COLLATOR: COMPOSITIONAL MULTI-AGENT ORCHESTRATION WITH COUNTERFACTUAL REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) provide a flexible foundation for multi-agent systems, but their effectiveness and computational cost depend critically on orchestration design. Across different tasks, role design, capacity assignment, and dependency construction jointly affect both solution quality and execution efficiency. Existing approaches automate parts of this design process, yet they often optimize these decisions partially or sequentially, and rely on execution-level feedback that provides limited credit assignment for local orchestration decisions. We propose COLLATOR (Compositional Multi-Agent Orchestration with Counterfactual Reinforcement Learning), an LLM-based orchestrator that learns to design efficient multi-agent orchestration. Given a task, COLLATOR designs a unified orchestration specification that composes customized agent duties, capacity levels, and dependency relations. To train the orchestrator, we augment the orchestration-level Group Relative Policy Optimization (GRPO) objective with a localized counterfactual credit signal that edits role, capacity, or dependency fields and applies the resulting reward contrast only to the edited spans. Experiments on six reasoning and coding benchmarks, including MMLU, GSM8K, AQuA, MultiArith, SVAMP, and HumanEval, show that COLLATOR achieves the best average performance among evaluated multi-agent orchestration methods while improving the accuracy–token trade-off. Our code is available at `https://anonymous.4open.science/r/LEMON-B23C`.

1 INTRODUCTION

Recent advances in large language models (LLMs) have driven growing interest in LLM-based multi-agent systems (MAS) Wu et al. (2024); Li et al. (2023); Hong et al. (2023). By coordinating multiple agents, these systems can leverage tool use, decomposition, verification, and collaborative reasoning to tackle tasks that are difficult for a single agent. Yao et al. (2022); Shinn et al. (2023) However, their effectiveness depends not only on the capability of individual agents but also on how the system is orchestrated Guo et al. (2024). In particular, role specialization, capacity assignment, and information flow jointly determine how well the agents collaborate. Zhuge et al. (2024); Zhang et al. (2024a) A fixed orchestration design is hard to reuse across tasks: the same design can waste computation on simple queries, be too weak for harder ones, and become mismatched when the required information flow changes Du et al. (2024); Yao et al. (2023). Together, these constraints make hand-designed, task-specific orchestration difficult to scale, and point to the need for automatic methods that adapt orchestration to each task.

Recent work has begun to move MAS orchestration beyond fixed manual designs by adapting different parts of the orchestration. Topology-learning methods Zhuge et al. (2024); Zhang et al. (2024b;a) optimize information-flow structures, but usually over a specified set of agents or role profiles. Workflow-orchestration methods learn executable procedures for decomposing tasks, invoking workers, and routing intermediate context Zhang et al. (2024c); Nielsen et al. (2025); Fan et al. (2024), while largely inheriting worker definitions and model capacity assignment from the available workflow space. Routing and team-selection methods further adapt role assignment, architecture search, or model routing Zhang et al. (2025); Cai et al. (2025), but often connect these decisions through staged or cascaded procedures rather than one unified orchestration specification. Despite

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.

these methods making MAS orchestration more learnable and adaptive, this field remains in its infancy.

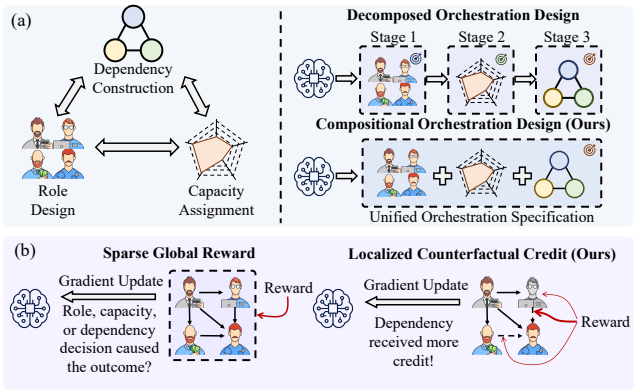


Figure 1: Compared with decomposed orchestration design and sparse global reward, COLLATOR composes role, capacity, and dependency fields within a unified orchestration specification and localizes counterfactual credit to the edited field.

The main limitation is that existing methods often optimize only part of the orchestration design space at the level of individual design components, rather than designing a unified orchestration specification that defines the multi-agent system compositionally Zhuge et al. (2024); Zhang et al. (2024b;a;c; 2025); Cai et al. (2025). As illustrated in Figure 1(a), role design, capacity assignment, and dependency construction are coupled orchestration components, yet prior methods often determine them through decomposed stage-wise decisions Zhang et al. (2024c); Nielsen et al. (2025); Fan et al. (2024); Zhang et al. (2025). Such decomposition separates task-specific agent duties, capacity levels, and dependency relations from a single specification-level design process, even though these choices interact during execution and jointly determine the resulting solution quality and computational cost. This motivates the problem of compositional multi-agent orchestration, where the goal is to design a task-adaptive unified specification that treats these orchestration fields as mutually dependent.

Given this formulation, learning compositional orchestration specifications introduces two challenges: **(i) Resource efficiency.** The orchestrator must decide how much computation each task deserves, since a more elaborate specification is not always a better one. Introducing more agents or dependencies can improve robustness, but it also raises two coupled costs: higher worker-token usage from running more agents, and higher structural cost, i.e., the overhead of a more complex orchestration structure Qian et al. (2025); Zhang et al. (2024a); Wang et al. (2025). In particular, denser dependencies increase inter-agent communication, so more intermediate outputs are passed as context and even more worker tokens are consumed Zhang et al. (2024a); Wang et al. (2025). An effective specification should therefore add computation only when the task actually benefits from it; **(ii) Sparse credit assignment.** Most training feedback for LLM-based orchestrators is derived from end-to-end execution outcomes Zhang et al. (2024c); Nielsen et al. (2025), such as task success or the overall utility of the executed specification. Such orchestration-level feedback can evaluate whether a specification succeeds as a whole, but it does not reveal which local role, capacity, or dependency decision is responsible for the outcome. As shown in Figure 1(b), a sparse global reward assigns only one coarse training signal to the whole specification, leaving it unclear whether the outcome should be attributed to a role specialization, a capacity choice, or a dependency relation.

To address these issues, we propose COLLATOR(Compositional Multi-Agent Orchestration with Counterfactual Reinforcement Learning), an LLM-based framework for learning efficient multi-agent orchestration. COLLATOR formulates MAS orchestration as **compositional orchestration design**: given an input task, the orchestrator produces a structured specification that composes task-specific agent duties, capacity levels, and information dependencies into a unified orchestration, thereby shifting learning from isolated component optimization, such as topology search, workflow construction, or routing, to specification-level orchestration design. To train the orchestrator, COLLATOR introduces **localized counterfactual reinforcement learning**. For each generated specification, COLLATOR edits one role, capacity, or dependency field to construct a counterfactual specification, executes it, and compares its execution reward with that of the original specification. Rather than exhaustively evaluating all possible local edits, COLLATOR uses sampled local counterfactuals to provide localized credit while keeping the additional training-time overhead manageable. The resulting reward contrast is mapped back only to the edited span, changing the granularity of credit assignment from the complete specification to the local orchestration field. Our contributions are summarized as follows:

- **Compositional orchestration design.** We formulate LLM-based MAS orchestration as designing a unified orchestration specification that composes task-specific role specifications, capacity levels, and dependency references, and can be directly validated, compiled, and executed.
- **Localized counterfactual reinforcement learning.** We address the sparse credit-assignment problem in orchestration learning by refining orchestration-level GRPO with a localized counterfactual credit signal. This signal converts execution-level reward differences into span-level supervision for the edited orchestration decision.
- **Empirical validation.** Experiments across reasoning and coding benchmarks show that COLLATOR improves over single-agent, fixed-structure MAS, topology-design, and adaptive workflow baselines, with ablations validating both compositional orchestration design and localized counterfactual supervision.

2 RELATED WORK

Multi-Agent Collaboration and Orchestration Design. Recent work has moved LLM-based multi-agent systems (MAS) beyond fixed manual designs by automating different components of orchestration. Topology-oriented methods learn or search communication structures, e.g., sparse communication, graph pruning, or task-adaptive graph generation, but usually operate over predefined agents or role profiles Zhuge et al. (2024); Zhang et al. (2024a;b); Li et al. (2026a;b). Workflow-oriented methods search or generate agentic procedures for task decomposition, worker invocation, and context routing, but typically rely on predefined worker interfaces, operators, or workflow primitives Hu et al. (2024); Zhang et al. (2024c); Nielsen et al. (2025); Su et al. (2026). Routing and team-construction methods adapt collaboration modes, agent sets, or heterogeneous LLM backbones, but often determine these choices through staged or cascaded controllers Zhang et al. (2025); Yue et al. (2025); Cai et al. (2025); Yao et al. (2026). Overall, these methods improve the adaptivity of multi-agent orchestration, but they typically optimize topology, workflow, routing, or model selection as separate components rather than composing agent duties, capacity levels, and dependency relations into a unified orchestration specification. In contrast, we formulate orchestration as compositional orchestration design, where agent duties, capacity levels, and dependency relations are specified together in the same orchestration specification that is directly validated, compiled, and executed.

