orchestration choices. For example, subgoal ambiguity exposed during execution may reflect unclear task delegation and guide its improvement; by contrast, if the user changes task requirements during sub-agent execution, the resulting need to revise the subgoal should not be attributed to that decision. When sufficient evidence supports attribution, we extract cues for harness improvement from proactive sub-agent feedback and distill them into a natural-language gradient, providing concrete revision guidance for prior orchestration decisions. Formally, whenever o_t^{orch} is returned, we use the same backbone model M as the trainable orchestrator to perform attribution under a fixed generative reward-modeling instruction I_{GRM}. Its inputs are the corresponding orchestration state and decision, the intervening sub-agent execution segment, and the returned feedback:

$$\{\texttt{FLAG}_t, \mathcal{H}_t\} = M(I_{\text{GRM}}(s_t^{\text{orch}}, a_t^{\text{orch}}, \langle h_k^{\text{sub}}, a_k^{\text{sub}}, o_k^{\text{env}} \rangle_{k=1}^{K_t}, o_t^{\text{orch}})). \tag{5}$$

Here, $\langle h_k^{\text{sub}}, a_k^{\text{sub}}, o_k^{\text{env}} \rangle_{k=1}^{K_t}$ denotes the sub-agent execution segment after a_t^{orch} takes effect and before o_t^{orch} is returned, where k indexes execution steps and K_t is the segment length. The flag $\texttt{FLAG}_t \in \{\texttt{true}, \texttt{false}\}$ indicates whether sufficient evidence supports attribution to the orchestration decision. When $\texttt{FLAG}_t = \texttt{true}$, $\mathcal{H}_t$ summarizes the harness mismatch and suggested correction; otherwise, $\mathcal{H}_t$ is empty and the attribution unit is excluded from distillation.

Self-Distillation Policy Optimization. To use $\mathcal{H}_t$ as policy-level supervision, we append it to the original orchestration context and use the same model to construct a feedback-augmented teacher policy. The original orchestration policy and teacher policy are defined as

$$\pi_\theta(a \mid s_t^{\text{orch}}) = M(a \mid I_{\text{orch}}(s_t^{\text{orch}})), \qquad \pi_\theta^+(a \mid s_t^{\text{orch}}, \mathcal{H}_t) = M(a \mid I_{\text{orch}}(s_t^{\text{orch}}) \oplus \mathcal{H}_t), \tag{6}$$

where $\oplus$ denotes context concatenation. The resulting feedback-augmented policy can be viewed as an in-context policy-improvement step: $\mathcal{H}_t$ changes the orchestration decision distribution without updating model parameters. Because π_θ^+ uses the same backbone model and differs only in its augmented context, it serves as a self-teacher without requiring an external model. For a complete trajectory τ, let $\mathcal{T}(\tau) = \{t \mid (s_t^{\text{orch}}, a_t^{\text{orch}}) \in \tau\}$ denote its orchestration decision indices. We define the on-policy distillation (OPD) loss (Agarwal et al., 2024) using full-vocabulary reverse KL:

$$\ell_{\text{OPD}}(\tau) = \textstyle\sum_{t \in \mathcal{T}(\tau)} \mathbb{1}[\texttt{FLAG}_t = \texttt{true}] D_{\text{KL}}(\pi_\theta(\cdot \mid s_t^{\text{orch}}) \,\|\, \text{sg}[\pi_\theta^+(\cdot \mid s_t^{\text{orch}}, \mathcal{H}_t)]). \tag{7}$$

where t indexes orchestration decisions and $\text{sg}[\cdot]$ stops gradients through the self-teacher. This loss aggregates guidance from all attribution units with valid gradients. We retain outcome supervision via reward-weighted regression (RWR) (Peters & Schaal, 2007), with trajectory loss:

$$\ell_{\text{RWR}}(\tau) = -R_\tau \textstyle\sum_{t \in \mathcal{T}(\tau)} \log \pi_\theta(a_t^{\text{orch}} \mid s_t^{\text{orch}}). \tag{8}$$

Finally, OPOD combines process-level OPD and trajectory-level RWR supervision: $\mathcal{L}_{\text{OPOD}}(\tau) = \ell_{\text{RWR}}(\tau) + \lambda \ell_{\text{OPD}}(\tau)$. In implementation, we initialize the orchestration policy through supervised fine-tuning on expert trajectories, then optimize it with OPOD on on-policy rollouts (Algorithm 2).

Proposition 2 (OPOD Reduces Long-Horizon Attribution Error)**.** *Under bounded attribution noise, terminal rewards propagate through an attribution horizon $O(H_{\text{orch}})$ times longer than that of OPD feedback, thereby accumulating noise over a longer execution chain. Furthermore, when the bias of OPD feedback is bounded, there exists a weight $\lambda > 0$ such that OPOD yields lower expected error relative to ideal process-level supervision than RWR alone (see Appendix D for both proofs).*

4 Experiments

We evaluate our framework and optimization method through four research questions:

- **RQ1:** Does proactive orchestration improve task success across benchmarks and backbone models? We compare POrchestra with single-agent and orchestration baselines (Section 4.2).
- **RQ2:** Does OPOD improve orchestrator optimization? We compare training methods and examine the contributions of process feedback and outcome supervision (Section 4.3).
- **RQ3:** How do POrchestra and OPOD affect cost–performance trade-offs? We analyze inference costs and the efficiency of trained small orchestrators (Section 4.4).
- **RQ4:** When and why does proactive orchestration help? We examine task length and dynamicity, and compare proactive feedback with predefined triggers and periodic polling (Section 4.5).

4.1 Experimental Setup

Benchmarks. We evaluate POrchestra on three long-horizon agent benchmarks with distinct interaction settings. **GAIA** (Mialon et al., 2024) evaluates the ability of deep-research assistants to complete tasks by using real tools and accessing real web pages; its interactions are affected by environmental changes and tool stability. **GAIA2** (Froger et al., 2026) is a challenging read–write interaction benchmark that emphasizes agents' adaptability in dynamic and asynchronous environments, such as adjusting task goals and subsequent actions based on replies received after sending an email. **SWE-Bench Verified** (Jimenez et al., 2024) evaluates software-engineering agents by requiring them to resolve real-world GitHub issues. We use task success rate as the evaluation metric.

Model & Baselines. For inference-time evaluation, we compare POrchestra with three categories of baselines: **(1) single-agent baselines,** including ReAct (Yao et al., 2023) and Mini-SWE (Yang et al., 2024); **(2) offline orchestration systems,** including Skill-MAS (Lin et al., 2026) and AutoAgents (Chen et al., 2024a); and **(3) re-orchestration systems,** including MAS^2 (Wang et al., 2026), AOrchestra (Ruan et al., 2026), and Orchestra-o1 (Zhang et al., 2026). Following

Table 1: Performance with three backbones across three dynamic, long-horizon environments: GAIA, GAIA2, and SWE-Bench. Baselines include single-agent systems (ReAct, Mini-SWE), offline orchestration (Skill-MAS, AutoAgents), and re-orchestration (MAS^2, Orchestra-o1, AORCHESTRA). Best results are **bold**, and second-best results are underlined.

Method	General AI Assistant				General AI Assistant 2					SWE-Bench
	Level 1	Level 2	Level 3	Average	Execution	Search	Adaptability	Time	Average	
DeepSeek-V3.2										
ReAct	79.2	58.1	50.0	63.6	34.4	56.3	21.9	21.9	33.6	62.0
Mini-SWE	64.2	64.0	26.9	58.2	25.0	50.0	18.8	12.5	26.6	70.0
Skill-MAS	67.9	58.1	30.8	57.0	21.9	43.8	0.0	0.0	16.4	60.0
AutoAgents	64.2	40.7	34.6	47.3	15.6	59.4	0.0	6.3	20.3	30.0
MAS^2	47.2	39.5	19.2	38.8	25.0	56.3	12.5	3.1	24.2	62.0
Orchestra-o1	**81.1**	61.6	30.8	63.0	**43.8**	59.4	25.0	9.4	34.4	66.0
AORCHESTRA	73.6	55.8	46.2	60.0	**43.8**	75.0	25.0	15.6	39.8	65.0
PORCHESTRA	79.2	**72.1**	**53.8**	**71.5**	**43.8**	**78.1**	**43.8**	**25.0**	**47.7**	**71.0**
Gemini-3-Flash										
ReAct	84.9	66.3	46.2	69.1	50.0	84.4	28.1	25.0	46.9	53.0
Mini-SWE	81.1	**76.7**	57.7	75.2	43.8	25.0	25.0	25.0	29.7	75.0
Skill-MAS	81.1	61.6	46.2	65.5	28.1	62.5	9.4	6.3	26.6	68.0
AutoAgents	69.8	51.2	53.8	57.6	25.0	46.9	0.0	3.1	18.8	67.0
Orchestra-o1	**86.8**	67.4	57.7	72.1	50.0	84.4	25.0	15.6	43.8	69.0
MAS^2	77.4	57.0	42.3	61.2	**56.3**	75.0	21.9	0.0	38.6	59.0
AORCHESTRA	**86.8**	74.4	61.5	76.4	46.9	75.0	18.8	18.8	39.8	76.0
PORCHESTRA	**86.8**	**76.7**	**65.4**	**78.2**	**56.3**	**90.6**	**31.3**	**37.5**	**53.9**	**77.0**
DeepSeek-V4-Flash										
ReAct	79.2	72.1	**61.5**	72.7	37.5	75.0	37.5	**28.1**	44.5	71.0
Mini-SWE	76.9	67.4	53.8	68.3	40.6	78.1	25.0	18.8	40.6	73.0
Skill-MAS	73.6	59.3	46.2	61.8	28.1	65.6	0.0	3.1	24.2	76.0
AutoAgents	86.8	62.8	46.2	67.9	21.9	53.1	0.0	0.0	18.8	69.0
Orchestra-o1	83.0	57.0	38.5	62.4	50.0	77.4	25.0	25.0	43.8	75.0
MAS^2	69.8	61.6	46.2	61.8	31.3	75.0	3.1	0.0	26.6	78.0
AORCHESTRA	86.8	**75.6**	50.0	75.2	53.1	81.3	24.1	15.6	43.0	71.0
PORCHESTRA	**88.7**	**75.6**	**61.5**	**77.6**	**59.4**	**87.5**	**46.9**	**28.1**	**55.5**	**79.0**

the AORCHESTRA setup, orchestrated systems use `Gemini-3-Flash` (Google DeepMind, 2025) as the orchestrator. All systems are evaluated with `Gemini-3-Flash`, `DeepSeek-V3.2` (Liu et al., 2025), and `DeepSeek-V4-Flash` (Xu et al., 2026) as execution backbones. For training-time evaluation, we train `Qwen3.5-9B` (Qwen Team, 2026) orchestrators across Orchestra-o1, AORCHESTRA, and PORCHESTRA. Methods include SFT, RWR, DA-GRPO (Zhang et al., 2026), and OPOD; each is applied to frameworks compatible with its feedback and execution interface. See Appendices E and F for implementation details and prompts.

4.2 EFFECTIVENESS OF PROACTIVE ORCHESTRATION IN EXECUTION (RQ1)

Obs. ❶: Existing orchestration methods struggle to adapt reliably to dynamic environments. Table 1 shows that offline orchestration (Skill-MAS and AutoAgents) substantially trails single-agent baselines in average GAIA2 success across all three backbones. For Adaptability tasks involving changing task requirements, AutoAgents scores 0% with all three backbones. Current re-orchestration baselines improve some results but remain mixed against single-agent baselines: with `Gemini-3-Flash`, AORCHESTRA reaches 76.4% on GAIA, above ReAct's 69.1%, but 39.8% on GAIA2, below ReAct's 46.9%. These results suggest static harnesses or delayed harness updates represent important bottlenecks to consistent gains from orchestration in dynamic environments.

Obs. ❷: Proactive orchestration outperforms existing methods. With all three sub-agent backbones, PORCHESTRA achieves the highest average success rates on GAIA and GAIA2, as well as the highest success rate on SWE-Bench. In particular, on the more dynamic GAIA2 benchmark, it reaches 55.5% with `DeepSeek-V4-Flash`, a 24.6% relative improvement over the strongest baseline, ReAct, at 44.5%. Notably, PORCHESTRA's sub-agents use a ReAct-style execution mech-

Table 2: Results and ablation of OPOD across orchestration harnesses and algorithms.