Reinforcement Learning for Multi-Agent Orchestration. Reinforcement learning provides a natural mechanism for optimizing LLM-based MAS from execution feedback. Existing methods train policies with rewards defined over final task success, execution efficiency, preference alignment, model or tool usage, and topology complexity Nielsen et al. (2025); Su et al. (2025); Gao et al. (2025); Wang et al.; Jiang et al. (2026). Recent topology-focused work further uses group-relative optimization to reduce reward noise and assign finer-grained credit to communication edges Cang et al. (2026). Execution-level rewards alone do not identify the contribution of individual orchestration decisions. Our approach obtains a field-specific training signal by comparing specifications that differ in one duty, capacity, or dependency field, and applying the resulting reward contrast to the corresponding token spans.

3 METHOD

We present COLLATOR, an LLM-based framework for learning compositional and efficient multi-agent orchestration design. Given an input task, an LLM orchestrator designs a structured orchestration specification consisting of agent entries, where each entry specifies a task-specific role, an assigned capacity level, and dependency references to other agents. The specification is written as a YAML object; after validation against an execution schema, it is compiled into an orchestration graph and executed by worker agents. Training combines an orchestration-level reward computed after executing the generated specification with a localized counterfactual comparison over a locally edited specification. The orchestration reward evaluates the generated design as a whole, while the counterfactual comparison edits one design field, namely role, capacity, or dependency, and attributes the resulting reward contrast to the edited span. Together, these signals optimize the specification as a complete orchestration design while providing localized credit for individual design choices. Figure 2 illustrates the overall pipeline.

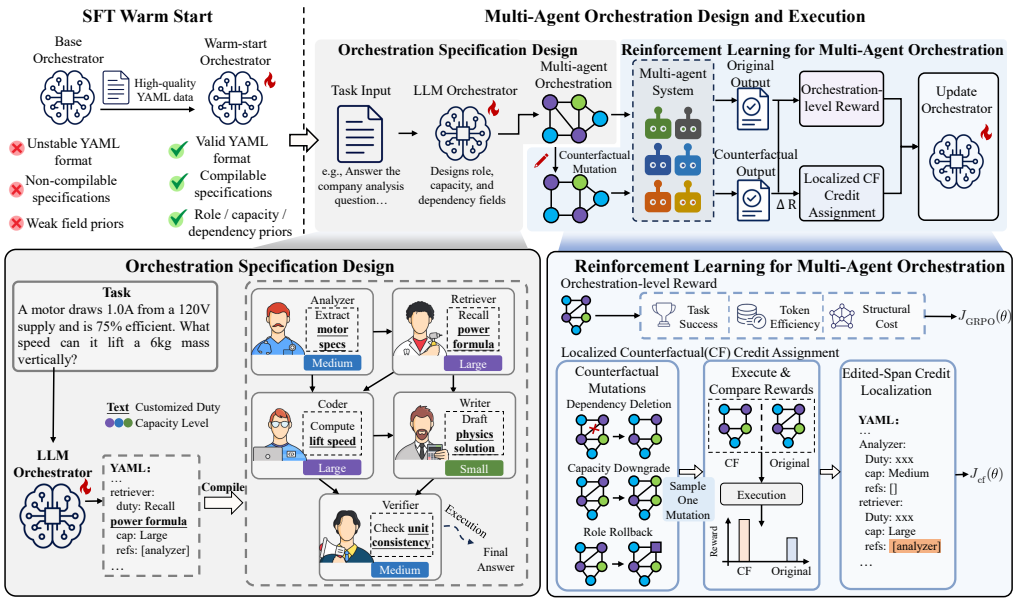


Figure 2: Overview of COLLATOR. The orchestrator designs a unified orchestration specification over roles, capacities, and dependencies. Training performs orchestration-level GRPO and localizes counterfactual reward contrasts to edited spans.

3.1 PROBLEM DEFINITION

Let $x \in \mathcal{X}$ be a task instance and let π_θ denote the orchestrator. Given x, the orchestrator generates a unified multi-agent specification

$$y \sim \pi_\theta(\cdot \mid x), \qquad y \in \mathcal{Y}, \tag{1}$$

where $\mathcal{Y}$ denotes the specification space. The generated specification y defines how a multi-agent system is instantiated and executed for the input task.

Executing a valid specification y produces a final answer $\hat{a}_y$ and an orchestration-level reward $R_{\text{orch}}(x, y)$. Our goal is to learn an orchestrator that generates task-specific multi-agent specifications with high orchestration reward. We next introduce the orchestration design space and the reinforcement learning objective used to optimize π_θ.

3.2 MULTI-AGENT ORCHESTRATION DESIGN

Given a task x, the orchestrator jointly specifies what each agent does, which worker capacity it uses, and which upstream outputs it receives. These choices are interdependent: the capacity needed to fulfill an agent's duty depends on both the assigned work and the information available from its dependencies. We therefore represent them within a single specification y generated by π_θ. We next formalize the resulting orchestration graph, describe how these components are represented in YAML, and introduce the rule-based validation applied before execution. Orchestration execution and learning are presented in Section 3.3.

Orchestration specification. For a valid specification y, the resulting multi-agent system is represented as a directed graph

$$G_y = (V_y, E_y), \tag{2}$$

where V_y is the set of instantiated agents and $E_y \subseteq V_y \times V_y$ is the set of dependency relations. Each agent $v \in V_y$ is specified by

$$v = (b_v, d_v, c_v), \qquad c_v \in \mathcal{L}, \tag{3}$$

where b_v denotes its base role, d_v denotes its task-specific duty, and c_v denotes its assigned capacity level from the available worker capacities $\mathcal{L}$. Each dependency $(u, v) \in E_y$ indicates that the output of agent u is provided as input context to agent v.

YAML specification. The specification y is written as a YAML object consisting of agent entries and dependency references. Each agent entry records its base role, task-specific duty, and capacity level, together with its dependency references to other agents. Agents in the same YAML step can execute concurrently, while references to agents in preceding steps determine how intermediate outputs are propagated across the orchestration.

Rule-based validation. Before execution, a rule-based parser checks whether the generated specification satisfies the YAML schema and execution constraints. Specifically, it verifies the required agent attributes, valid capacity values, unique agent identifiers, and references to agents in preceding steps. We denote the validation result by

$$\mathrm{Valid}(y) \in \{0, 1\}. \tag{4}$$

Only specifications satisfying $\mathrm{Valid}(y) = 1$ define G_y and are passed to worker execution. Because dependency references can only target agents in preceding steps, the resulting orchestration graph is directed and acyclic. Appendix A provides the complete YAML schema, validation rules, and an example specification.

3.3 Reinforcement Learning for Multi-Agent Orchestration

We initialize the orchestrator with supervised fine-tuning on executable specifications, then optimize it using orchestration-level GRPO and localized counterfactual supervision. Details of supervised initialization are provided in Appendix I.6. The two objectives are combined to train the orchestrator using both complete-specification rewards and localized field comparisons.

Orchestration execution. For a valid specification y, the worker agents execute G_y according to the dependency relations in E_y. Each agent $v \in V_y$ produces

$$o_v = f_{c_v}(x, b_v, d_v, \{o_u : (u, v) \in E_y\}), \tag{5}$$

where f_{c_v} is the worker model associated with capacity level c_v, and the outputs of agents on which v depends are provided as input context. The designated terminal agent returns the final answer $\hat{a}_y$. Let n_v^{in} and n_v^{out} denote the input- and output-token counts of worker v, respectively. We use the same capacity-weighted worker-token cost for reward computation and evaluation:

$$C(x, y) = \sum_{v \in V_y} \omega_{\mathrm{cap}}(c_v) \left(n_v^{\mathrm{in}} + n_v^{\mathrm{out}}\right), \tag{6}$$

where $\omega_{\mathrm{cap}}(c_v)$ weights both input and output tokens by the assigned worker capacity. The capacity weights and evaluation aggregation are specified in Appendix B.