Orchestration	Optimization	General AI Assistant				General AI Assistant 2				
		Level 1	Level 2	Level 3	Average	Execution	Search	Adaptability	Time	Average
Orchestra-o1	Qwen3.5-9B	69.8	57.0	50.0	60.0	15.6	37.5	0.0	0.0	13.3
	SFT	**81.1**	60.5	42.3	64.2	25.0	68.8	9.4	12.5	28.9
	DA-GRPO	77.4	67.4	50.0	67.9	35.9	68.8	6.3	12.5	30.9
	SFT + RWR	79.2	62.8	42.3	64.8	34.4	75.0	9.4	12.5	32.8
AORCHESTRA	Qwen3.5-9B	70.4	63.6	46.2	63.0	31.3	75.0	3.1	9.4	29.7
	SFT	76.7	69.8	57.7	70.1	35.9	75.0	26.0	18.8	39.1
	SFT + RWR	75.5	74.4	61.5	72.7	43.8	80.6	19.4	16.1	40.0
PORCHESTRA	Qwen3.5-9B	73.0	66.3	60.3	67.5	31.3	65.6	12.9	10.0	29.7
	SFT	80.2	70.9	61.5	72.4	45.3	75.0	34.4	20.6	43.8
	OPOD	**81.1**	**77.3**	**73.1**	**77.9**	43.8	**84.4**	**37.5**	**31.3**	**49.2**
	w/o RWR	**81.1**	75.6	65.4	75.8	40.6	79.7	35.9	28.1	46.1
	w/o OPD	80.2	75.6	69.2	76.1	**46.9**	**84.4**	29.7	23.4	46.1

anism. This shows that PORCHESTRA seamlessly integrates immediate harness adaptation into sub-agent execution, alleviates the adaptation delay of existing orchestration methods, and thereby more fully realizes the gains from orchestration on long-horizon tasks.

4.3 EFFECTIVENESS OF OPOD IN OPTIMIZATION (RQ2)

Obs. ❸: OPOD effectively optimizes small orchestrators for PORCHESTRA. Table 2 compares `Qwen3.5-9B` across orchestration frameworks and optimization methods. Overall, training improves small-orchestrator success on both benchmarks. Across frameworks, PORCHESTRA achieves the highest or joint-highest average success before training and retains this lead under the same training methods, preserving its framework advantage. With OPOD, PORCHESTRA further reaches 77.9% on GAIA and 49.2% on GAIA2, improving over its SFT initialization by 5.5 and 5.4 percentage points, respectively, and achieving the highest average success reported in the table.

Obs. ❹: Process feedback and outcome supervision play complementary roles in orchestrator optimization. Removing either OPD or RWR reduces average success on both benchmarks. In particular, without OPD, success drops by 7.8 and 7.9 percentage points on GAIA2's Adaptability and Time tasks. These results support the synergy between PORCHESTRA's interleaved orchestration and OPOD on dynamic tasks: the former enables on-demand harness revisions during sub-agent execution; the latter converts feedback from these interactions into process-level supervision to improve orchestration. Removing RWR also lowers average success on GAIA and GAIA2 by 2.1 and 3.1 points, respectively, supporting the complementary value of task-level outcome supervision.

4.4 COST ANALYSIS (RQ3)

Obs. ❺: PORCHESTRA improves the trade-off between task performance and inference cost. Figure 4 shows that PORCHESTRA achieves the highest success for each backbone on GAIA and GAIA2 at competitive cost. On GAIA2 with `DeepSeek-V4-Flash`, it exceeds ReAct by 11.0 percentage points at 36% lower per-task API cost and Orchestra-o1 by 11.7 points at 31% lower cost. Its efficiency stems from (1) independent sub-agent contexts and task-relevant tool sets, which substantially reduce context overhead amid GAIA2's large tool set and extensive application feedback; and (2) on-demand configuration changes during execution, which avoid wasted inference under mismatched harnesses.

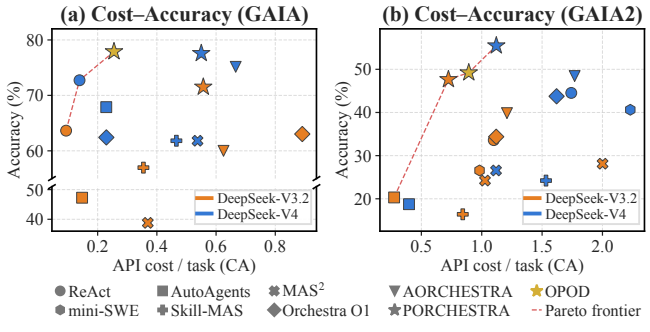
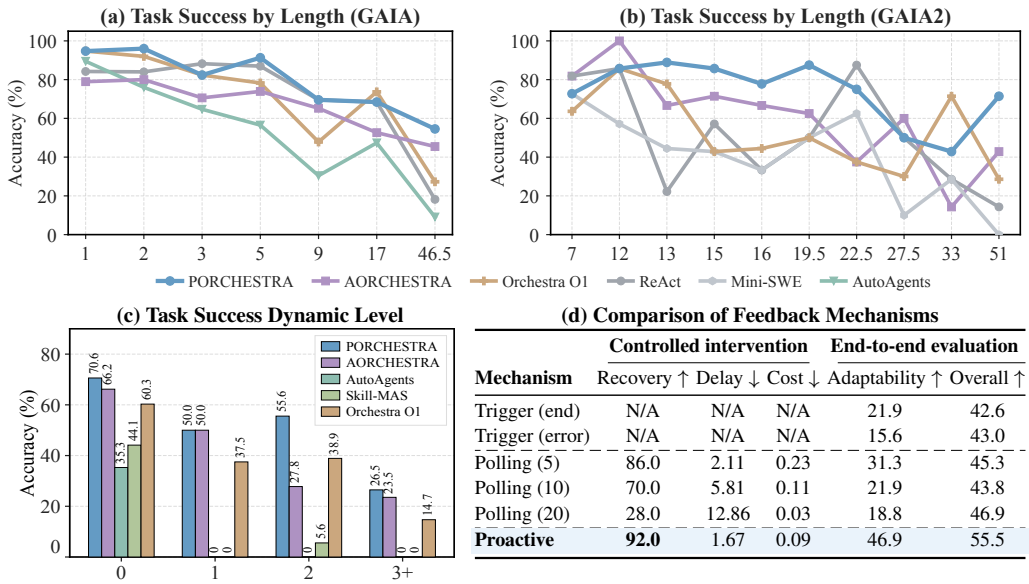


Figure 4: Cost–performance trade-offs on GAIA and GAIA2.



(d) Comparison of Feedback Mechanisms

	Controlled intervention			End-to-end evaluation	
Mechanism	Recovery ↑	Delay ↓	Cost ↓	Adaptability ↑	Overall ↑
Trigger (end)	N/A	N/A	N/A	21.9	42.6
Trigger (error)	N/A	N/A	N/A	15.6	43.0
Polling (5)	86.0	2.11	0.23	31.3	45.3
Polling (10)	70.0	5.81	0.11	21.9	43.8
Polling (20)	28.0	12.86	0.03	18.8	46.9
Proactive	**92.0**	1.67	0.09	46.9	55.5

Figure 3: Analysis of PORCHESTRA. Task success by (a,b) task length and (c) dynamicity. (d) Intervention mechanisms under controlled intervention and end-to-end evaluation. Recovery is in percent, delay in execution steps (including idle), and cost in CA per case.

Obs. ❻: OPOD extends the cost–performance frontier. Figure 4 shows the OPOD-trained `Qwen3.5-9B` orchestrator on the Pareto frontier of both GAIA and GAIA2. By reducing reliance on larger orchestrators, OPOD extends the frontier toward comparable performance at lower cost.

4.5 TRACING THE GAINS FROM PROACTIVE ORCHESTRATION (RQ4)

Obs. ❼: PORCHESTRA shows a clear advantage on longer and more dynamic tasks. (1) **Task length.** For each task solved by at least one method, we estimate its required length by the fewest environment actions among successful trajectories and group tasks accordingly. As shown in Figure 3(a,b), PORCHESTRA's advantage becomes more pronounced in longer task groups on both GAIA and GAIA2. (2) **Task dynamicity.** Figure 3(c) groups GAIA2 tasks by the number of future exogenous events. Notably, PORCHESTRA has its largest relative advantage in the group with two future events, representing moderate-to-high dynamicity. Together, these findings highlight the distinctive value of proactive orchestration for long-horizon, dynamic interaction tasks.

Obs. ❽: End-to-end performance gains trace to seamless adaptation enabled by proactive feedback. To trace PORCHESTRA's performance gains, we replace proactive feedback with two triggering schemes: *Trigger (end/error)* invokes the orchestrator when a sub-agent finishes or encounters an error; *Polling (5/10/20)* re-orchestrates every 5, 10, or 20 execution steps. Figure 3(d) compares them under controlled intervention and end-to-end evaluation. (1) **Controlled intervention.** We randomly revoke tool access during sub-agent execution to simulate tool unavailability. More frequent polling improves recovery and reduces response delay at higher cost. Proactive feedback achieves 92% recovery with a mean delay of 1.67 steps. (2) **End-to-end evaluation.** On GAIA2, more frequent polling improves performance on the Adaptability subset but can disrupt execution continuity and reduce overall success; predefined triggers struggle to detect adjustment needs without explicit errors. By enabling timely adaptation with execution continuity, PORCHESTRA outperforms these variants overall, translating seamless intervention into higher task success.

5 CONCLUSION

We introduce PORCHESTRA, a proactive orchestration framework where sub-agents proactively communicate with the orchestrator at any logical point, enabling seamless harness adaptation in dynamic, long-horizon tasks. OPOD, in turn, converts this feedback into process-level supervision to optimize the orchestrator. Experiments across three benchmarks show improved task performance, favorable cost–performance trade-offs, and more effective training of small orchestrators.

AI USE STATEMENT

We used generative AI tools to assist with literature research, coding, translation and language editing, interpretation of experimental results, drafting figure captions and labels, and manuscript formatting. All AI-assisted content was reviewed by the authors, who take responsibility for the final paper, including its claims, analyses, and artifacts.

REPRODUCIBILITY STATEMENT

We provide the inference and training algorithms in Appendix A, benchmark descriptions and implementation settings in Appendix E, theoretical proofs in Appendix D, and prompt templates in Appendix F. An anonymous code repository is linked in the abstract.

REFERENCES

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2306.13649.

Anthropic. Create custom subagents, 2026. URL https://code.claude.com/docs/en/sub-agents. Accessed September 17, 2026.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F. Karlsson, Jie Fu, and Yemin Shi. AutoAgents: A framework for automatic agent generation. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, pp. 22–30, 8 2024a. doi: 10.24963/ijcai.2024/3. URL https://doi.org/10.24963/ijcai.2024/3.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=EHg5GDnyq1.

Zhiqing Cui, Haotong Xie, Jiahao Yuan, Cheng Yang, Hanqing Wang, Yuxin Wu, Yifan Wu, Siru Zhong, Tao Yu, Yifu Guo, Siyu Zhang, Xinlei Yu, Qibing Ren, and Usman Naseem. Uno-orchestra: Parsimonious agent routing via selective delegation, 2026. URL https://arxiv.org/abs/2605.05007.

Yufeng Du, Minyang Tian, Srikanth Ronanki, Subendhu Rongali, Sravan Babu Bodapati, Aram Galstyan, Azton Wells, Roy Schwartz, Eliu A Huerta, and Hao Peng. Context length alone hurts LLM performance despite perfect retrieval. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 23281–23298, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-335-7. doi: 10.18653/v1/2025.findings-emnlp.1264. URL https://aclanthology.org/2025.findings-emnlp.1264/.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, et al. ARE: Scaling up agent environments and evaluations. *arXiv preprint arXiv:2509.17158*, 2025. URL https://arxiv.org/abs/2509.17158.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, et al. GAIA2: Benchmarking LLM agents on dynamic and asynchronous environments. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=9gw03JpKK4.

Jay R Galbraith. Organization design: An information processing view. *Interfaces*, 4(3):28–36, 1974.

Google DeepMind. Gemini 3 Flash model card, 2025. URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Flash-Model-Card.pdf.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

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.

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

Zixuan Ke, Yifei Ming, Austin Xu, Ryan Chin, Xuan-Phi Nguyen, Prathyusha Jwalapuram, Jiayu Wang, Semih Yavuz, Caiming Xiong, and Shafiq Joty. MAS-Orchestra: Understanding and improving multi-agent reasoning through holistic orchestration and controlled benchmarks. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=3fGXBm4c3S.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Hehai Lin, Qi Yang, and Chengwei Qin. Skill-mas: Evolving meta-skill for automatic multi-agent systems. *arXiv preprint arXiv:2606.18837*, 2026.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

James G March and Herbert A Simon. *Organizations*. John wiley & sons, 1993.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: a benchmark for general AI assistants. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=fibxvahvs3.

OpenAI. Subagents, 2026. URL https://learn.chatgpt.com/docs/agent-configuration/subagents. Accessed September 17, 2026.

Jan Peters and Stefan Schaal. Reinforcement learning by reward-weighted regression for operational space control. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pp. 745–750, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937933. doi: 10.1145/1273496.1273590. URL https://doi.org/10.1145/1273496.1273590.

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

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. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=8nGX6WUKE7.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. TRL: Transformers Reinforcement Learning, 2020. URL https://github.com/huggingface/trl.

Kun Wang, Guibin Zhang, ManKit Ye, Xinyu Deng, Dongxia Wang, Xiaobin Hu, Jinyang Guo, Yang Liu, and Yufei Guo. MAS^2: Self-generative, self-configuring, self-rectifying multi-agent systems. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=qumy27hMDY.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of computer science*, 18(6):186345, 2024.

Yinjie Wang, Ling Yang, Guohao Li, Mengdi Wang, and Bryon Aragam. ScoreFlow: Mastering LLM agent workflows via score-based preference optimization. *arXiv preprint arXiv:2502.04306*, 2025.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. DeepSeek-V4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

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.