Global orchestration reward. For a generated specification y, we define the orchestration reward as

$$R_{\mathrm{orch}}(x, y) = \begin{cases} R_{\mathrm{task}}(x, \hat{a}_y) + \lambda_{\mathrm{tok}} B_{\mathrm{tok}}(x, y) - \lambda_{\mathrm{str}} \Omega_{\mathrm{str}}(G_y), & \mathrm{Valid}(y) = 1, \\ r_{\mathrm{invalid}}, & \mathrm{Valid}(y) = 0, \end{cases} \tag{7}$$

where R_{task} measures task success, B_{tok} is the token-efficiency bonus, Ω_{str} is the structural cost, and $r_{\mathrm{invalid}} < 0$ penalizes invalid specifications. The coefficients λ_{tok} and λ_{str} control the contributions of the efficiency bonus and structural cost, respectively. We define the structural cost as

$$\Omega_{\mathrm{str}}(G_y) = \omega_V |V_y| + \omega_E |E_y| + \omega_H h(G_y), \tag{8}$$

where $h(G_y)$ is the depth of the orchestration graph, and ω_V, ω_E, and ω_H weight the costs of agents, dependencies, and graph depth.

To compute the token-efficiency bonus, we map the capacity-weighted worker-token cost $C(x, y)$ using task-specific lower and upper bounds, τ_x^- and τ_x^+, with $\tau_x^- < \tau_x^+$. These bounds remain fixed during RL and define the cost interval over which the efficiency bonus decreases. Costs at or below the lower bound receive the maximum efficiency bonus, whereas costs at or above the upper bound receive no efficiency bonus. Within this interval, the bonus decreases linearly as worker-token usage increases. The bounded token-efficiency bonus is

$$B_{\mathrm{tok}}(x, y) = g_y \left[\frac{\tau_x^+ - C(x, y)}{\tau_x^+ - \tau_x^-}\right]_0^1, \qquad g_y = \begin{cases} 1, & \hat{a}_y \text{ is correct}, \\ \rho, & y \text{ is valid but unsolved}, \end{cases} \tag{9}$$

where $[z]_0^1 = \min(1, \max(0, z))$ and $\rho \in [0, 1)$ is the reduced bonus coefficient for a valid but unsolved specification. Costs at or below τ_x^- receive the maximum efficiency bonus, whereas costs at or above τ_x^+ receive no efficiency bonus. The reward weights are reported in Appendix I.4.

For each training task $x \sim \mathcal{D}_{\text{train}}$, we sample K specifications from $\pi_{\theta_{\text{old}}}(\cdot \mid x)$ and compute their rewards $R_i = R_{\text{orch}}(x, y_i)$. We minimize the standard clipped GRPO loss Guo et al. (2025):

$$J_{\text{GRPO}}(\theta) = -\mathbb{E}_{x, y_{1:K}}\left[\frac{1}{K}\sum_{i=1}^{K}\frac{1}{|y_i|}\sum_{t=1}^{|y_i|}\ell_{\text{clip}}\Big(r_{i,t}(\theta), \widehat{A}_i\Big)\right], \tag{10}$$

where $\widehat{A}_i$ is the group-normalized reward, $r_{i,t}$ is the token-wise importance ratio, and ℓ_{clip} is the clipped surrogate defined in Appendix I.3. GRPO assigns the same advantage to all tokens in a specification, providing a global signal without distinguishing individual design fields.

Localized counterfactual credit assignment. To obtain credit for an individual orchestration field, we compare a factual specification y with a counterfactual specification that differs in exactly one field. We consider the counterfactual families $\mathcal{F}_{\text{cf}} = \{\varphi_{\text{dep}}, \varphi_{\text{role}}, \varphi_{\text{cap}}\}$, corresponding to dependency deletion, role rollback, and capacity downgrade, respectively. Dependency deletion removes one feasible upstream reference, role rollback replaces d_v with the generic duty associated with b_v, and capacity downgrade replaces c_v with the next lower capacity level.

Given a feasible counterfactual family $\varphi \in \mathcal{F}_{\text{cf}}$ and a valid field location ℓ, the edit operator constructs $\widetilde{y} = \mu_{\varphi,\ell}(y)$. Sampling the counterfactual family and field location induces the conditional distribution $q_{\text{cf}}(\cdot \mid y)$, from which $\widetilde{y} \sim q_{\text{cf}}(\cdot \mid y)$. Let S^{fact} and S^{cf} denote the token spans of the selected field values in y and $\widetilde{y}$, respectively. Mutation sampling and span construction are described in Appendix I.5.

We evaluate y and $\widetilde{y}$ using the same orchestration reward and compute

$$\Delta R = R_{\text{orch}}(x, y) - R_{\text{orch}}(x, \widetilde{y}). \tag{11}$$

The sign of ΔR indicates whether the factual or counterfactual field value produces the higher orchestration reward. Because the two specifications differ in only one field, their reward contrast provides localized credit for that field.

We use this contrast to favor the field value with the higher orchestration reward, applying the supervision only to the selected field-value spans. For a serialized specification z and a nonempty field span U, we define the average log-probability

$$s_\theta(x, z, U) = \frac{1}{|U|}\sum_{t \in U}\log \pi_\theta(z_t \mid x, z_{<t}). \tag{12}$$

We then set $s_\theta^{\text{fact}} = s_\theta(x, y, S^{\text{fact}})$ and $s_\theta^{\text{cf}} = s_\theta(x, \widetilde{y}, S^{\text{cf}})$. Tokens outside these spans remain conditioning context but do not contribute directly to the localized loss. We use the contrast weight

$$w_{\text{cf}}(\Delta R) = \frac{\min(|\Delta R|, \delta_{\max})}{\delta_{\max}}, \qquad \delta_{\max} > 0, \tag{13}$$

where $\delta_{\max}$ limits the influence of large reward contrasts. The localized counterfactual loss is

$$\begin{aligned} J_{\text{cf}}(\theta) = -\mathbb{E}_{x,y,\widetilde{y}}\Big[&\mathbf{1}\{|\Delta R| \geq \delta_{\min}\}\, w_{\text{cf}}(\Delta R) \\ &\cdot \log \text{sigmoid}\big(\beta_{\text{cf}}\, \text{sign}(\Delta R)\,\big[s_\theta^{\text{fact}} - s_\theta^{\text{cf}}\big]\big)\Big]. \end{aligned} \tag{14}$$

Here, $\text{sigmoid}(u) = (1 + e^{-u})^{-1}$, $\beta_{\text{cf}} > 0$ controls comparison sharpness, and $\delta_{\min} > 0$ filters negligible reward differences. The expectation is over valid sampled specifications and their feasible counterfactuals. Hyperparameter values are reported in Appendix I.5. A theoretical analysis of the localized counterfactual objective is provided in Appendix E.

Overall optimization. We minimize the overall training loss

$$J_{\text{total}}(\theta) = J_{\text{GRPO}}(\theta) + \lambda_{\text{cf}} J_{\text{cf}}(\theta), \tag{15}$$

where λ_{cf} controls the contribution of localized counterfactual credit assignment. The GRPO term optimizes complete orchestration specifications, whereas the counterfactual term provides localized supervision for the tested field. Counterfactual execution is used only during training, and unchanged worker calls can be reused through the execution cache described in Appendix H.

Table 1: Performance comparison (%) across six benchmarks. We report answer-option accuracy for MMLU and AQuA, exact-match accuracy for GSM8K, MultiArith, and SVAMP, and pass@1 for HumanEval. Avg. is the unweighted arithmetic mean of the six reported percentages.

Method	MMLU†	GSM8K	AQuA	MultiArith	SVAMP	HumanEval‡	Avg.
Single-Agent Reasoning							
Vanilla	74.46	85.59	46.46	76.00	80.80	80.64	73.99
CoT	66.49↓ 7.97	91.36↑ 5.77	81.89↑ 35.43	98.67↑ 22.67	93.60↑ 12.80	81.45↑ 0.81	85.58↑ 11.59
SC-CoT	67.54↓ 6.92	93.25↑ 7.66	82.68↑ 36.22	98.83↑ 22.83	94.40↑ 13.60	83.88↑ 3.24	86.76↑ 12.77
Fixed Multi-Agent Orchestration							
Chain-MAS	78.51↑ 4.05	92.19↑ 6.60	77.17↑ 30.71	98.83↑ 22.83	94.40↑ 13.60	80.65↑ 0.01	86.96↑ 12.97
Tree-MAS	79.56↑ 5.10	92.49↑ 6.90	82.68↑ 36.22	98.67↑ 22.67	93.90↑ 13.10	83.88↑ 3.24	88.53↑ 14.54
Complete-MAS	79.16↑ 4.70	92.65↑ 7.06	83.46↑ 37.00	98.83↑ 22.83	94.00↑ 13.20	82.25↑ 1.61	88.39↑ 14.40
Random-MAS	79.69↑ 5.23	92.19↑ 6.60	81.10↑ 34.64	98.67↑ 22.67	94.00↑ 13.20	83.06↑ 2.42	88.12↑ 14.13
Adaptive Workflow Orchestration							
AFlow	70.35↓ 4.11	91.05↑ 5.46	83.86↑ 37.40	96.33↑ 20.33	93.80↑ 13.00	76.61↓ 4.03	85.33↑ 11.34
Topology Design							
AgentDropout	80.73↑ 6.27	92.87↑ 7.28	83.07↑ 36.61	97.50↑ 21.50	93.10↑ 12.30	77.42↓ 3.22	87.45↑ 13.46
AgentPrune	80.54↑ 6.08	91.36↑ 5.77	83.86↑ 37.40	98.00↑ 22.00	95.20↑ 14.40	79.84↓ 0.80	88.13↑ 14.14
G-Designer	78.25↑ 3.79	92.17↑ 6.58	82.28↑ 35.82	98.17↑ 22.17	93.10↑ 12.30	71.77↓ 8.87	85.96↑ 11.97
ARG-Designer	**85.30**↑ 10.84	93.35↑ 7.76	69.29↑ 22.83	99.00↑ 23.00	94.90↑ 14.10	76.61↓ 4.03	86.41↑ 12.42
OFA-MAS	78.38↑ 3.92	84.84↓ 0.75	82.67↑ 36.21	98.67↑ 22.67	94.20↑ 13.40	79.84↓ 0.80	86.43↑ 12.44
Compositional Orchestration Design							
Ours	84.32↑ 9.86	**94.32**↑ 8.73	**86.22**↑ 39.76	**99.17**↑ 23.17	**95.60**↑ 14.80	**84.68**↑ 4.04	**90.72**↑ 16.73