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

Fan Zhang, Vireo Zhang, Shengju Qian, Haoxuan Li, Hao Wu, Jinyang Wu, Donghao Zhou, Zhihong Zhu, Zheng Lian, Xin Wang, and Pheng-Ann Heng. Orchestra-o1: Omnimodal agent orchestration, 2026. URL https://arxiv.org/abs/2606.13707.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. G-Designer: Architecting multi-agent communication topologies via graph neural networks. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=LpE54NUnmO.

Siyan Zhao, Zhihui Xie, Mengchen Liu, Jing Huang, Guan Pang, Feiyu Chen, and Aditya Grover. Self-distilled reasoner: On-policy self-distillation for large language models. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=Jpxfof0EaS.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. LlamaFactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL http://arxiv.org/abs/2403.13372.

APPENDIX CONTENTS

A Algorithms **14**
A.1 PORCHESTRA Inference 14
A.2 OPOD Training 15

B Additional Experiments and Analysis **16**
B.1 Effect of OPD Weight (RQ5) 16
B.2 Harness Revision and Communication Ablations (RQ6) 16
B.3 Communication Cost Analysis (RQ7) 17
B.4 Distillation Objective Comparison (RQ8) 17

C Case Studies **18**
C.1 Proactive Adaptation in Execution 18
C.2 Process Supervision in Optimization 22

D Proofs **25**
D.1 Proof of Proposition 1 25
D.2 Proof of Proposition 2 26

E Implementation Details **29**
E.1 Baseline and Benchmark Details 29
E.2 Training Configuration 31
E.3 Inference and Evaluation Configuration 32

F Prompts and Communication Protocol **34**
F.1 Orchestrator Prompts 34
F.2 Sub-Agent Prompts 37
F.3 Generative Reward Model Prompt 40

G Limitations and Future Work **41**

H Ethics Statement **41**

A ALGORITHMS

A.1 PORCHESTRA INFERENCE

Algorithm 1 Interleaved orchestrator and sub-agent execution in PORCHESTRA.

Require: orchestrator π^{orch}; sub-agent model M_{sub}; execution budgets
Input: User goal G
Output: Final answer y, or accumulated results at the budget limit

Orchestrator: delegation and feedback-driven coordination
$s^{\text{orch}} \leftarrow \text{Initialize}(G)$
select `Delegate`(Φ_i); create sub-agent i with $h_i \leftarrow \varnothing$
mark sub-agent i as ready to execute
while budgets permit **do**
  if a sub-agent i is ready to execute **then**
    $o^{\text{orch}} \leftarrow \text{SubAgent}(i)$ ▷ Run until feedback or completion
    update s^{orch} with o^{orch}, worker status, and remaining budgets
  $a^{\text{orch}} \sim \pi^{\text{orch}}(\cdot \mid s^{\text{orch}})$ over valid actions
  if $a^{\text{orch}} =$ `Revise`(Φ_i') and sub-agent i is paused **then**
    $\Phi_i \leftarrow \Phi_i'$; apply the task and resource changes
    append the applied changes as observation o^{comm} to h_i
    mark the same sub-agent i as ready to resume ▷ Preserve its local history
  else if $a^{\text{orch}} =$ `Delegate`(Φ_j) **then**
    create sub-agent j with $h_j \leftarrow \varnothing$; set $i \leftarrow j$ and mark it ready
  else if $a^{\text{orch}} =$ `Stop`(i) **then**
    terminate sub-agent i and update s^{orch} ▷ No worker is ready
    continue ▷ Orchestrator decides again
  else if $a^{\text{orch}} =$ `Finish`(y) **then**
    return y ▷ Complete the overall task
return accumulated results in s^{orch}

Sub-Agent(i)**: local execution and proactive feedback**
while sub-agent i has local and total execution budget **do**
  $a^{\text{sub}} \leftarrow M_{\text{sub}}(I_{\text{sub}}(\Phi_i, h_i, \mathcal{A}_i^{\text{comm}}))$
  if $a^{\text{sub}} \in \Omega_i$ **then**
    $o^{\text{env}} \leftarrow \text{Execute}(a^{\text{sub}})$; append $(a^{\text{sub}}, o^{\text{env}})$ to h_i
  else ▷ `revise`, `submit`, or `stop`
    $o^{\text{orch}} \leftarrow$ (status, request, evidence) from a^{sub} and h_i
    append o^{orch} to h_i
    if $a^{\text{sub}} =$ `revise` **then**
      pause sub-agent i, preserving (Φ_i, h_i) ▷ Await the orchestrator's adjustment
    else
      close sub-agent i as completed (`submit`) or terminated (`stop`)
    return o^{orch} to the orchestrator ▷ Sub-agent initiates the handoff
pause sub-agent i; **return** budget-exhaustion feedback to the orchestrator

Algorithm 1 describes inference from the complementary perspectives of the orchestrator and the sub-agent. The orchestrator delegates tasks and responds to feedback, while the sub-agent executes local actions and proactively returns control when a harness adjustment is needed. A revision updates a paused worker's configuration and records the applied changes in its local history, allowing the same worker to resume with its accumulated context.

A.2 OPOD TRAINING

Algorithm 2 Feedback-guided policy optimization with OPOD.

Require: expert data $\mathcal{D}_{\text{SFT}}$; task distribution $\mathcal{D}$; OPD weight λ
Input: Base model M_{θ_0}
Output: Optimized orchestration policy π_θ

$\theta_{\text{SFT}} \leftarrow \text{SFT}(\theta_0, \mathcal{D}_{\text{SFT}})$; $\theta \leftarrow \theta_{\text{SFT}}$ ▷ Initialize the student
freeze M_{θ_0} and $M_{\theta_{\text{SFT}}}$ ▷ Base GRM and SFT teacher
$\mathcal{B} \leftarrow \text{Collect}(\pi_{\theta_{\text{SFT}}}, \mathcal{D})$ ▷ Inference process (Algorithm 1)
$\mathcal{D}_R \leftarrow \text{SuccessfulDecisions}(\mathcal{B})$; $\mathcal{D}_P \leftarrow \varnothing$
for each delegation or revision attribution unit u_t in $\mathcal{B}$ **do**
    $(\texttt{FLAG}_t, \mathcal{H}_t) \leftarrow M_{\theta_0}(I_{\text{GRM}}(u_t))$ ▷ Feedback attribution
    if $\texttt{FLAG}_t = \texttt{true}$ **then**
        $\mathcal{D}_P \leftarrow \mathcal{D}_P \cup \{(s_t^{\text{orch}}, \mathcal{H}_t)\}$
for each minibatch from $\text{PairByContext}(\mathcal{D}_R, \mathcal{D}_P)$ **do**
    $\mathcal{L}_R \leftarrow \text{RWR}(\pi_\theta; \text{successful-decision targets})$ ▷ Outcome supervision
    for each OPD record $(s, \mathcal{H})$ in the minibatch **do**
        $a \sim \pi_\theta(\cdot \mid s)$ ▷ Current-student generation
        $p_k \leftarrow \pi_\theta(\cdot \mid s, a_{<k})$; $q_k \leftarrow \pi^{+}_{\theta_{\text{SFT}}}(\cdot \mid s, \mathcal{H}, a_{<k})$
        $\ell_P(s) \leftarrow \frac{1}{|a|} \sum_k D_{\text{KL}}(p_k \| \text{sg}[q_k])$ ▷ Feedback-guided distillation
    $\mathcal{L}_P \leftarrow \text{Mean}_s[\ell_P(s)]$; $\theta \leftarrow \text{OptimizerStep}(\theta, \mathcal{L}_R + \lambda \mathcal{L}_P)$
return π_θ

Algorithm 2 summarizes how OPOD constructs and uses complementary outcome and process supervision. We first collect execution trajectories with the SFT policy, extracting decisions from successful trajectories as RWR targets and valid feedback hints from attribution units as OPD supervision. After pairing these records by decision context, we optimize the student on the resulting fixed dataset: RWR reinforces the stored successful decisions, while OPD guides actions newly generated by the current student at the stored contexts toward the frozen SFT teacher conditioned on the corresponding feedback hints. The OPD loss is computed as full-vocabulary reverse KL along these student-generated actions and combined with RWR to update the student. Loss normalization and supervision masks are specified in Section E.2.4.

B Additional Experiments and Analysis

We extend our evaluation through four additional research questions:

- **RQ5:** How sensitive is OPOD to the weight of process-level supervision? We vary λ on GAIA2 (Section B.1).
- **RQ6:** How do harness revision capabilities and communication protocol components contribute to proactive orchestration? We ablate these components individually (Section B.2).
- **RQ7:** Do savings in orchestration and execution offset the cost of proactive communication? We break down inference costs on GAIA and GAIA2 (Section B.3).
- **RQ8:** Which distillation objective is best suited to OPOD? We compare reverse KL, forward KL, and JSD on GAIA2 (Section B.4).

B.1 Effect of OPD Weight (RQ5)

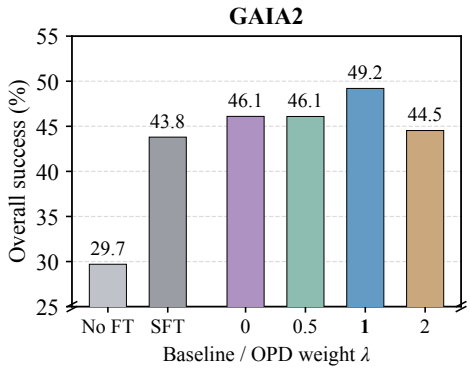


Figure 5: Hyperparameter analysis of λ on GAIA2.

Obs. ❾: A moderate OPD weight balances process and outcome supervision. We vary $\lambda \in \{0, 0.5, 1, 2\}$ on GAIA2 to assess sensitivity to the relative strength of process and outcome supervision, with $\lambda = 0$ removing OPD. Figure 5 shows that $\lambda = 1$ achieves the highest success among the tested settings, suggesting a favorable balance between the two signals. Smaller weights may underuse process feedback, while larger weights may overemphasize the self-teacher relative to terminal rewards. Across the three nonzero settings, success ranges from 44.5% to 49.2%: a spread of 4.7 percentage points over a fourfold change in weight. This indicates moderate sensitivity within the tested range and supports $\lambda = 1$ as our default.

B.2 Harness Revision and Communication Ablations (RQ6)

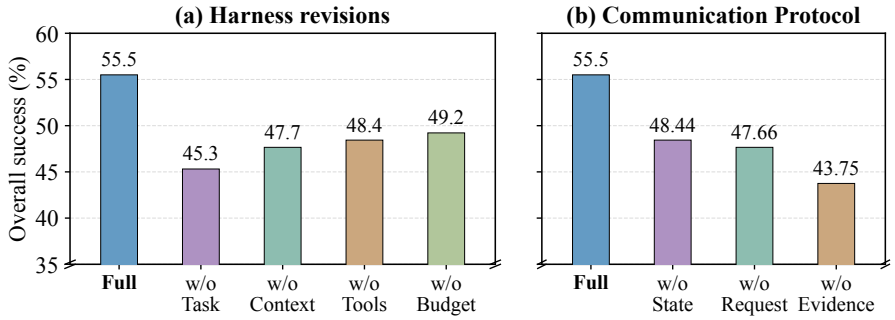


Figure 6: Ablations on GAIA2: (a) harness revisions and (b) communication protocol components. Protocol ablation scores average two runs.

Harness Revision Ablation

Obs. ❿: Task and resource revisions contribute to effective proactive orchestration. On GAIA2, we disable revisions to task scope, working context, tools, or execution budget one at a

time, preserving each component's initial configuration and the other revision capabilities. The context ablation still allows local tool observations to accumulate. Figure 6(a) shows that removing any revision capability reduces task success. Task revision has the largest observed effect, with a 10.2-point drop, highlighting the value of adjusting delegated subgoals as requirements evolve. Disabling context, tool, or budget updates yields drops of 7.8, 7.1, and 6.3 points, respectively. The losses persist with the other three capabilities retained, indicating that task and resource revisions provide complementary support during execution.

COMMUNICATION PROTOCOL ABLATION

Obs. ⓫: Effective orchestration relies on all three communication components. We remove State, Request, or Evidence from the communication protocol one at a time on GAIA2 and average each ablation over two runs. Figure 6(b) shows that every removal reduces success relative to the full protocol. Without State, the orchestrator loses the sub-agent's summary of progress, uncertainty, and blockage, making it harder to understand why intervention is needed. Without Request, the orchestrator must infer the direction of adjustment from the remaining signals, without an explicit proposal from the sub-agent carrying out the task. Removing Evidence causes the largest drop: the orchestrator lacks the supporting tool observations needed to assess the reliability of the sub-agent's report and justify an update. These results highlight the complementary roles of execution awareness, adjustment guidance, and reliable supporting evidence in proactive orchestration.

B.3 COMMUNICATION COST ANALYSIS (RQ7)

Table 3: Cost breakdown on GAIA and GAIA2 (CA per task).

Benchmark	Method	Orchestration	Communication	Sub-agent	Total
GAIA	AORCHESTRA	0.236	—	0.390	0.625
	Orchestra-o1	0.201	—	0.690	0.891
	PORCHESTRA	0.151	0.193	0.213	**0.557**
GAIA2	AORCHESTRA	0.208	—	1.001	1.209
	Orchestra-o1	0.244	—	0.875	1.120
	PORCHESTRA	0.097	0.065	0.559	**0.721**

Obs. ⓬: Lower orchestration and execution costs offset communication overhead. Table 3 compares per-task costs using `Gemini-3-Flash` as the orchestrator and `DeepSeek-V3.2` as the executor on both benchmarks. We separate routine orchestration, proactive communication, and sub-agent execution into non-overlapping components, using estimates for GAIA communication. PORCHESTRA spends less on routine orchestration and sub-agent execution than both baselines. These savings exceed the expense of generating and handling feedback, yielding the lowest total inference cost on both benchmarks. The breakdown shows that proactive communication can support efficient execution even after its own overhead is included.

B.4 DISTILLATION OBJECTIVE COMPARISON (RQ8)

Table 4: Distillation objective comparison on GAIA2. All values are success rates (%). Reverse KL uses the full OPOD results from Table 2.

Objective	Execution	Search	Adaptability	Time	Overall
Forward KL	**50.0**	71.9	28.1	21.9	43.0
JSD	40.6	75.0	**37.5**	28.1	45.3
Reverse KL	43.8	**84.4**	**37.5**	**31.3**	**49.2**

Obs. ⓭: Reverse KL best complements OPOD among the compared objectives. Our main experiments use reverse KL for the OPD loss, raising the question of whether this choice is well suited to OPOD. To examine this, we compare three formulations of the distillation loss: reverse KL,

forward KL, and Jensen–Shannon divergence (JSD). Table 4 reports their success rates on GAIA2. Reverse KL achieves the highest overall success, exceeding forward KL and JSD by 6.3 and 3.9 percentage points, respectively. These results support its use in our main joint-training configuration. One possible explanation lies in the local and retrospective nature of sub-agent feedback. Reverse KL penalizes student-favored actions assigned low probability by the feedback-augmented teacher. This targeted correction may complement RWR's reinforcement of successful actions.

C Case Studies

We present five cases illustrating how proactive sub-agent feedback supports runtime adaptation and orchestrator optimization:

- **Tool adaptation:** Sub-agents proactively request tool revisions, prompting the orchestrator to adjust their tool configurations (Figure 7).
- **Task adaptation:** Sub-agents proactively request task revisions, prompting the orchestrator to adjust their task scope (Figure 8).
- **Context allocation optimization:** OPOD supervises the orchestrator's context allocation at delegation (Figure 9).
- **Tool configuration optimization:** OPOD supervises the orchestrator's tool configuration at delegation (Figure 10).
- **Revision policy optimization:** OPOD supervises how the orchestrator revises harnesses in response to sub-agent requests (Figure 11).

C.1 Proactive Adaptation in Execution

C.1.1 Tool Adaptation

GAIA case: tool adaptation

User task: *The longest-lived vertebrate is named after an island. According to Wikipedia as of January 1, 2021, what is the 2020 estimated population of that island, to the nearest thousand?*

1. Orchestrator: Initial delegation — **Communication**

Action: `delegate_task`

Task input: Identify the longest-lived vertebrate and its namesake island, then find the island's 2020 population estimate in Wikipedia as of January 1, 2021. The instruction suggests the Greenland shark and Greenland as likely targets.

Assigned tools: `GoogleSearchAction`, `ExtractUrlContentAction`.

2. Sub-agent: Search — **Environment interaction**

Action (entry 1): `GoogleSearchAction`

Input query: `longest-lived vertebrate named after an island`

Observation: The result identifies the Greenland shark as the longest-living vertebrate.

Action (entry 2): `GoogleSearchAction`

Input query: `Greenland population 2020 Wikipedia January 1 2021`

Observation excerpt: "As of 1 January **2024** the resident population of Greenland was estimated to be 56,699, an increase of 90 (0.2%) from the previous year."

This search result does not meet the requested year and historical-source constraints.

Repeated search attempts in entries 3–24 are omitted. The sub-agent tries date constraints, Wikipedia site restrictions, and infobox or revision-history terms before the next displayed call.

GAIA case (continued)

3. Sub-agent: Page inspection — **Environment interaction**

Action (entry 25): `ExtractUrlContentAction`

Input URL: `https://en.wikipedia.org/wiki/Greenland`

Input `browse_query`: `2020 population estimate`

Observation: The extractor reports that the **2020 estimate is not explicitly provided**. Its answer lists a 2025 estimate of 56,831 and a 2022 population of 56,583, concluding that it cannot provide the requested 2020 estimate from the available content.

Further search (entry 26): Another date-specific `GoogleSearchAction` returns "No good Google Search Result was found."

4. Sub-agent: Revision request — **Communication to Orchestrator**

Action (entry 27): `revise`

Status: Greenland has been identified, but the historical population estimate remains unresolved.

Request: Provide access to **historical Wikipedia revisions** or archived versions around January 1, 2021.

Evidence: Searches have not located the required snapshot, and extraction from the current page returns more recent population figures.

5. Orchestrator: Harness update — **Communication to Sub-agent**

Action: `revise_task`

Task update: Keep the objective; retrieve the Wikipedia revision just before January 1, 2021.

Tool update: Retain search and URL extraction; add `ExecuteCodeAction`.

Context update: Supply a Python example for querying the MediaWiki revision API and reading population-related content.

Budget input: `step_budget = 15`.

Instruction excerpt: "Use ExecuteCodeAction to query the MediaWiki API for the Greenland page revision just before 2021-01-01T00:00:00Z, retrieve its content, locate the 2020 population estimate inside the infobox, and report that figure to the nearest thousand."

6. Sub-agent: Historical query — **Environment interaction**

Action (entry 28): `ExecuteCodeAction`

Input: MediaWiki query: `titles=Greenland; rvstart=2021-01-01T00:00:00Z; rvdir=older; rvlimit=1`.

Output: Population estimate: **56,081**; rounded to the nearest thousand: **56,000**.

7. Sub-agent: Result return — **Communication to Orchestrator**

Action (entry 29): `finish`; return the population estimate and its supporting tool output.

8. Orchestrator: Completion — **Final answer**

Action: `complete`, with `answer = "56,000"`.

Recorded evaluation: `success = true`.

Figure 7: Execution trace of proactive harness adaptation on GAIA.

Figure 7 shows how execution feedback guides tool and context adaptation. Search and current-page extraction identify Greenland but fail to retrieve the required historical population estimate. The sub-agent requests access to historical revisions, prompting the orchestrator to add `ExecuteCodeAction` and MediaWiki API guidance. The same worker resumes with its accumulated search history, retrieves 56,081, and returns the correctly rounded answer of 56,000.

C.1.2 TASK ADAPTATION

GAIA2 case: task adaptation

User task (condensed): Schedule "Exclusive Wine Bar" with Deniz Öztürk for the upcoming Sunday, 5–7 PM, tagged as work; send a "Reservation Confirmed" email and notify the user. If Deniz confirms within three minutes of the email, book the specified premium cab for Sunday at 4:30 PM and send a confirmation message to yesterday's training attendees in a new "Client Confirmation" group. Otherwise, take no further action.

1. Orchestrator: Initial delegation — **Communication**

Action: `delegate_task`

Assigned subtask: Look up Deniz, create the calendar event, send the reservation email, and report completion to the user.

Initial tool scope: Contact search, calendar creation, email sending, and user notification. Waiting, cab booking, and group messaging are not yet assigned.

2. Sub-agent: Initial app actions — **Environment interaction**

Action: `Calendar__add_calendar_event`

Key inputs: Title: "Exclusive Wine Bar"; start/end: `2024-10-20 17:00:00 / 19:00:00`; tag: `work`; attendee: Deniz Öztürk.

Action: `Emails__send_email`

Key inputs: Recipient: `deniz.ozturk@gaia2mail.com`; subject: "Reservation Confirmed"; content confirms the Sunday booking.

Observed result: The tools return a calendar-event ID and a sent-email ID. The sub-agent informs the user that the initial actions are complete.

3. Sub-agent: First follow-up request — **Communication to Orchestrator**

Trigger: A new-email notification arrives from Nalani Jensen.

Action: `revise`, with `task_scope.operation=follow_up`.

Status: The assigned setup is complete; the new notification has not yet been acted on.

Request: Review the notification and decide whether it requires further action.

Evidence: The notification sender, the previously created event and email IDs, and Deniz's contact address are passed to the orchestrator.

4. Orchestrator: Extend the task scope — **Communication to Sub-agent**

Action: `revise_task`

Task update: Check Nalani's message and ignore it if unrelated. Monitor for Deniz's confirmation, then perform the cab and group-message branch if the condition is met.

Resource update: Add inbox reading, notification waiting, cab booking, calendar lookup, and group-messaging capabilities to the same sub-agent.

Preserved constraint: No cab booking or group creation without the required confirmation.

5. Sub-agent: Second follow-up request — **Communication to Orchestrator**

Observed context: `Emails__list_emails`, with `folder_name=INBOX`, returns Nalani's message: "Hey Aurélien, I'll be there on Sunday. Nalani". It concerns Nalani, not Deniz. After a notification wait, a new email arrives from Aurelia.

Action: `revise`, with `task_scope.operation=follow_up`.

Request: Is the new message relevant to the conditional actions? Should the worker check the inbox again or finish without those actions?

Evidence: Nalani's sender and message, the new notification, and the identifiers from the initial setup.

GAIA2 case (continued)

6. Orchestrator: Refine the confirmation check — **Communication to Sub-agent**

Action: `revise_task`

Task and context update: Inspect Aurelia's message and the inbox to determine whether Deniz, or someone on her behalf, confirmed attendance. Retain the tools and conditional cab/group instructions.

Subsequent observation: Aurelia's message says, "Thank you for the confirmation, Aurélien. I will try to be there." While the worker waits, a notification arrives from `deniz.ozturk@gaia2mail.com`.

7. Sub-agent: Third follow-up request — **Communication to Orchestrator**

Action: `revise`, with `task_scope.operation=follow_up`.

Status: The worker believes the waiting window has elapsed, but the new notification may change whether the conditional actions are required.

Request: Read and evaluate the new email, or stop because of the deadline? The worker requests guidance before taking further side effects.

Evidence: The new notification and the earlier email contents and identifiers are carried forward.

The worker reports its uncertainty before committing to the next branch.

8. Orchestrator: Authorize conditional follow-up — **Communication to Sub-agent**

Action: `revise_task`

Instruction excerpt: "Verify the email from Deniz Öztürk (received at 07:02:36). If she confirmed her attendance, book the premium cab for Oct 20 at 4:30 PM, create the 'Client Confirmation' Messages group with the attendees of yesterday's 'Staff Training: Wine and Food Pairing' event, and send them the message that the VIP confirmed."

Resource update: Keep the same worker and its inbox, cab, calendar, and messaging tools.

9. Sub-agent: Execute the confirmed branch — **Environment interaction**

Inbox check: `Emails__list_emails`: `folder_name=INBOX, offset=0, limit=10`.

Observed email: From Deniz, timestamp `07:02:36`: "Thanks Aurélien, I'll be there on Sunday at 5 PM. Looking forward to it!"

Cab action: `Cabs__order_ride`; `service_type=Premium`, `ride_time=2024-10-20 16:30:00`; route: 9012 Park Ave to 4567 Broadway, New York. The tool returns **BOOKED**.

Group actions: Look up yesterday's training attendees, then call `Messages__create_group_conversation` with title "Client Confirmation" and the retrieved user IDs. Call `Messages__send_message_to_group_conversation` with content "the VIP has confirmed and preparations should begin".

Completion: `AgentUserInterface__send_message_to_user` reports the completed actions. The recorded scenario evaluation is **Valid**.

AOrchestra: End-of-subtask coordination — **Paired reference run**

Recorded behavior: The first worker schedules the event, sends the email, requests a 180-second wait, and concludes that no qualifying confirmation arrived. A second worker checks the Sent folder and inbox, then reaches the same conclusion.

Recorded outcome: **Invalid**. The evaluator identifies missing cab booking, group creation, and group messaging, as well as an email-operation count mismatch.

Execution structure: PORCHESTRA: one worker and three revisions. AORCHESTRA: two workers and no within-worker revisions.

Figure 8: Proactive task revisions under new notifications on GAIA2.

Figure 8 shows how proactive communication adapts an active delegation to new notifications. After completing the reservation setup, the worker reports successive messages and requests guidance on the conditional follow-up. Through three revisions, the orchestrator extends the task and tools, distinguishes unrelated replies from relevant confirmation, and directs verification of Deniz's response. The same worker then books the cab and sends the group message, carrying forward the earlier event, email, and contact information.

C.2 Process Supervision in Optimization

C.2.1 Context Allocation Optimization

GAIA2 OPOD case: context allocation optimization

Relevant task objective: Identify the postal code with the lowest property crime rate across the apartment catalog, save all lofts in that postal code, and report the saved properties to the user.

1. Orchestrator: Delegation with incomplete context — **Communication**

Action: `delegate_task`

Task input: Query the remaining 12 postal codes, compare rates across **all catalog postal codes**, and save the lofts in the global minimum.