†MMLU is evaluated on the validation set with 1,531 examples rather than the full test set. ‡HumanEval is evaluated on the held-out 124-example split for all methods; the remaining 40 examples are used only for training/adaptation by methods that require it. Full evaluation protocol is listed in Appendix I.7.

4 EXPERIMENTS

We evaluate COLLATOR across reasoning and coding tasks. We compare task performance against single-agent prompting, fixed-structure MAS, topology-design, and adaptive workflow baselines, and further analyze the accuracy–token trade-off of the learned orchestration. We also ablate supervised initialization, orchestration-level GRPO, role–capacity–dependency design components, and localized counterfactual mutation families, followed by case studies.

4.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate on six benchmarks: GSM8K Cobbe et al. (2021), MultiArith Roy & Roth (2015), SVAMP Patel et al. (2021), and AQuA Ling et al. (2017) for mathematical reasoning; MMLU Hendrycks et al. (2020) for multiple-choice reasoning; and HumanEval Chen et al. (2021) for code generation. For each task, the orchestrator generates a specification that is validated, compiled, and executed to produce the final answer. Evaluation metrics and dataset splits are detailed in Appendices B and I.7.

Baselines. We compare against four groups of baselines: single-agent prompting with Vanilla, CoT Wei et al. (2022), and SC-CoT Wang et al. (2022); fixed-structure MAS with Chain, Tree, Complete, and Random topologies; topology-design methods including AgentPrune Zhang et al. (2024a), AgentDropout Wang et al. (2025), G-Designer Zhang et al. (2024b), ARG-Designer Li et al. (2026a), and OFA-MAS Li et al. (2026b); and the adaptive workflow method AFlow Zhang et al. (2024c).

Performance. Table 1 compares performance across six benchmarks. COLLATOR achieves the highest average score of 90.72, outperforming SC-CoT, Tree-MAS, AgentPrune, and AFlow by 3.96, 2.19, 2.59, and 5.39 percentage points, respectively. It ranks first on five benchmarks and second on MMLU, where it trails ARG-Designer by only 0.98 points. These results demonstrate the effectiveness of jointly designing task-specific agent duties, capacity levels, and dependency relations within a unified orchestration specification.

Token Efficiency. Figure 3 compares task accuracy against capacity-weighted worker-token usage, using the cost $C(x, y)$ defined in Section 3.3 and averaged over evaluation tasks. COLLATOR lies on the Pareto frontier on all four benchmarks. On MMLU, it achieves competitive accuracy using approximately 0.2×10^4 worker tokens, compared with 0.3×10^4 for ARG-Designer and more than

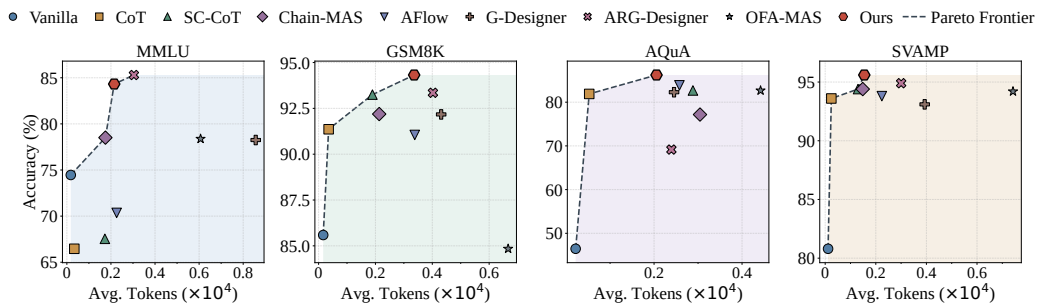


Figure 3: Accuracy–token trade-off on four benchmarks. Token usage is the average capacity-weighted sum of worker input and output tokens per task. Dashed lines connect methods on the Pareto frontier.

0.6×10^4 for several topology-design baselines. On GSM8K, AQuA, and SVAMP, it achieves the highest accuracy while using fewer worker tokens than several topology-design baselines. Together, these results show that task-adaptive capacity assignment and dependency construction improve the accuracy–token trade-off.

Ablation Study. Table 2 examines the training stages and orchestration design components on MMLU, GSM8K, and AQuA. SFT substantially improves specification validity, but SFT only remains below the strongest single-agent result on each task (Appendix D.1). Starting from the same SFT checkpoint, orchestration-level GRPO improves accuracy by 10.19 / 11.75 / 5.90 percentage points. Localized counterfactual credit assignment adds a further 1.24 / 0.92 / 0.79 points while reducing worker tokens by 15.5% / 10.2% / 31.7%. These results show that supervised initialization provides a foundation for executable generation, while execution-based RL learns more effective orchestration and counterfactual supervision further improves both accuracy and efficiency.

Table 2: Ablation studies on representative benchmarks.

Method	MMLU		GSM8K		AQuA	
	Acc.	Tok.↓	Acc.	Tok.↓	Acc.	Tok.↓
SFT only	72.89	1950.51	81.65	1564.01	79.53	1421.90
SFT + GRPO	83.08	2543.21	93.40	3742.59	85.43	3012.54
Ours	**84.32**	**2148.00**	**94.32**	**3362.00**	**86.22**	**2056.10**
w/o Role Design	82.43	2496.77	92.72	3734.76	83.07	2829.91
w/o Cap. Assignment	82.76	3401.06	93.18	4310.98	84.65	4154.38
w/o Dep. Construction	80.86	2347.71	93.33	3991.04	83.86	3296.94
w/o Dep. CF	83.54	2516.13	93.78	3797.33	85.43	3303.37
w/o Role CF	83.67	2484.87	93.93	3553.81	85.83	2680.48
w/o Cap. CF	84.00	2663.06	94.09	4362.87	85.83	3589.65
w/o SFT	48.53	2777.60	75.82	3964.21	64.96	3729.20

Removing any orchestration design component reduces accuracy. Dependency construction has the largest effect on MMLU (-3.46 points), while role design contributes most on AQuA (-3.15 points), highlighting the task-dependent importance of information flow and specialized duties. Capacity assignment improves both accuracy and efficiency: removing it more than doubles worker-token usage on AQuA, from 2056 to 4154, while reducing accuracy. Removing any counterfactual mutation family also lowers accuracy, supporting the complementary contributions of dependency-, role-, and capacity-level credit assignment.

Training Dynamics. Figure 5 shows the effect of localized counterfactual credit assignment on training. Shaded regions in panels (a) and (b) indicate ± 1 standard deviation across random seeds. Under otherwise identical training settings, the full method increases the final reward from approximately 2.0 to 2.25 compared with orchestration-level GRPO alone. This improvement involves additional counterfactual executions during training, whose overhead is reduced by node-level caching (Appendix H). In return, final worker-token usage decreases by approximately 11.1%, from 1.30K to 1.15K per execution. Panel (c) shows the absolute reward contrast $|\Delta R|$ for each mutation family: larger values indicate a stronger effect of the tested edits on orchestration reward. Dependency deletion yields the largest early contrasts, while role rollback and capacity downgrade provide sustained signals later in training. Panel (d) shows each mutation family's sampling probability, with the dashed $1/3$ line indicating uniform sampling. The probabilities vary around this reference, allocating more comparisons to families with larger recent reward contrasts while maintaining exploration across all three families.