Context excerpt: "Property crime rates have been queried for many zip codes already."

Missing handoff: The orchestrator already has rates for 18 postal codes, including `170 00 = 3.44`, but does not include those values in the delegation.

2. Sub-agent: Query the remaining postal codes — **Environment interaction**

Actions: `City__get_crime_rate`, with API-count and limit checks.

Observed result: All 12 remaining rates are collected. The lowest within this subset is `500 01 = 12.17`. The earlier 18 values remain unavailable in the worker's context.

3. Sub-agent: Request the missing evidence — **Communication to Orchestrator**

Action: `revise`; keep the task scope and request `context`.

Status: The remaining queries are complete, but the global minimum cannot be confirmed.

Request: Provide the previously queried postal codes and property crime rates.

Evidence: The local minimum is `500 01 = 12.17`; selecting it without the earlier results could lead to saving lofts in the wrong postal code.

4. Orchestrator: Observed runtime repair — **Communication to Sub-agent**

Action: `revise_task`

Context update: Supply all 30 rates, including `170 00 = 3.44`, and instruct the same worker to compare them before searching for and saving lofts.

Observed continuation: The worker identifies `170 00`, saves **Prague 7 Loft**, and reports the result. The recorded scenario evaluation is **Valid**.

5. Attribution: Repair the original delegation — **Training signal**

Target: The original delegation; $\texttt{FLAG}_t = \texttt{true}$.

Natural-language gradient $\mathcal{H}_t$: "Include the previously queried zip codes and their property crime rates in the delegation context so the SubAgent can determine the global minimum without needing to request additional data."

Training record: The saved OPD sample pairs the original decision context with $\mathcal{H}_t$.

Figure 9: A missing evidence handoff yields a context-repair gradient for OPOD.

Figure 9 shows how OPOD generates supervision for context allocation at delegation. When constructing the sub-agent's context, the orchestrator omits previously collected crime rates, leaving the sub-agent unable to compare all postal codes after completing its queries. The proactive request identifies information that was available at decision time but not passed to the sub-agent. Runtime revision supplies the missing data, while OPOD attributes the omission to the original delegation and generates a gradient to include those results when creating the sub-agent. Thus, even a successful trajectory can provide supervision for improving delegation.

C.2.2 TOOL CONFIGURATION OPTIMIZATION

TaskCraft OPOD case: tool configuration optimization

User task (condensed): Find the update date of an article on `academiascribes.com` comparing Claude and ChatGPT for academic writing.

1. Orchestrator: Delegate archive retrieval — **Communication**

Action: `delegate_task`

Known context: The suspected article URL returns **404**; the homepage appears repurposed.

Task input: Search Wayback Machine archives, caches, and other pages for the article's update date.

Assigned tools: `GoogleSearchAction`, `ExtractUrlContentAction`.

2. Sub-agent: Search and inspect the missing page — **Environment interaction**

Actions: Search the target URL with cache and archive terms; inspect the article using `ExtractUrlContentAction`.

Observed result: The page returns **404 Not Found**. Archive searches yield no usable snapshot, and broader results concern other websites.

3. Sub-agent: Request direct archival access — **Communication to Orchestrator**

Action: `revise`; request a tool resource.

Status: The article is unavailable, and search-based archive discovery has not found the required date.

Request: Enable direct Wayback Machine API queries and retrieval of the archive URLs they return.

Evidence: Searches return unrelated pages or no relevant results; direct archival queries may locate snapshots absent from search-engine results.

4. Attribution: Repair the delegated tool set — **Training signal**

Target: The original `delegate_task`; $\texttt{FLAG}_t = \texttt{true}$.

Natural-language gradient $\mathcal{H}_t$: "Enable ExecuteCodeAction in the delegation to allow the SubAgent to programmatically query the Wayback Machine API or construct direct archive URLs, as GoogleSearchAction proved insufficient for locating the specific archive snapshot."

Training record: The original decision context and $\mathcal{H}_t$ appear in the saved OPD training sample.

Recorded task outcome: **Timeout / Invalid**.

Figure 10: A proactive tool request supports a gradient for archive-retrieval delegation.

Figure 10 shows how OPOD generates supervision for tool configuration at delegation. When creating the sub-agent, the orchestrator assigns only search and extraction tools, failing to account for the need to retrieve the article's update date from archives after direct page access has failed. After archive searches fail, the sub-agent requests direct archival API access. OPOD attributes this resource mismatch to the delegation and generates a gradient to include `ExecuteCodeAction`, retaining the retrieval objective while improving the assigned tool set. This supervision encourages the orchestrator to configure tools suited to each sub-agent's task and known execution constraints.

C.2.3 REVISION POLICY OPTIMIZATION

TaskCraft OPOD case: revision policy optimization

User task (condensed): Find an AI-writing-assistant comparison in an Academic Writing category and report its account of ChatGPT's development origins and current versions.

TaskCraft OPOD case (continued)

1. Orchestrator: Supply a speculative domain — **Communication to Sub-agent**

Decision context: The active worker has requested context and clues to locate the official website.

Action: `revise_task`; retain search and URL extraction, with `step_budget = 15`.

Context excerpt: "The official website is likely 'academic-writing.org' or a similar site. Please search specifically using `site:academic-writing.org` combined with keywords like 'selecting', 'AI writing assistant', or 'ChatGPT'."

2. Sub-agent: Follow the supplied lead — **Environment interaction**

Action: `GoogleSearchAction`

Input query: `site:academic-writing.org "AI writing assistant" ChatGPT`

Observation: "No good Google Search Result was found". Further searches and page inspection follow.

Action: `ExtractUrlContentAction`, with URL `https://academic-writing.org`.

Observation: The extracted page describes a human essay-writing service and provides no relevant information about the requested AI-writing-assistant article.

3. Attribution: Correct the revision context — **Training signal**

Target: The original `revise_task`; $\texttt{FLAG}_t = \texttt{true}$.

Natural-language gradient $\mathcal{H}_t$: "Avoid providing unverified specific domain names in the context during revisions. Instead, instruct the SubAgent to search for the article title and keywords to identify the correct official website, preventing wasted steps on incorrect domains."

Training record: The original decision context and $\mathcal{H}_t$ appear in the saved OPD training sample.

Recorded task outcome: **Timeout / Invalid**.

Figure 11: Execution evidence identifies unsupported context introduced by a revision.

Figure 11 shows how OPOD also supervises the orchestrator's revision policy. In response to a request for help, the orchestrator supplies an unverified domain that directs the worker toward an irrelevant website. Subsequent search and extraction expose this mismatch. OPOD attributes the problem to the revision and generates a gradient to establish the website from article titles and keywords before supplying domain-specific guidance, providing supervision for how the orchestrator updates an active worker's context.

D PROOFS

D.1 PROOF OF PROPOSITION 1

Proposition 1 (Restated). Suppose a sub-agent requires at least $H_{\min}$ steps to maintain coherent execution, while re-orchestration is permitted only after its lifecycle ends. If a mismatch occurs uniformly over horizon H, then the probability of intervention within a recovery window δ satisfies

$$\Pr(D_H \le \delta) \le \min\left\{1, \frac{\delta}{H_{\min}}\right\}.$$

Proof. We formalize the adaptation delay induced by lifecycle-level re-orchestration. Consider a sub-agent execution with a fixed maximum horizon H, during which the orchestrator cannot modify the active configuration until the sub-agent reaches its lifecycle boundary. Let $\tau \in [0, H)$ denote the time at which an execution-time mismatch emerges within the current sub-agent lifecycle. Under the asynchronous-arrival assumption in Proposition 1,

$$\tau \sim \mathrm{Unif}(0, H). \tag{9}$$

Since re-orchestration is unavailable before the lifecycle boundary at H, the earliest admissible intervention occurs at time H. The resulting adaptation delay is therefore

$$D_H \triangleq H - \tau. \tag{10}$$

For any $d \in [0, H]$, the cumulative distribution function of D_H is

$$\begin{aligned} F_{D_H}(d) &= \Pr(D_H \le d) \\ &= \Pr(H - \tau \le d) \\ &= \Pr(\tau \ge H - d) \\ &= \int_{H-d}^{H} \frac{1}{H}\,\mathrm{d}\tau \\ &= \frac{d}{H}. \end{aligned} \tag{11}$$

Together with $F_{D_H}(d) = 0$ for $d < 0$ and $F_{D_H}(d) = 1$ for $d \ge H$, this yields

$$D_H \sim \mathrm{Unif}(0, H). \tag{12}$$

The expected adaptation delay is thus

$$\begin{aligned} \mathbb{E}[D_H] &= \int_0^H d \cdot \frac{1}{H}\,\mathrm{d}d \\ &= \frac{1}{H}\left[\frac{d^2}{2}\right]_0^H \\ &= \frac{H}{2}. \end{aligned} \tag{13}$$

For completeness, its second moment is

$$\mathbb{E}[D_H^2] = \int_0^H d^2 \cdot \frac{1}{H}\,\mathrm{d}d = \frac{H^2}{3}, \tag{14}$$

which gives

$$\mathrm{Var}(D_H) = \mathbb{E}[D_H^2] - \mathbb{E}[D_H]^2 = \frac{H^2}{3} - \frac{H^2}{4} = \frac{H^2}{12}. \tag{15}$$

Hence, lifecycle-level re-orchestration structurally ties both the magnitude and dispersion of adaptation delay to the sub-agent horizon H.

We next characterize timely intervention. Let $\delta > 0$ denote the effective recovery window of a mismatch, such that an orchestration adjustment is considered timely only if it becomes available within δ steps after the mismatch emerges. From the cumulative distribution above,

$$\Pr(D_H \le \delta) = F_{D_H}(\delta) = \min\left\{1, \frac{\delta}{H}\right\}. \tag{16}$$

In particular, when $\delta < H$,

$$\Pr(D_H \le \delta) = \frac{\delta}{H}, \qquad \frac{\partial}{\partial H}\Pr(D_H \le \delta) = -\frac{\delta}{H^2} < 0. \tag{17}$$

Thus, within lifecycle-level re-orchestration, increasing the temporal extent of a sub-agent necessarily decreases the probability that an asynchronously occurring mismatch can receive an intervention within a fixed recovery window.

Now suppose that maintaining a meaningful temporally extended sub-agent execution requires $H \ge H_{\min}$. Among all lifecycle-level re-orchestration schemes satisfying this requirement, define the maximum attainable timely-intervention probability as

$$C^{\star}_{\text{life}}(\delta) \triangleq \sup_{H \ge H_{\min}} \Pr(D_H \le \delta). \tag{18}$$

Since the timely-intervention probability is non-increasing in H, the supremum is attained at the smallest admissible horizon $H = H_{\min}$. Therefore,

$$C^{\star}_{\text{life}}(\delta) = \sup_{H \ge H_{\min}} \min\left\{1, \frac{\delta}{H}\right\} = \min\left\{1, \frac{\delta}{H_{\min}}\right\}. \tag{19}$$

Consequently, every lifecycle-level re-orchestration scheme that preserves the required sub-agent temporal horizon satisfies

$$\boxed{\Pr(D_H \le \delta) \le \min\left\{1, \frac{\delta}{H_{\min}}\right\}.} \tag{20}$$

For $H_{\min} \gg \delta$, lifecycle-level re-orchestration has maximum timely-intervention probability

$$C^{\star}_{\text{life}}(\delta) = \frac{\delta}{H_{\min}} \ll 1. \tag{21}$$

Equivalently, defining the unavoidable timely-intervention gap as

$$\Gamma_{\text{life}}(\delta) \triangleq 1 - C^{\star}_{\text{life}}(\delta), \tag{22}$$

we obtain, for $\delta < H_{\min}$,

$$\boxed{\Gamma_{\text{life}}(\delta) = 1 - \frac{\delta}{H_{\min}} > 0.} \tag{23}$$

Importantly, this gap follows from the lifecycle-level intervention structure rather than from the quality of the orchestrator's decision policy: before the lifecycle boundary is reached, no policy can revise the active sub-agent configuration without an admissible intervention point.

POrchestra removes this boundary-induced constraint by augmenting the sub-agent action space with proactive communication. Let τ_{comm} denote the first local decision point after mismatch emergence at which the sub-agent policy chooses to communicate and request adaptation. Its intervention delay takes the form

$$D_{\text{PORCH}} = \tau_{\text{comm}} - \tau, \tag{24}$$

rather than the residual lifecycle duration $D_H = H - \tau$. Therefore, D_{PORCH} is governed by mismatch detection and the sub-agent's execution and communication policy, and is not structurally constrained to scale with the predefined lifecycle horizon H. In particular, reducing the intervention timescale no longer requires reducing the horizon over which the same sub-agent can maintain a coherent local execution trajectory. This establishes the claimed temporal decoupling. □

D.2 Proof of Proposition 2

Proposition 2 (Restated). Under bounded attribution noise, terminal rewards propagate through an attribution horizon $O(H_{\text{orch}})$ times longer than that of OPD feedback, thereby accumulating noise over a longer execution chain. Furthermore, when the bias of OPD feedback is bounded, there exists a weight $\lambda > 0$ such that OPOD yields lower expected error relative to ideal process-level supervision than RWR alone.

Proof. We analyze the learning signal associated with an orchestration decision at round t through a bias–variance decomposition. Throughout, we condition on the current decision context and model parameters, treating the ideal update, attribution horizons, and process-supervision bias as fixed. Let $g_t^\star$ denote the ideal decision-level orchestration update direction obtained by evaluating the current decision without downstream stochasticity or attribution error. Here, we represent ideal process-level supervision by $g_t^\star$ and measure error as the expected squared distance of the normalized update from this reference. The proof uses the following sufficient conditions: stochastic attribution perturbations are bounded, zero-mean martingale differences; RWR and OPD perturbations are conditionally uncorrelated; the OPD bias is bounded; and terminal supervision has non-zero variance.

We first compare the attribution horizons of outcome-level and process-level supervision. Let $H_{\mathrm{orch}}^{(t)}$ denote the number of remaining orchestration rounds after decision t, and let H_{sub} denote the characteristic execution horizon of a sub-agent. A terminal reward associated with decision t is observed only after all subsequent orchestration rounds and their corresponding sub-agent executions have completed. Assuming that the remaining orchestration rounds have comparable sub-agent execution lengths, its attribution horizon scales as

$$L_t^R = \Theta((1 + H_{\mathrm{orch}}^{(t)})H_{\mathrm{sub}}). \tag{25}$$

By contrast, PORCHESTRA returns proactive feedback during the execution of the current sub-agent. When feedback arrives after a comparable fraction of the current execution, the corresponding OPD attribution horizon scales as

$$L_t^P = \Theta(H_{\mathrm{sub}}). \tag{26}$$

Consequently,

$$\frac{L_t^R}{L_t^P} = \Theta\Big(1 + H_{\mathrm{orch}}^{(t)}\Big), \tag{27}$$

where the additive one accounts for the current execution. This gives an $O(H_{\mathrm{orch}})$ attribution-horizon ratio over a task with H_{orch} orchestration rounds under these comparable-length assumptions.

We next characterize horizon-induced stochasticity. Suppose each downstream sub-agent–environment interaction introduces a zero-mean attribution perturbation. For terminal rewards,

$$g_t^R = g_t^\star + \sum_{k=1}^{L_t^R} \xi_{t,k}^R, \tag{28}$$

where

$$\mathbb{E}[\xi_{t,k}^R \mid \mathcal{F}_{t,k-1}] = 0, \qquad \mathbb{E}[\|\xi_{t,k}^R\|^2 \mid \mathcal{F}_{t,k-1}] \le \sigma_R^2. \tag{29}$$

Here, $\mathcal{F}_{t,k}$ denotes the execution history up to the k-th downstream interaction. Under the martingale-difference assumption, cross-step covariance terms vanish, yielding

$$\begin{aligned} V_R &\triangleq \mathbb{E}[\|g_t^R - g_t^\star\|^2] \\ &= \mathbb{E}\left[\left\|\sum_{k=1}^{L_t^R} \xi_{t,k}^R\right\|^2\right] \\ &\le L_t^R \sigma_R^2. \end{aligned} \tag{30}$$

The variance bound grows linearly with the attribution horizon.

The OPD signal is generated from proactive feedback returned within the current sub-agent execution. Because this feedback is converted into a natural-language gradient and then into a feedback-augmented proxy teacher, it may introduce systematic attribution bias. We therefore write

$$g_t^P = g_t^\star + b_t + \sum_{k=1}^{L_t^P} \xi_{t,k}^P, \tag{31}$$

where b_t denotes the bias induced by the process-level proxy supervision and $\|b_t\| \le B$ for some finite B. Similarly, assume

$$\mathbb{E}[\xi_{t,k}^P \mid \mathcal{G}_{t,k-1}] = 0, \qquad \mathbb{E}[\|\xi_{t,k}^P\|^2 \mid \mathcal{G}_{t,k-1}] \le \sigma_P^2. \tag{32}$$

Then

$$V_P \triangleq \mathbb{E}[\|g_t^P - \mathbb{E}[g_t^P]\|^2] \le L_t^P \sigma_P^2. \tag{33}$$

Thus, the variance bounds scale with the respective attribution horizons: up to $O(H_{\text{orch}} H_{\text{sub}})$ for terminal supervision and $O(H_{\text{sub}})$ for process supervision, with the latter also introducing finite bias b_t.

We now analyze their combination. The OPOD objective

$$\mathcal{L}_{\text{OPOD}} = \mathcal{L}_{\text{RWR}} + \lambda \mathcal{L}_{\text{OPD}} \tag{34}$$

induces the raw update $g_t^{\text{raw}} = g_t^R + \lambda g_t^P$. To compare the signals at a common scale, we analyze the normalized update

$$g_t^{\text{joint}} = \frac{g_t^R + \lambda g_t^P}{1+\lambda}. \tag{35}$$

This normalization preserves the raw update direction. Define

$$\alpha = \frac{\lambda}{1+\lambda} \in [0,1), \qquad g_t^{\text{joint}} = (1-\alpha) g_t^R + \alpha g_t^P. \tag{36}$$

Let

$$\varepsilon_t^R = g_t^R - g_t^\star, \qquad \varepsilon_t^P = g_t^P - g_t^\star - b_t. \tag{37}$$

Then

$$g_t^{\text{joint}} - g_t^\star = (1-\alpha)\varepsilon_t^R + \alpha \varepsilon_t^P + \alpha b_t. \tag{38}$$

By assumption,

$$\mathbb{E}[\langle \varepsilon_t^R, \varepsilon_t^P \rangle] = 0. \tag{39}$$

Since both stochastic components are zero mean, their cross terms with b_t also vanish in expectation. Therefore,

$$\begin{aligned} \mathcal{E}_t(\alpha) &\triangleq \mathbb{E}[\|g_t^{\text{joint}} - g_t^\star\|^2] \\ &= (1-\alpha)^2 V_R + \alpha^2 V_P + \alpha^2 \|b_t\|^2. \end{aligned} \tag{40}$$

Using terminal-reward RWR alone corresponds to $\alpha = 0$ and gives $\mathcal{E}_t(0) = V_R$. Moreover,

$$\left. \frac{\partial \mathcal{E}_t(\alpha)}{\partial \alpha} \right|_{\alpha=0} = -2V_R < 0, \tag{41}$$

where the strict inequality follows from $V_R > 0$. Since $\|b_t\| < \infty$ and $V_P < \infty$, continuity implies that some sufficiently small $\alpha > 0$ satisfies $\mathcal{E}_t(\alpha) < \mathcal{E}_t(0)$. By the one-to-one mapping

$$\lambda = \frac{\alpha}{1-\alpha}, \tag{42}$$

there exists a non-zero $\lambda > 0$ satisfying

$$\boxed{\mathbb{E}[\|g_t^{\text{joint}} - g_t^\star\|^2] < \mathbb{E}[\|g_t^R - g_t^\star\|^2].} \tag{43}$$

Under the stated assumptions, a suitable positive OPD weight reduces the normalized update's expected squared error. The shorter feedback horizon limits the number of downstream interactions that can contribute attribution noise, while the bounded bias captures deviations of feedback-based supervision from the ideal update direction. Combining RWR and OPD preserves task-level outcome supervision while introducing local corrective signals, addressing sparse supervision and long-horizon attribution in orchestration-policy optimization. □

E Implementation Details

E.1 Baseline and Benchmark Details

E.1.1 Baseline Details

We group inference-time baselines into *single-agent systems*, *offline orchestration*, and *re-orchestration*, as in the main experiments. We evaluate them on GAIA, GAIA2, and SWE-Bench, and compare SFT, RWR, and DA-GRPO for orchestrator optimization on GAIA and GAIA2.

ReAct. ReAct (Yao et al., 2023) interleaves reasoning and actions: reasoning traces support planning and progress tracking, while actions obtain information from the environment to inform subsequent decisions. The full task is handled within one agent's interaction history, making it a single-agent baseline. We instantiate this reasoning–action loop with each benchmark's available tools.

Mini-SWE. Mini-SWE (Yang et al., 2024) is a minimal software-engineering agent that uses shell commands and a linear message history to inspect, modify, and test code. Its planning and execution remain within a single agent, providing another single-agent baseline. We use its repository interaction on SWE-Bench and adapt its tool interface to GAIA and GAIA2 while retaining the same execution pattern.

Skill-MAS. Skill-MAS (Lin et al., 2026) represents orchestration expertise as an evolvable meta-skill covering task decomposition, agent engineering, and workflow orchestration. At inference time, the skill guides the generation of a task-specific agent team and executable workflow before execution, corresponding to offline orchestration in our comparison. We evaluate its initial meta-skill and connect the generated executors to each benchmark's tools.

AutoAgents. AutoAgents (Chen et al., 2024a) constructs a task-specific agent team and initial execution plan during a drafting stage, specifying roles and tool assignments before execution. In our offline-orchestration comparison, we fix these roles and the generated workflow and connect tool-using agents to each benchmark environment. Individual agents retain the ability to reason over observations and plan local actions within their assigned tasks.

MAS^2. MAS^2 (Wang et al., 2026) uses a Generator–Implementer–Rectifier team to construct and adapt multi-agent systems. The Generator produces a workflow, the Implementer assigns model backbones, and the Rectifier modifies the system when predefined conditions, such as execution failures or resource limits, are triggered. It represents re-orchestration driven by explicit triggers. We instantiate its workflow operators with benchmark-specific executors; on SWE-Bench, these operators generate, review, and select patch candidates.

Orchestra-o1. Orchestra-o1 (Zhang et al., 2026) combines modality-aware task decomposition, sub-agent specialization, and dependency-aware parallel execution. The orchestrator assigns instructions, context, tools, and model backends to independent subtasks, then uses their returned summaries to plan the next delegation round or produce a final answer. Re-orchestration therefore occurs between rounds of sub-agent execution. We retain this staged delegation; on SWE-Bench, workers generate independent repair candidates for refinement and final selection.

AOrchestra. AOrchestra (Ruan et al., 2026) represents each sub-agent as a four-tuple of instruction, context, tools, and model, enabling on-demand executor specialization. Its orchestrator alternates between delegating to a newly instantiated sub-agent and finishing the task, using returned execution results to configure subsequent delegations. This places re-orchestration at sub-agent return boundaries. We preserve this delegation-and-return mechanism while connecting executors to each benchmark's tool environment.

SFT. SFT denotes standard supervised fine-tuning on offline expert trajectories, maximizing the likelihood of expert actions given their decision contexts. In our experiments, we fine-tune `Qwen3.5-9B` on orchestration decisions collected on TaskCraft for GAIA and on training scenar-

ios for GAIA2. Each framework has its own SFT checkpoint, trained on structurally valid decisions from successful and unsuccessful expert trajectories, and initializes subsequent optimization.

RWR. RWR denotes reward-weighted regression, weighting action log-likelihoods by trajectory rewards. With binary success rewards, its objective reduces to supervised learning on successful trajectories, providing an outcome-supervised baseline. We collect trajectories from each framework's SFT policy and fine-tune its orchestrator on successful decision records. We evaluate RWR with Orchestra-o1, AORCHESTRA, and PORCHESTRA on GAIA and GAIA2.

DA-GRPO. DA-GRPO (Zhang et al., 2026) is the decision-aligned group relative policy optimization method introduced with Orchestra-o1. It scores candidate decisions against expert references using a rubric for format correctness, action validity, tool reasonableness, and decision quality, then forms group-relative advantages for policy optimization. This provides process-level supervision without executing each candidate in the environment. On GAIA and GAIA2, we train Orchestra-o1 using eight candidates per group and `DeepSeek-V3.2` for rubric scoring.

E.1.2 BENCHMARK DETAILS

Table 5 summarizes the evaluated subsets, available tools, tool counts, and evaluation procedures for the three benchmarks.

Table 5: Benchmark subsets, available tools, and evaluation procedures.

Benchmark	Evaluation subset	Tools	Tool count	Evaluation
GAIA	2023 validation; 165 tasks	`GoogleSearchAction`, `ExtractUrlContentAction`, `ExecuteCodeAction`, `ImageAnalysisAction`, `ParseAudioAction`.	5	Answer matching + semantic check.
GAIA2	Mini subset; 128 scenarios	Examples: `Emails__search_emails`, `Calendar__add_calendar_event`, `Shopping__add_to_cart`, `SystemApp__wait_for_notification`, ...	101	ARE scenario validation.
SWE-Bench Verified	Fixed test subset; 100 issues	`open`, `scroll_up`, `scroll_down`, `goto`, `edit`, `str_replace`, `insert`, `create`, `delete_lines`, `search_file`, `search_dir`, `find_file`, `state`, and `bash`.	14	Official patch evaluation.

GAIA. GAIA (Mialon et al., 2024) evaluates general-purpose assistants on real-world questions that combine multi-step reasoning, web research, tool use, and multimodal information processing. We evaluate the full 2023 validation set of 165 tasks, comprising 53 Level-1, 86 Level-2, and 26 Level-3 tasks, with task-provided attachments available to the agents. The evaluation implementation combines normalized numerical, list, and string matching against reference answers with an LLM-based semantic-equivalence fallback. Overall success is computed across all 165 tasks, weighted by task count across levels. For orchestrator training, we follow AORCHESTRA (Ruan et al., 2026) and use its released training set of 2,339 unique TaskCraft tasks.