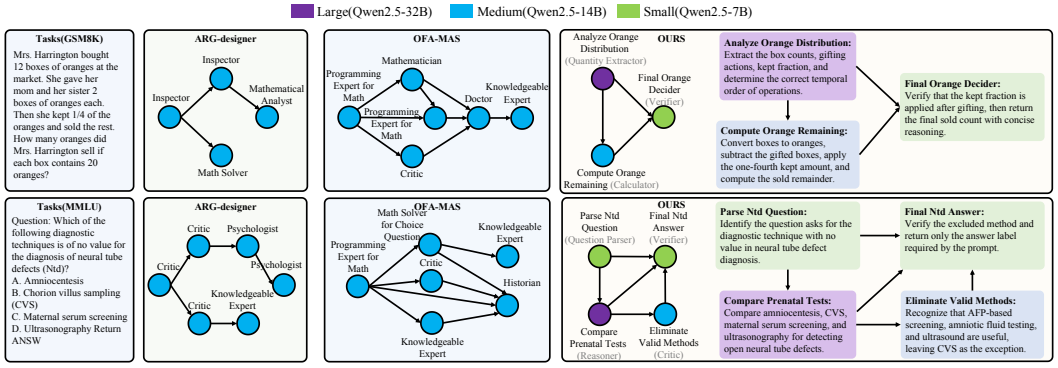


Figure 4: Case studies on GSM8K and MMLU. Node colors denote capacity levels, inherited base roles are shown as (Base Role), and the expanded views show COLLATOR's customized duties and dependency references.

Case Study. Figure 4 provides qualitative examples on GSM8K and MMLU. Compared with ARG-Designer and OFA-MAS, COLLATOR designs more task-aligned orchestration specifications rather than only adapting the graph topology. In the GSM8K example, the generated agents correspond to the main reasoning steps: analyzing the orange distribution, computing the remaining oranges, and verifying the final answer. In the MMLU example, COLLATOR decomposes the medical question into parsing the neural tube defect diagnosis target, comparing prenatal tests, identifying the exception among valid diagnostic methods, and producing the final answer. The expanded views show that customized duties and dependency references follow these reasoning steps, while capacity levels are assigned according to each agent's expected workload. In contrast, ARG-Designer and OFA-MAS use broader or repeated roles in these examples, such as *Inspector*, *Critic*, and *Knowledgeable Expert*, whose responsibilities are less directly tied to the task requirements. These cases illustrate the joint adaptation of role specifications, capacity levels, and dependency references to task-specific reasoning.

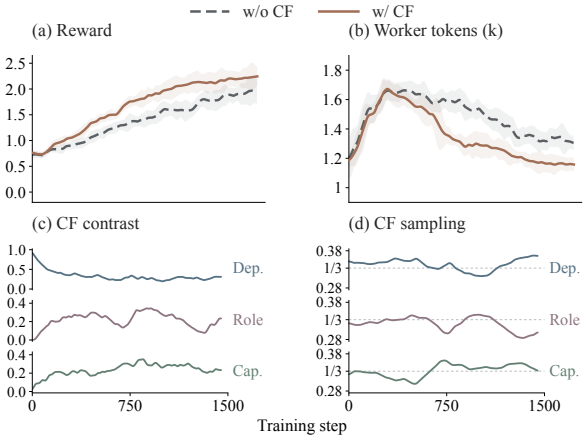


Figure 5: Training dynamics with and without localized counterfactual credit assignment in (a) orchestration reward and (b) worker tokens per execution, in thousands. For the full method, (c) shows absolute reward contrasts and (d) shows adaptive sampling probabilities for dependency deletion (Dep.), role rollback (Role), and capacity downgrade (Cap.).

5 CONCLUSION

We presented COLLATOR, a framework for learning compositional and efficient multi-agent orchestration. It jointly designs task-specific role specifications, capacity levels, and dependency references within a unified executable orchestration specification, adapting the design to each task. Training combines orchestration-level GRPO with localized counterfactual credit assignment, translating orchestration-reward contrasts into span-level supervision for edited fields. This allows the orchestrator to refine individual orchestration fields while optimizing complete specifications. Experiments across reasoning and coding benchmarks demonstrate strong task performance and token efficiency compared with single-agent prompting and existing multi-agent orchestration methods. Ablations and training-dynamics analyses further highlight the complementary roles of the orchestration design components and the benefits of localized counterfactual credit assignment.

AI USE STATEMENT

We used generative AI tools for language polishing, grammar correction, and improving the clarity of the manuscript, as well as for LaTeX formatting and consistency checks of notation and reported numerical summaries. The authors reviewed the AI-assisted revisions and take full responsibility for the accuracy and integrity of the final manuscript.

REPRODUCIBILITY STATEMENT

To support reproducibility, we provide the orchestration schema and an example specification in Appendix A, evaluation metrics in Appendix B, and prompt templates in Appendix C. Appendix I documents the model deployment, training hyperparameters, supervised warm start, and evaluation protocol, including dataset splits and evaluation sizes. The assumptions and derivation underlying our local counterfactual analysis are provided in Appendix E. Execution caching and computational resources are described in Appendices H and J, respectively. An anonymous code repository is linked in the abstract.

REFERENCES

Shuowei Cai, Yansong Ning, and Hao Liu. Agentbalance: Backbone-then-topology design for cost-effective multi-agent systems under budget constraints. *arXiv preprint arXiv:2512.11426*, 2025.

Yueyang Cang, Xiaoteng Zhang, Erlu Zhao, Zehua Ji, Yuhang Liu, Yuchen He, Zhiyuan Ning, Chen Yijun, Wenge Que, and Li Shi. Graph-grpo: Stabilizing multi-agent topology learning via group relative policy optimization. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 20222–20231, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *Forty-first international conference on machine learning*, 2024.

Shengda Fan, Xin Cong, Yuepeng Fu, Zhong Zhang, Shuyan Zhang, Yuanwei Liu, Yesai Wu, Yankai Lin, Zhiyuan Liu, and Maosong Sun. Workflowllm: Enhancing workflow orchestration capability of large language models. *arXiv preprint arXiv:2411.05451*, 2024.

Hongcheng Gao, Yue Liu, Yufei He, Longxu Dou, Chao Du, Zhijie Deng, Bryan Hooi, Min Lin, and Tianyu Pang. Flowreasoner: Reinforcing query-level meta-agents. *arXiv preprint arXiv:2504.15257*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*, 2024.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2023.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

Haoning Jiang, Han Wu, Zhuoli Ouyang, Ziheng Wang, Tinghuan Chen, and Junmin Jiang. Fd-magrpo: Functionality-driven multi-agent group relative policy optimization for analog-ldo sizing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 22310–22317, 2026.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in neural information processing systems*, 36:51991–52008, 2023.

Shiyuan Li, Yixin Liu, Qingsong Wen, Chengqi Zhang, and Shirui Pan. Assemble your crew: Automatic multi-agent communication topology design via autoregressive graph generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 23142–23150, 2026a.

Shiyuan Li, Yixin Liu, Yu Zheng, Mei Li, Quoc Viet Hung Nguyen, and Shirui Pan. Ofa-mas: One-for-all multi-agent system topology design based on mixture-of-experts graph generative models. In *Proceedings of the ACM Web Conference 2026*, pp. 1333–1344, 2026b.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 158–167, 2017.

Stefan Nielsen, Edoardo Cetin, Peter Schwendeman, Qi Sun, Jinglue Xu, and Yujin Tang. Learning to orchestrate agents in natural language with the conductor. *arXiv preprint arXiv:2512.04388*, 2025.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. Are nlp models really able to solve simple math word problems? In *Proceedings of the 2021 conference of the North American chapter of the association for computational linguistics: human language technologies*, pp. 2080–2094, 2021.

Chen Qian, Zihao Xie, Yifei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, et al. Scaling large language model-based multi-agent collaboration. In *International Conference on Learning Representations*, volume 2025, pp. 41488–41505, 2025.

Subhro Roy and Dan Roth. Solving general arithmetic word problems. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pp. 1743–1752, 2015.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Hongjin Su, Shizhe Diao, Ximing Lu, Mingjie Liu, Jiacheng Xu, Xin Dong, Yonggan Fu, Peter Belcak, Hanrong Ye, Hongxu Yin, et al. Toolorchestra: Elevating intelligence via efficient model and tool orchestration. *arXiv preprint arXiv:2511.21689*, 2025.

Jinwei Su, Qizhen Lan, Yinghui Xia, Lifan Sun, Weiyou Tian, Tianyu Shi, and Lewei He. Difficulty-aware agentic orchestration for query-specific multi-agent workflows. In *Proceedings of the ACM Web Conference 2026*, pp. 2060–2070, 2026.