GAIA2. GAIA2 (Froger et al., 2026) benchmarks digital assistants on completing user requests in dynamic, asynchronous application environments. It is built on Meta Agents Research Environments (ARE) (Froger et al., 2025), which simulates stateful applications and delivers user messages and environmental events during execution. Tasks require agents to retrieve information and perform operations across applications such as email, calendars, messaging, and shopping, while adapting to incoming updates and temporal constraints. We evaluate four capability-specific subsets:

- **Execution:** Following multi-step instructions and producing the required application state changes, such as sending messages or updating records.
- **Search:** Locating and combining information distributed across applications, such as emails, contacts, and files, to answer a query or identify the correct entities for subsequent actions.
- **Adaptability:** Adjusting ongoing task execution in response to asynchronous user updates or environmental events that change the requirements or relevant information.
- **Time:** Performing actions within specified time windows or deadlines, including deciding when to act and when to wait for future notifications.

The evaluation subset contains 128 mini scenarios, with 32 per category. Training uses the remaining 512 scenarios from these four categories, with 128 per category. The available tool catalog contains 101 tools, including the examples in Table 5. Each scenario initializes its own application state; actions and notifications within the scenario preserve preceding state changes. ARE's scenario validator checks required events and outcomes using task-specific rules and semantic checks. Overall success is the fraction of valid scenarios among the 128 evaluation tasks, with timeouts and missing outputs counted as unsuccessful.

SWE-Bench Verified. SWE-Bench (Jimenez et al., 2024) evaluates software-engineering agents on resolving real GitHub issues. Following AORCHESTRA (Ruan et al., 2026), we evaluate a fixed subset of 100 issues from the Verified test split and adopt its containerized repository execution environment and patch-evaluation setup.

E.2 TRAINING CONFIGURATION

E.2.1 BASE MODEL AND WARM-START STAGE

We initialize the trainable orchestrator from `Qwen3.5-9B` and perform SFT separately for Orchestra-o1, AORCHESTRA, and PORCHESTRA on each benchmark. Expert trajectories use `Gemini-3-Flash` as the orchestrator. For GAIA, we collect one pass over the 2,339 TaskCraft tasks with each of two execution backbones, `DeepSeek-V3.2` and `DeepSeek-V4-Flash`. For GAIA2, we collect four passes over the 512 training scenarios with `DeepSeek-V4-Flash` as the executor. Table 6 reports the decision records retained for SFT.

Table 6: SFT data by benchmark and orchestration framework. Task pools and collection passes precede filtering; decision counts refer to the retained examples.

Benchmark	Framework	Task pool	Passes	Train decisions
GAIA	Orchestra-o1	2,339	2	11,746
	AORCHESTRA	2,339	2	9,977
	PORCHESTRA	2,339	2	9,999
GAIA2	Orchestra-o1	512	4	5,276
	AORCHESTRA	512	4	4,087
	PORCHESTRA	512	4	4,539

SFT uses structurally valid orchestration decisions from both successful and unsuccessful expert trajectories. Subsequent optimization starts from the corresponding framework's SFT checkpoint; sub-agent models remain fixed.

E.2.2 TRAJECTORY COLLECTION AND OPTIMIZATION DATA

For RWR and OPOD, we collect trajectories from each framework's SFT policy using `DeepSeek-V4-Flash` as the executor, with one pass over the TaskCraft training set for GAIA and four passes over the training scenarios for GAIA2. Successful trajectories provide RWR targets, while OPOD additionally uses feedback from both successful and unsuccessful trajectories for OPD supervision. The two objectives are paired by decision context. During optimization, the stored

contexts and hints remain fixed while the current student resamples actions for OPD. DA-GRPO samples and scores candidate actions at expert decision contexts.

E.2.3 Optimization Hyperparameters

Table 7: Main SFT and joint OPOD training settings.

Parameter	**SFT**	**OPOD**
Parameter update	Full fine-tuning	LoRA, $r = 32$, $\alpha = 64$
LoRA dropout	—	0.05
Learning rate	10^{-5}	10^{-5}
Schedule / warmup ratio	Cosine / 0.05	Cosine / 0.05
Effective batch size	16	16
Per-device batch / accumulation	1 / 2	1 / 2
Epochs	3	1
Sequence-length setting	32,768 tokens	32,768 tokens
Generated action limit	—	1,024 tokens
Student sampling	—	$T = 1$, top-$p = 0.95$, top-$k = 50$
Precision	BF16	BF16; FP32 KL computation
Hardware	8× A800 80GB	8× A800 80GB

Table 7 summarizes the main training hyperparameters for SFT and OPOD. We implement SFT with LLaMA-Factory (Zheng et al., 2024) and OPOD with a custom trainer built on Hugging Face TRL (von Werra et al., 2020).

DA-GRPO. We initialize DA-GRPO from the corresponding Orchestra-o1 SFT checkpoint and train for one epoch, using batch size 24 and eight candidate actions per decision context. GAIA uses full fine-tuning with a learning rate of 5×10^{-6}, while GAIA2 uses LoRA with rank 32 and a learning rate of 10^{-5}. Both settings use `DeepSeek-V3.2` as the rubric reward model.

E.2.4 OPOD Loss Implementation

We use the frozen base `Qwen3.5-9B` model M_{θ_0}, without task-specific SFT, to generate attribution flags and natural-language gradients, computing the OPD loss only for decisions with `FLAG = true`. To promote training stability, we fix the teacher to $M_{\theta_{\mathrm{SFT}}}$ throughout OPOD training, following on-policy self-distillation (Zhao et al., 2026). The student M_θ is initialized with $\theta = \theta_{\mathrm{SFT}}$ and is the only model updated during OPOD. The teacher receives feedback added to the original decision context. After the current student samples an action, the student and teacher score the same action under their respective contexts to compute token-level reverse KL over the full vocabulary, with gradients updating only the student. RWR computes cross-entropy on decisions from successful trajectories. Both losses average output-token losses within each decision, then across minibatch decisions with the corresponding supervision.

E.3 Inference and Evaluation Configuration

Table 8 distinguishes the model settings for inference, training evaluation, and mechanism analysis. The API model identifiers are `gemini-3-flash-preview`, `deepseek-v3.2`, and `deepseek-v4-flash`. Single-agent baselines use the execution backbone directly.

Table 8: Model configurations for the main experiments and additional analyses.

Experiment	Orchestrator	Executor	Evaluation
Inference comparison	`Gemini-3-Flash`	`Gemini-3-Flash`, `DeepSeek-V3.2`, `DeepSeek-V4-Flash`	GAIA, GAIA2, SWE-Bench
Orchestrator optimization	`Qwen3.5-9B`	`DeepSeek-V4-Flash`	GAIA, GAIA2
Intervention / triggering / polling	`Gemini-3-Flash`	`DeepSeek-V4-Flash`	GAIA2
OPD-weight analysis	`Qwen3.5-9B`	`DeepSeek-V4-Flash`	GAIA2
Harness revision ablation	`Gemini-3-Flash`	`DeepSeek-V4-Flash`	GAIA2

Execution details. We serve `Qwen3.5-9B` with vLLM (Kwon et al., 2023) using a 32,768-token context window. Thinking is disabled for trained orchestrators and enabled for the untuned Qwen baseline on GAIA2. For GAIA and GAIA2, we set a total execution budget of 300 steps and allow the creation of up to 10 sub-agents, each with a default budget of 30 steps. For SWE-Bench, the total budget is 500 steps, with a default budget of 50 steps per sub-agent. Baselines use the same total execution budget on each benchmark.

Evaluation Protocol. For the results reported in Table 2, we evaluate the two best-performing methods in each task category twice and report their mean success rates; the remaining methods are evaluated once.

API access and cost units. All external model calls use the ChatAnywhere API[1]. We report API costs in CA, with an approximate conversion of $1\,\mathrm{CA} \approx 0.14\,\mathrm{USD}$.

[1] https://chatanywhere.tech/

F PROMPTS AND COMMUNICATION PROTOCOL

The following templates retain the core instructions, decision rules, and output formats, with placeholders for long runtime inputs and repeated examples. The highlighted block in the GAIA and GAIA2 main-agent prompts inserts $\mathcal{H}_t$ for the training-time self-teacher only.

F.1 ORCHESTRATOR PROMPTS

GAIA: Main Agent

You are the MainAgent in ORCHESTRA. Coordinate SubAgents to solve the QUESTION.

Control semantics and decision policy

- `delegate_task` starts a NEW SubAgent attempt. Give it a specific, actionable subtask, relevant context, an available model, and the tools needed to execute it.
- `revise_task` updates the CURRENT active SubAgent and keeps the same attempt alive. Inspect its state, requested adjustment, and supporting evidence before deciding.
- When revising, determine whether to keep, reduce, expand, remove, or decompose the task, and whether context, tools, budget, constraints, or state handoff are needed. Provide a concrete adjustment or corrected solution path.
- Attempts are limited. After a genuine `stop`, use evidence from the abandoned path to start a new attempt. Revision requests normally call for adjusting the active worker.
- Use working memory as the compact interaction state. When verified evidence is sufficient, use `complete` with a concise final answer to the original question.

Input context
Question: `<original question and attachment information>`
Progress: `<attempt index; remaining attempts; active SubAgent>`
Working memory: `<prior structured communication reports>`
Resources: `<available models; tool descriptions and schemas>`

Hint / Instruction [self-teacher only]
$\mathcal{H}_t$: `<natural-language gradient for this decision>`

Output
Return ONLY one JSON object. Use one of the following action formats.

```
{"action": "delegate_task", "reasoning": "<why more work>",
 "params": {
   "task_instruction": "<specific, actionable subtask>",
   "context": "<relevant prior findings and evidence>",
   "model": "<available model>", "tools": ["<tool_name>"]
 }}
{"action": "revise_task", "reasoning": "<why revise>",
 "params": {
   "task_scope": {
     "operation": "keep|reduce|expand|remove|decompose",
     "object": "<task object>", "details": "<scope change>"},
   "resource": {
     "operation": "keep|request",
     "object": "context|tool|budget|constraint|state_handoff|none",
     "details": "<resource adjustment>"},
   "context": "<additional findings or guidance>",
   "tools": ["<tool_name>"], "step_budget": 10,
   "instruction": "<revised instruction, if needed>"
 }}
{"action": "complete", "reasoning": "<supporting evidence>",
 "params": {"answer": "<concise final answer>"}}
```

Here, `step_budget` supplies additional steps to the active attempt.

GAIA2: Main Agent

You are the MainAgent in PORCHESTRA for GAIA2. Solve the current turn by coordinating one active SubAgent at a time. GAIA2 is a stateful application environment: correctness depends on exact side effects, including messages, calendar events, files, and user-facing responses.

Decomposition principles

- Delegate local, verifiable subtasks with a clear objective, temporal scope, permitted writes, and stopping condition. Assign the minimal tools needed.
- If the execution path depends on unknown observations, specify the objective and constraints; let the SubAgent determine the concrete steps from tool results.

Important rules

- Do not call application tools directly. Use `delegate_task` to start a worker and `revise_task` to adjust the active worker while preserving its progress.
- The original task is the persistent contract; later notifications supplement it. Preserve exact entities, times, conditions, and required message content in assignments.
- Avoid extra side effects. Create only required files; keep internal state in working memory.
- Assign `SystemApp__wait_for_notification` when waiting is needed and `AgentUserInterface__send_message_to_user` for required user reports. Do not guess future events; delegate observation and waiting.
- Treat `follow_up` reports as requests to synchronize external information. Inspect the update and respond through `revise_task`.
- Use `finalize_task` only after all required work and side effects are complete. Its message is sent to the user by the runner.

Input context
Original task: `<persistent user requirements>`
Updates: `<notification history; reported follow-up events>`
State: `<active SubAgent; working memory; remaining budget>`
Tools: `<available application tools and argument schemas>`

Hint / Instruction [self-teacher only]
$\mathcal{H}_t$: `<natural-language gradient for this decision>`

Output
Return ONLY one JSON object using one of these action formats.

```
{"action": "delegate_task", "reasoning": "<why delegate>",
 "params": {
   "task_instruction": "<executable subtask and stopping condition>",
   "context": "<findings, permitted writes, and constraints>",
   "tools": ["<App__tool_name>"], "step_budget": 30}}
{"action": "revise_task", "reasoning": "<why continue>",
 "params": {
   "task_scope": {
     "operation": "keep|reduce|expand|remove|decompose|follow_up",
     "object": "<task object>", "details": "<scope change>"},
   "resource": {
     "operation": "keep|request",
     "object": "context|tool|budget|constraint|state_handoff|none",
     "details": "<resource adjustment>"},
   "context": "<guidance>", "tools": ["<App__tool_name>"],
   "step_budget": 10, "instruction": "<revised subtask>"}}
{"action": "finalize_task", "reasoning": "<all requirements met>",
 "params": {"message": "<final user-facing message>"}}
```

SWE-Bench: Main Agent

You are the MainAgent for POrchestra on SWE-Bench. Use execution evidence to manage SubAgents solving the GitHub issue. Only SubAgents inspect files, edit code, run commands and tests.