Siyu Wang, Ruotian Lu, Zhihao Yang, Yuchao Wang, Lei Xu, Qimin Xu, Guojun Yin, Cailian Chen, Xinping Guan, et al. Topoweaver-r1: Reinforcing difficulty-aware topology evolution in multi-agent competition-level code generation.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Zhexuan Wang, Yutong Wang, Xuebo Liu, Liang Ding, Miao Zhang, Jie Liu, and Min Zhang. Agentdropout: Dynamic agent elimination for token-efficient and high-performance llm-based multi-agent collaboration. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 24013–24035, 2025.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First conference on language modeling*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Tianjun Yao, Zhaoyi Li, and Zhiqiang Shen. Hieramas: Optimizing intra-node llm mixtures and inter-node topology for multi-agent systems. *arXiv preprint arXiv:2602.20229*, 2026.

Yanwei Yue, Guibin Zhang, Boyang Liu, Guancheng Wan, Kun Wang, Dawei Cheng, and Yiyan Qi. Masrouter: Learning to route llms for multi-agent systems. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15549–15572, 2025.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Xu Yu, and Tianlong Chen. Cut the crap: An economical communication pipeline for llm-based multi-agent systems. *arXiv preprint arXiv:2410.02506*, 2024a.

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. *arXiv preprint arXiv:2410.11782*, 2024b.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. *arXiv preprint arXiv:2502.04180*, 2025.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*, 2024c.

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

A ORCHESTRATION SPECIFICATION AND VALIDATION

An orchestration specification contains an optional `defaults` block and an ordered list of `steps`, each containing agent entries. Each active agent has a unique task-specific identifier `type`, an inherited `base_role`, a customized `duty`, and dependency references `ref`. The runtime capacity is specified by `capacity_level`, either explicitly or through the applicable step-level or default setting. Its allowed values are `small`, `medium`, and `large`; their worker backbones are listed in Appendix I.1.

The validator checks the schema, required fields, unique active identifiers, legal base roles and capacity levels, and a valid final-answer role. References in the first step must be empty; later references must point to active agents in preceding steps. These constraints ensure that a valid specification compiles into a layered directed acyclic graph. Agents in the same step can execute concurrently. The allowed base-role interfaces are listed in Appendix F.

The example below illustrates the schema and information flow for a mathematical reasoning task. The duties are illustrative; in the experiments, the orchestrator generates task-specific duties for each input.

```
defaults:
  capacity_level: medium
steps:
  - agents:
      - type: extract_quantities
        base_role: math_analyst
        duty: Extract known quantities, unknown target, and explicit
  constraints.
        ref: []
        capacity_level: small

  - agents:
      - type: build_equations
        base_role: math_solver
        duty: Translate extracted quantities into arithmetic relations.
        ref: [extract_quantities]
        capacity_level: medium
      - type: check_units
        base_role: solution_inspector
        duty: Verify that each number is attached to the correct entity
  or unit.
        ref: [extract_quantities]
        capacity_level: small

  - agents:
      - type: compute_answer
        base_role: math_solver
        duty: Perform the arithmetic from the equation and unit check.
        ref: [build_equations, check_units]
        capacity_level: medium

  - agents:
      - type: verify_final_answer
        base_role: final_decider
        duty: Return the final answer after checking referenced
  arithmetic.
        ref: [compute_answer, check_units]
        capacity_level: medium
```

B EVALUATION METRICS

For MMLU and AQuA, we report answer-option accuracy after deterministic answer extraction. For GSM8K, MultiArith, and SVAMP, we report exact-match accuracy after normalizing the extracted

numerical answer. For HumanEval, we report pass@1 under unit-test execution. The average score in Table 1 is the unweighted arithmetic mean of the six reported percentages. All task metrics are computed using deterministic evaluators and do not rely on an LLM judge.

Worker-token usage is the average capacity-weighted sum of worker input and output tokens per task, using the cost $C(x, y)$ defined in Section 3.3. The capacity weights are

$$\omega_{\text{cap}}(c) = \begin{cases} 0.5, & c = \text{small (7B)}, \\ 1, & c = \text{medium (14B)}, \\ 2, & c = \text{large (32B)}. \end{cases} \tag{16}$$

For each evaluation instance x, let y_x denote the executed specification. For methods without a generated YAML specification, y_x denotes their instantiated worker configuration. The reported worker-token usage on evaluation dataset $\mathcal{D}_{\text{eval}}$ is

$$\overline{C}_{\mathcal{D}_{\text{eval}}} = \frac{1}{|\mathcal{D}_{\text{eval}}|} \sum_{x \in \mathcal{D}_{\text{eval}}} C(x, y_x). \tag{17}$$

C Prompts

C.1 Prompt Templates

We provide the prompt templates used in Collator for orchestration specification design, YAML repair, and role-conditioned worker execution. Bracketed fields such as `[TASK]`, `[ALLOWED_BASE_ROLES]`, and `[REFERENCED_OUTPUTS]` are filled by the runtime for each task instance.

The templates support configurable capacity-to-model mappings. In the reported experiments, `small`, `medium`, and `large` map to 7B, 14B, and 32B workers, respectively (Appendix I.1).

Orchestrator System Prompt

```
You are a strict topology planner for a multi-agent reasoning system.
Return exactly one YAML code block and nothing else.
Generate one executable graph, inline custom roles, and runtime
    capacity_level allocation in a single pass.
Do not directly answer the task. Your output is only the orchestration
    specification.

Schema must be: optional defaults -> steps -> agents.

Hard constraints (must satisfy all):
1) Allowed top-level keys are only: defaults, steps.
2) Allowed nested keys are only: capacity_level, agents, type,
    base_role, duty, ref, active.
3) Do not introduce any other keys, comments, metadata fields,
    explanations, or natural-language notes.
4) Every active agent.type must be a unique custom role id for this
    topology.
5) The agent.type must not be a fixed-pool base role name; it should
    describe the task-specific local function of the agent.
6) Every active agent.base_role must come from Allowed base roles in
    the user prompt.
7) Every active agent.duty must be non-empty, problem-specific, and
    describe the local contribution expected from the agent.
8) Avoid generic duties such as "solve the problem" or "analyze the
    task" unless they specify a concrete local output.
9) Step-0 ref must be [].
10) Any ref item must reference an active agent.type that has already
    appeared in previous steps.
11) A ref item must not refer to the current agent, a later agent, an
    inactive agent, or an undefined agent.
```

```
12) The final active step must include a custom final-answer agent
    using one of the final base roles in the user prompt.
13) Valid capacity_levels are exactly: small, medium, large.
14) Include explicit runtime capacity_level information using
    defaults.capacity_level, step capacity_level, agent capacity_level,
    or a combination of them.
15) Use small for cheap extraction or formatting nodes, medium for
    normal reasoning nodes, and large only when stronger reasoning is
    clearly justified.
16) Keep the graph executable and concise. Do not add redundant
    agents, unnecessary steps, or avoidable dependency references.
17) The YAML must be parseable and must preserve the required
    hierarchy of steps and agents.
18) Do not output prose, comments, analysis, or markdown outside the
    YAML block.
19) The first characters must be exactly ```yaml and the final
    characters must be exactly ```.
```

Orchestrator User Prompt Template

```
Task type: [TASK_TYPE]
Difficulty: [DIFFICULTY]
Role pool variant: [ROLE_POOL_VARIANT]
Allowed base roles: [ALLOWED_BASE_ROLES]
Allowed final base roles: [FINAL_BASE_ROLES]

Problem:
[TASK]

Generate one complete YAML that decides custom agent names, base-role
    inheritance, topology wiring, and runtime model capacity_levels in
    one pass.
Return exactly one fenced YAML block: ```yaml ... ```.

Required output contract:
- First non-empty key must be defaults: or steps:
- Allowed keys only: defaults, steps, capacity_level, agents, type,
    base_role, duty, ref, active
- Do not include any extra fields such as name, description, model,
    prompt, output, or rationale
- Every active agent.type is a unique custom role id, not a fixed-pool
    base role
- Every active agent.base_role is one of the allowed base roles
- Every active agent.duty is non-empty and describes the node's local
    contribution
- The duty should be specific to the input problem and should state
    the expected local output
- Valid capacity_levels are exactly: small, medium, large
- Use small for cheaper/simple nodes, medium for normal reasoning, and
    large only when stronger reasoning is justified
- step[0] refs must be empty
- refs must point to active agent.type values from earlier steps only
- refs must not point to undefined, inactive, duplicate, current-step,
    or future agents
- final active step must include a custom final-answer agent using one
    of the allowed final base roles
- Keep the graph executable and concise, but do not force a fixed step
    count
- Avoid redundant agents that perform the same local duty
- Avoid unnecessary all-to-all references when a smaller dependency
    set is sufficient
- no prose, no analysis, no comments, no extra fields outside the YAML
    block
```

```
small/medium/large are runtime compute capacity_levels.

Duty writing requirements:
- Use one concrete sentence per agent duty.
- Mention the local role intent and expected local output.
- Tie the duty to the given problem rather than writing a generic
    agent description.
- Avoid generic duties that do not identify what the node contributes.
- The final-answer duty should explicitly state the required benchmark
    answer format.

Example style:
Minimal topology example (schema only):
```yaml
defaults:
  capacity_level: medium
steps:
  - agents:
      - type: draft_agent
        base_role: [BASE_ROLE]
        duty: Extract the target information and produce a concise
    draft.
        ref: []
        capacity_level: medium
  - agents:
      - type: final_agent
        base_role: [FINAL_BASE_ROLE]
        duty: Check referenced work and return the benchmark answer
    format.
        ref: [draft_agent]
        capacity_level: medium
```

Now generate ONE valid full YAML for the target problem.
```

YAML Repair Prompt

```
You are a strict topology-and-budget YAML repairer.
Return exactly one YAML code block and nothing else.
Repair the provided full YAML so it satisfies inline custom-role,
    structure, ref, and capacity_level constraints.
Do not explain the fix.

Task type: [TASK_TYPE]
Difficulty: [DIFFICULTY]
Role pool variant: [ROLE_POOL_VARIANT]
Allowed base roles: [ALLOWED_BASE_ROLES]
Allowed final base roles: [FINAL_BASE_ROLES]

Problem:
[TASK]

Repair contract:
- Return exactly one fenced YAML block: ```yaml ... ```
- Repair inline custom-role fields, structure, refs, and
    capacity_level legality
- Preserve valid parts of the original YAML whenever possible
- Remove unsupported keys and illegal fields
- Every active agent must have a unique type, legal base_role,
    non-empty duty, ref, and capacity_level
- If an agent.type duplicates another active agent.type, rename it to
    a unique custom role id
```

```
- If an agent.base_role is not in the allowed base roles, replace it
    with the closest legal base role
- Rewrite short or generic duties so each one identifies the local
    output
- Ensure step[0] refs are empty
- Ensure all refs point only to active agent.type values introduced in
    earlier steps
- Remove refs to undefined, inactive, same-step, current, or future
    agents
- Ensure all capacity_level values are exactly one of: small, medium,
    large
- Add missing capacity_level fields using defaults, step-level, or
    agent-level capacity_level
- Use small for simple extraction or formatting nodes, medium for
    normal reasoning, and large only when clearly justified
- Ensure the final active step includes a custom final-answer agent
    using one of the allowed final base roles
- Keep the repaired graph executable and concise
- Do not add unnecessary steps, redundant agents, or avoidable
    dependency references

small/medium/large are runtime compute capacity_levels.

Duty writing requirements:
- Use one concrete sentence per agent duty.
- Mention the local role intent and expected local output.
- Avoid generic duties that do not identify what the node contributes.
- The final-answer duty should explicitly state the required benchmark
    answer format.

Invalid full YAML to repair:
[INVALID_YAML]

Repair feedback:
[VALIDATION_FEEDBACK]
```

Role-conditioned Worker Prompt Interface

```
Role-conditioned worker instructions are dispatched by base_role.
The worker receives the instantiated custom role, inherited base role,
    task-specific duty, task input, referenced upstream outputs, and
    optional feedback.

Task type: [TASK_TYPE]
Derived role: [TYPE]
Inherited base role: [BASE_ROLE]
Extra duty: [DUTY]
Assigned capacity_level: [CAPACITY_LEVEL]

Problem:
[TASK]

Referenced upstream outputs:
[REFERENCED_OUTPUTS]

Feedback:
[FEEDBACK]

Worker instruction:
Follow the inherited role category and the task-specific duty for this
    problem.
Use referenced upstream outputs when they are relevant to the assigned
    duty.
```

```
Do not repeat upstream reasoning unless verification or correction is
    required by the duty.
Do not solve unrelated parts of the task outside the assigned
    responsibility.
Produce a concise output that can be consumed by downstream agents.
If the duty requires verification, explicitly state whether the
    referenced answer is accepted or corrected.
If the duty requires finalization, return the answer in the
    benchmark-required format.
```

D Additional Experiments

D.1 Training-Stage Ablations

Table 3 compares the complete training pipeline with supervised initialization alone and reinforcement learning without supervised initialization. The SFT-only, SFT + GRPO, and full variants start from the same supervised checkpoint, isolating the gains from orchestration-level GRPO and localized counterfactual credit assignment. SFT only reaches 72.89% / 81.65% / 79.53% accuracy on MMLU / GSM8K / AQuA, below the strongest single-agent results of 74.46% / 93.25% / 82.68% in Table 1. Subsequent reinforcement learning improves task accuracy, and adding counterfactual supervision further reduces worker-token usage. RL without SFT improves over the initial specification generator but remains below the full method, supporting supervised initialization as a useful starting point for execution-based learning. Full-evaluation-set specification validity is reported in Appendix D.3.

Table 3: Complete training-stage comparison. Acc. denotes task accuracy (%); Tok. denotes average worker-token usage per task.

	MMLU		GSM8K		AQuA	
Setting	Acc.	Tok.	Acc.	Tok.	Acc.	Tok.
Before SFT, no RL	16.39	3443.61	46.24	3741.83	4.33	2902.22
RL without SFT	48.53	2777.60	75.82	3964.21	64.96	3729.20
SFT only	72.89	1950.51	81.65	1564.01	79.53	1421.90
SFT + GRPO	83.08	2543.21	93.40	3742.59	85.43	3012.54
SFT + GRPO + CF	84.32	2148.00	94.32	3362.00	86.22	2056.10

Orchestration design ablations. Table 4 defines the component ablations reported in Table 2. Each variant removes one type of task-adaptive orchestration decision. The counterfactual mutation ablations separately disable dependency deletion, role rollback, or capacity downgrade to examine the contribution of each localized credit signal.

Table 4: Definitions of the orchestration design ablations.

Variant	Modification
w/o Role Design	Replace each customized duty with the default duty associated with its inherited base role.
w/o Capacity Assignment	Assign the 14B worker to all LLM worker agents.
w/o Dependency Construction	Remove all inter-agent dependency references (`ref: []`), so workers receive no upstream agent outputs.

D.2 Sensitivity to Reward and Counterfactual Weights

The reward preference configuration was developed on GSM8K and then fixed for the other five main benchmarks, without benchmark-specific tuning. Table 5 varies one coefficient at a time while holding the others at their defaults: $\lambda_{\text{tok}} = 1.5$, $\lambda_{\text{str}} = 0.1$, and $\lambda_{\text{cf}} = 0.05$.

Increasing λ_{tok} from 0.75 to 1.5 reduces tokens by approximately 30–33% with accuracy changes of at most 0.66 points; increasing it to 3.0 trades further token savings for larger accuracy losses.

Table 5: One-factor-at-a-time sensitivity. Accuracy and average worker tokens are reported in the order MMLU / GSM8K / AQuA.

Coefficient	Value	Accuracy (%)	Worker tokens
λ_{tok}	0.75	84.98 / 94.69 / 86.22	3212.19 / 4801.30 / 2964.02
	1.5	84.32 / 94.32 / 86.22	2148.00 / 3362.00 / 2056.10
	3.0	72.24 / 86.20 / 77.56	1645.08 / 2454.70 / 1443.78
λ_{str}	0.05	84.25 / 93.93 / 85.82	2256.43 / 3585.36 / 2210.45
	0.1	84.32 / 94.32 / 86.22	2148.00 / 3362.00 / 2056.10
	0.2	80.27 / 92.64 / 83.46	1894.00 / 3204.30 / 1903.20
λ_{cf}	0	83.08 / 93.40 / 85.43	2543.21 / 3742.59 / 3012.54
	0.05	84.32 / 94.32 / 86.22	2148.00 / 3362.00 / 2056.10
	0.10	78.38 / 87.33 / 77.56	2643.23 / 3421.03 / 2490.54

Changing λ_{str} from 0.05 to 0.1 changes accuracy by at most 0.40 points while reducing token usage. The default counterfactual coefficient improves both accuracy and token efficiency over $\lambda_{\text{cf}} = 0$, whereas 0.10 degrades accuracy. These results illustrate how the reward weights control the accuracy–token trade-off and how the counterfactual weight balances localized supervision with orchestration-level GRPO.

Reward interpretation. Task correctness is evaluated externally, and cost is measured from worker execution and the compiled graph. The bounded efficiency bonus and the reduced bonus for unsolved specifications discourage cost reduction at the expense of correctness. We report task accuracy, worker-token usage, and specification validity separately to evaluate the learned orchestration beyond its shaped reward.

D.3 Serialization and Specification Validity

YAML and JSON can encode the same orchestration fields and dependencies. Table 6 compares the two representations on GSM8K. We train separate orchestrators for YAML and JSON. The accuracy difference is 0.24 percentage points, with similar worker-token usage and parse success, indicating that the effectiveness of the learned orchestration is preserved across these two serialization formats.

Table 6: Serialization comparison on GSM8K. Parse success is reported separately from task accuracy.