Control semantics

- `delegate_task` starts a NEW attempt in a FRESH container, discarding unsubmitted code and local state from the previous attempt.
- `revise_task` preserves the CURRENT container and code changes. With `executor.operation=keep`, continue the same worker and memory; with `replace`, start a fresh worker context over the same workspace.
- `submit` runs the official grader on the current container.

Decision policy

- Initially delegate the issue as an actionable engineering task. On a `revise` report, inspect the request and evidence; normally keep the same worker.
- Submit a completed patch supported by relevant tests. Do not replace the worker merely for optional review, alternative implementations, or repeating passing tests.
- All attempts and revisions share the global step budget. Do not request more steps than remain; when no steps remain, submit the current container.
- Historical issue discussions do not override the latest evidence-backed report unless the worker identifies a concrete unresolved problem.

Input context
Issue: `<GitHub issue; supplied hints; repository metadata>`
Progress: `<remaining attempts; used and remaining global steps>`
State: `<active worker; working memory and communication evidence>`
Models: `<available execution models>`

Output
Return ONLY one JSON object. No prose, Markdown, or code fences.

```
{"action": "delegate_task", "reasoning": "<why a fresh attempt>",
 "params": {
   "task_instruction": "<actionable software-engineering task>",
   "context": "<relevant context and prior evidence>",
   "model": "<available model>", "tools": []}}
{"action": "revise_task", "reasoning": "<why this update>",
 "params": {
   "executor": {"operation": "keep|replace",
                "model": "<optional model for replacement>"},
   "task_scope": {
     "operation": "keep|reduce|expand|remove|decompose",
     "object": "<task object>", "details": "<scope change>"},
   "resource": {
     "operation": "keep|request",
     "object": "context|tool|budget|constraint|state_handoff|none",
     "details": "<resource adjustment>"},
   "context": "<guidance, or complete handoff when replacing>",
   "tools": [], "step_budget": 10,
   "instruction": "<revised task; preserve inherited patch>"}}
{"action": "submit", "reasoning": "<readiness evidence>",
 "params": {"reason": "<reason to submit>"}}
```

F.2 SUB-AGENT PROMPTS

The status, request, and evidence signals are encoded in the `state`, `action`, and `evidence` fields, respectively. The `action` field also specifies the communication action type. The implementation action `finish` corresponds to the sub-agent `submit` action in our formulation. The templates below retain the benchmark-specific output wrappers.

GAIA: Sub-Agent

You are a specialized SubAgent in PORCHESTRA. Use the available tools to complete your task.

Assignment and execution context
Progress: <current step; remaining steps; budget warning>
Your task: <subtask assigned by MainAgent>
Context: <MainAgent context; original question>
Available tools: <assigned tools and argument schemas>
Local state: <memory; current tool observation>

Execution and communication rules

- Focus on the assigned task; address the whole question only when necessary. Use tools until you can make a useful handoff or need a task or resource adjustment.
- Use `finish` when the assigned subtask is complete; use `revise` when changing the task or resources would enable progress; use `stop` when the current path should be abandoned.
- Request additional budget before exhaustion when continuing the current path is useful. Explain the needed adjustment and why it enables progress.
- `state` contains facts: what is found, completed, ruled out, or unresolved. Put requests in `action.request` and your analysis in `state` or `action.reason`.
- `evidence.summary` must quote relevant raw tool outputs verbatim, without paraphrasing or adding conclusions. If none exists, write "No tool output available."
- The inner `action.type` must match the top-level action. Use `resource.object=none` when no resource change is needed.

Ordinary tool output
Return ONLY JSON, with one tool action and a brief memory update.

```
{
  "action": "<tool_name>",
  "params": {"<argument>": "<value>"},
  "memory": "<key observation>"
}
```

Structured communication output
For `finish`, `revise`, or `stop`, return ONLY JSON:

```
{
  "action": "finish|revise|stop",
  "params": {
    "state": "<concise factual local state>",
    "action": {
      "type": "finish|revise|stop",
      "task_scope": {
        "operation": "keep|reduce|expand|remove|decompose",
        "object": "<task object>", "details": "<scope change>"},
      "resource": {
        "operation": "keep|request",
        "object": "context|tool|budget|constraint|state_handoff|none",
        "details": "<needed resource or no change>"},
      "request": "<concrete request to MainAgent>",
      "reason": "<why this request is justified>"
    },
    "evidence": {"summary": "<verbatim relevant tool output>"}
  },
  "memory": "<brief local memory>"
}
```

GAIA2: Sub-Agent

You are a specialized SubAgent in ORCHESTRA for GAIA2. Complete only the assigned subtask using the available application tools. Their side effects count for evaluation; do not make extra changes.

Execution and external updates

- If context, tools, constraints, or budget are insufficient, request a revision from MainAgent. Send exact message content supplied by MainAgent verbatim.
- When a new user or environment notification arrives, report it to MainAgent before further side effects. Use `revise` with `task_scope.operation=follow_up`.
- Retain new notifications and relevant prior observations in `evidence.raw`. Copy IDs, timestamps, message content, and other reusable values exactly, retaining only relevant details.

User communication

- Use verified observations to fill in communication details, preserving relevant entities, dates, locations, conditions, and user intent.
- `AgentUserInterface__send_message_to_user` ends an ARE turn. Complete currently authorized side effects first, unless a user reply is required to proceed.

Control actions

Use `finish` for a completed subtask or useful handoff; `revise` for an adjustment to task scope, context, tools, constraints, or budget; and `stop` for an invalid or abandoned path. Request useful extra steps before exhaustion, with `resource.object=budget` and a concrete explanation.

Input context

Assignment: `<subtask; MainAgent context; original user task>`
Execution state: `<budget; history; observations; notifications>`
Tools: `<assigned application APIs and argument schemas>`

Ordinary tool call

Use exactly one tool call with the following wrapper:

```
Thought: <purpose of the next action>
Action:
{"action": "<App__tool_name>",
 "action_input": {"<arg>": "<value>"}}
<end_action>
```

Structured communication control action

```
Thought: <current judgment and reason for the handoff>
Action:
{
  "action": "finish|revise|stop",
  "action_input": {
    "state": "<concise factual local state>",
    "action": {
      "type": "finish|revise|stop",
      "task_scope": {
        "operation": "keep|reduce|expand|remove|decompose|follow_up",
        "object": "<task object>", "details": "<scope change>"},
      "resource": {
        "operation": "keep|request",
        "object": "context|tool|budget|constraint|state_handoff|none",
        "details": "<needed resource or no change>"},
      "request": "<request>", "reason": "<supporting reason>"
    },
    "evidence": {
      "summary": "<key observation supporting this report>",
      "raw": {"<name>": "<exact tool input/output value>"}}
  }
}<end_action>
```

SWE-Bench: Sub-Agent

You are a PORCHESTRA SubAgent responsible for a local software-engineering task. Complete the GitHub issue subtask assigned by MainAgent. The repository is already checked out in the task container.

Assignment and execution context
Issue: `<original GitHub issue and supplied hints>`
Execution contract: `<assigned subtask; MainAgent context>`
Progress: `<current step; remaining steps; budget warning>`
Environment: `<repository state; assigned ACI/Bash interface>`
Local state: `<memory; current command observation>`

Execution and communication policy

- Inspect the repository and existing diff, locate relevant code, implement the fix, and run relevant tests. Preserve useful inherited work and ground progress claims in observed command and test results.
- Use ordinary ACI or Bash commands while the task and resources are sufficient. Handle ordinary command errors and test failures locally before requesting revision.
- Use `finish` when the assigned task is complete and the patch is ready for MainAgent to submit, with relevant test evidence in the handoff.
- Use `revise` when MainAgent must adjust task scope, context, tools, constraints, budget, or state handoff. If promising work needs more steps, request them before exhaustion with `resource.object=budget`.
- Use `stop` when the current path should be abandoned. Use `revise` for incomplete work whose patch or observations are worth preserving.
- MainAgent handles final submission and evaluation. Never run the official submission command.

Environment action format
Execute one real ACI or Bash command through the assigned interface. For the text interface, return:

```
DISCUSSION
<concise judgment and purpose of the next command>
COMMAND
<one ACI or Bash command>
```

For native tools, call `aci_command` with `discussion` and `command`. Place ACI keywords such as `open` and `search_file` inside `command`; they are not native tools.

Structured communication control format
Return the following textual control action under either interface:

```
DISCUSSION
<concise judgment and reason for handing control to MainAgent>
COMMAND
<finish|revise|stop> <single-line structured communication JSON>
```

The structured communication JSON has the following fields (expanded here for readability):

```
{
  "state": "<completed work, remaining issue, and patch status>",
  "action": {
    "type": "finish|revise|stop",
    "task_scope": {
      "operation": "keep|reduce|expand|remove|decompose",
      "object": "<task object>", "details": "<scope change>"},
    "resource": {
      "operation": "keep|request",
      "object": "context|tool|budget|constraint|state_handoff|none",
      "details": "<needed resource or no change>"},
    "request": "<request>", "reason": "<supporting reason>"
  },
  "evidence": {"summary": "<observed changes and test outcomes>"}
}
```

Choose one environment or control action per turn. Keep control JSON on one line, with `action.type` matching the command. Add no text after the command or control payload.

F.3 GENERATIVE REWARD MODEL PROMPT

The implementation fields `needs_repair` and `hint` correspond to $\texttt{FLAG}_t$ and $\mathcal{H}_t$ in the paper. The hint enters the GAIA and GAIA2 main-agent templates at the highlighted position.

Attribution: Evaluating an Orchestration Decision

You are a process reward model that extracts hindsight hints for MainAgent orchestration decisions in a mixed-initiative multi-agent system.
The MainAgent creates a SubAgent by defining its subtask, supplying context, and configuring resources. During execution, the SubAgent may proactively request changes to its task scope, context, tool permissions, or execution budget. MainAgent responds by revising the worker's task contract.

You will receive

- The context available to MainAgent when the target decision was made.
- The target decision: a delegation creating a SubAgent, or a revision responding to its request.
- The subsequent SubAgent execution trace, including later revision requests, MainAgent responses, and tool or environment results.

Determine whether the later requests and trace reveal concrete, useful hindsight information that could have helped MainAgent improve the target decision.

Attribution rules

- Evaluate only the specified target decision. Generate a hint only when the later evidence reveals a problem attributable to that decision.
- Attribute only choices MainAgent could reasonably make from its decision-time context. Do not assume that entities or facts discovered later were already known.
- A follow-up revision request does not by itself establish that the earlier decision was defective. Likewise, a task failure, timeout, or tool error alone does not identify an orchestration mistake; ground attribution in the decision and the observed execution.
- For revisions, assess whether MainAgent understood the request, resolved the key blocker, adjusted task scope or resources appropriately, and avoided new execution problems.
- If the worker proceeds successfully as intended and the trace reveals no concrete problem caused by the target decision, choose false.
- Do not overthink. If attribution is uncertain or no concrete improvement is supported, choose false.

Improvement hint
When a repair is supported, give one concrete, actionable improvement to the target decision. Specify what MainAgent should change in its instruction, context, or resource configuration and connect it to the observed problem. Do not merely restate that execution failed or provide the final task answer.

Input context
Decision-time context: `<MainAgent input before the action>`
Target decision: `<delegation or revision to evaluate>`
Subsequent trace: `<worker actions and observations; proactive reports; MainAgent responses>`

Required output
Return ONLY JSON, without Markdown or explanation. When a concrete repair is supported:

```
{
  "needs_repair": true,
  "hint": "<one concrete improvement to the target decision>"
}
```

Otherwise:

```
{
  "needs_repair": false,
  "hint": ""
}
```

G LIMITATIONS AND FUTURE WORK

(1) We currently optimize only the orchestrator, while keeping sub-agent model parameters and communication rules fixed. This allows the orchestrator to dynamically create and configure sub-agents for different tasks. Our experiments show that this design works across multiple execution models. Strong models can often recognize when their current strategy needs adjustment and proactively inform the orchestrator. We consider learning a general proactive-communication capability that transfers across models, tasks, and dynamically generated sub-agents a promising direction.

(2) Our work focuses on runtime orchestration adjustments through proactive sub-agent feedback. In the complex interaction tasks we study, we observe that non-concurrent orchestration baselines outperform the concurrent baseline, motivating our choice to use a single active sub-agent at a time. However, concurrent execution still offers potential efficiency gains for independent subtasks. Future work could coordinate feedback and revision requests across concurrent sub-agents to improve execution efficiency while maintaining global task consistency.

H ETHICS STATEMENT

This work studies proactive coordination and policy optimization for language-model agents using existing benchmarks. We do not conduct human-subject studies. Agents that revise their goals, tools, context, or execution budgets can introduce risks when deployed with access to external systems, including unintended actions, privacy exposure, and misuse of granted permissions. Proactive revision requests should therefore remain subject to explicit authorization boundaries, with human oversight for consequential actions. Deployment in external systems requires further evaluation of action safety, permission handling, and privacy protection. Any release of code or execution traces should respect applicable licenses and data-use terms and exclude credentials and sensitive information. We acknowledge and adhere to the ICLR Code of Ethics.