Representation	Accuracy (%)	Worker tokens	Parse success (%)
YAML	94.32	3362.00	99.70
JSON	94.08	3305.42	99.31

Specification validity requires a parseable, field-complete object with legal base roles and capacities, backward-only references, a valid terminal role, and successful DAG compilation. It does not imply task success: valid but ineffective orchestrations are evaluated using the same execution reward. Table 7 reports specification validity across the full evaluation sets, with rates of 99.40% or higher on all six benchmarks. Parse success in Table 6 checks whether an output can be parsed, whereas specification validity additionally requires the execution constraints above.

Table 7: Specification validity across the full evaluation sets.

Dataset	Evaluated	Valid	Rate (%)
MMLU (val)	1531	1527	99.74
GSM8K (test)	1319	1315	99.70
AQuA (test)	254	254	100.00
MultiArith	600	598	99.67
SVAMP	1000	994	99.40
HumanEval (held-out)	124	124	100.00

E Local First-Order Analysis of Counterfactual Learning

We analyze one fixed retained pair $(y, \widetilde{y})$ with observed reward contrast $|\Delta R| \geq \delta_{\min} > 0$. The sampled specifications, token spans, observed rewards, and contrast weight are held fixed during the update. Define the preferred score margin

$$m_\theta = \operatorname{sign}(\Delta R)\big(s_\theta^{\text{fact}} - s_\theta^{\text{cf}}\big), \qquad \ell_{\text{pair}}(\theta) = -w_{\text{cf}}(\Delta R) \log \operatorname{sigmoid}(\beta_{\text{cf}} m_\theta). \tag{18}$$

Local margin improvement. Assume m_θ is twice continuously differentiable in a neighborhood of the current parameters. For an isolated gradient-descent step $\theta^+ = \theta - \eta \nabla_\theta \ell_{\text{pair}}(\theta)$, with sufficiently small $\eta > 0$,

$$m_{\theta^+} - m_\theta = \eta a_\theta \|\nabla_\theta m_\theta\|_2^2 + O(\eta^2), \qquad a_\theta = w_{\text{cf}}(\Delta R)\beta_{\text{cf}} \operatorname{sigmoid}(-\beta_{\text{cf}} m_\theta) > 0. \tag{19}$$

Thus the first-order change is nonnegative and is strictly positive when the margin gradient is nonzero.

Derivation. Differentiating the pairwise loss gives $\nabla_\theta \ell_{\text{pair}} = -a_\theta \nabla_\theta m_\theta$. Substituting $\theta^+ - \theta = \eta a_\theta \nabla_\theta m_\theta$ into the first-order Taylor expansion of m_{θ^+} gives Eq. 19.

This result explains how the counterfactual loss favors the field value with the higher observed reward through a local update on the selected pair.

F Role Interfaces and Worker Execution

The fixed schema and runtime interfaces constrain executability, while the policy generates the YAML instance and its orchestration graph for each task. A generated `type` and `duty` specify a task-specific identity and responsibility; `base_role` selects an execution interface. Capacity chooses the worker backbone, and dependency references specify upstream context. Table 8 lists the domain-specific interface pools.

Table 8: Domain-specific base-role interfaces. Task-specific duties are generated rather than selected from this list.

Domain	Interfaces
GSM8K / MultiArith / SVAMP	`math_solver`, `math_analyst`, `python_solver`, `solution_inspector`, `final_decider`, `majority_voter`
AQuA	`choice_math_solver`, `choice_analyst`, `choice_python_solver`, `choice_inspector`, `choice_final_decider`, `majority_voter`
MMLU	`knowledge_expert`, `domain_specialist`, `option_analyst`, `knowledge_critic`, `choice_final_decider`, `majority_voter`
HumanEval	`code_planner`, `algorithm_designer`, `code_writer`, `test_designer`, `bug_fixer`, `final_code_writer`

Most interfaces instantiate an LLM worker; `majority_voter` performs deterministic aggregation. Tool availability is bound to compatible interfaces rather than freely selected through generated duties: `python_exec` serves `python_solver` and `choice_python_solver`; `unit_test_runner` serves `final_code_writer`; and `answer_parser` supports final-answer interfaces and voting. The `math_calc` interfaces are `math_solver`, `math_analyst`, `python_solver`, `choice_math_solver`, and `choice_python_solver`.

The runtime constructs each worker prompt from its inherited base role, task-specific duty, assigned capacity, task input, and referenced upstream outputs. The validated specification determines the resulting execution structure.

G Limitations

COLLATOR has two main limitations. First, training requires repeated worker execution to evaluate sampled specifications and their counterfactual variants. Node-level caching reduces redundant

computation, but execution-based reinforcement learning still incurs substantial training cost. Second, our evaluation focuses on text-based reasoning and coding benchmarks. Extending compositional orchestration to long-horizon tool use, multimodal inputs, and open-ended interactive environments remains an important direction for future work.

H Counterfactual Overhead and Execution Caching

Figure 6 compares recorded cumulative training-token usage with and without localized counterfactual credit assignment. Training-token usage is accumulated using the same capacity weights as the worker-token metric in Appendix B. The full method uses approximately 8.90M recorded tokens, compared with 7.33M for orchestration-level GRPO alone, an increase of 21.6%. Together with Figure 5, these results illustrate a training–execution trade-off: the additional training computation accompanies a higher final orchestration reward and an 11.1% reduction in worker-token usage per execution. Counterfactual comparisons therefore increase the training investment while helping the orchestrator learn more effective and token-efficient specifications.

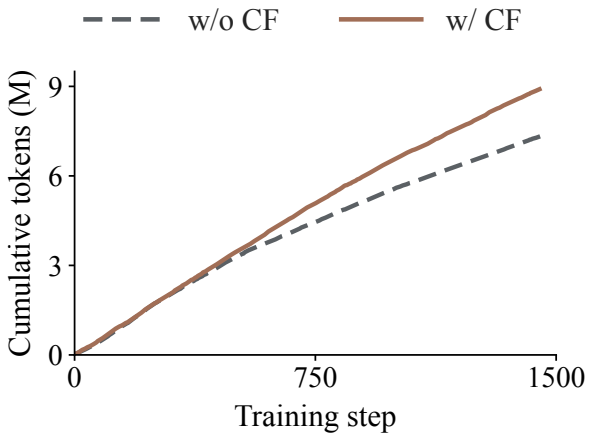


Figure 6: Recorded cumulative training-token usage with and without localized counterfactual credit assignment, in millions of tokens.

For each valid rollout, we sample at most one feasible mutation family and one edit location to construct a counterfactual specification. Evaluating this specification requires additional worker executions. To reduce redundant computation, we use node-level caching keyed by the input task, base role, customized duty, capacity level, and referenced upstream outputs. Worker outputs are reused whenever these inputs remain unchanged.

A local edit requires recomputation only for worker calls whose prompts or upstream contexts change. Dependency deletion affects the receiving agent and its downstream dependents, while role rollback and capacity downgrade affect the edited agent and its downstream dependents. Unrelated branches remain reusable, concentrating the additional execution on the affected portion of the orchestration graph.

Counterfactual evaluation is used only during reinforcement learning. At inference time, COLLATOR generates and executes one orchestration specification per task, so the learned improvements require no counterfactual comparisons during deployment.

I Reproducibility Details

I.1 Model Deployment

Table 9 reports the model backbones used by the orchestration policy and worker agents. The orchestration policy is initialized from a 7B instruction model and is trained to generate executable specifications. During execution, the generated capacity level selects one of three worker backbones.

The small, medium, and large capacity levels correspond to increasingly stronger Qwen2.5-Instruct models.

Table 9: Model deployment used in our experiments.

Role	Model
Orchestration policy backbone	`Qwen/Qwen2.5-7B-Instruct`
Small-capacity worker	`Qwen/Qwen2.5-7B-Instruct`
Medium-capacity worker	`Qwen/Qwen2.5-14B-Instruct`
Large-capacity worker	`Qwen/Qwen2.5-32B-Instruct`

This deployment allows the orchestrator to allocate different worker capacities within the same generated specification, rather than using a homogeneous worker model for all agents.

I.2 Main Training Hyperparameters

Table 10 summarizes the main optimization and generation hyperparameters used for reinforcement learning. We train for 1500 steps with at most 480 RL training samples per dataset. For each task, the policy samples a group of candidate orchestration specifications, executes valid candidates, and uses group-relative rewards for policy optimization. The prompt and generation length limits are set to 2048 tokens to allow the policy to output complete YAML specifications while keeping rollout cost bounded.

Table 10: Main training hyperparameters.

Hyperparameter	Value
Training-dynamics seeds	42, 1701, 67, 0
Training steps	1500
Max train samples	480
Split preset	`cross_dataset_v1`
Group size / generations	4
Prompts per reward call	4
Candidate rows per reward call	16
Learning rate	1×10^{-6}
LoRA	enabled
LoRA rank / alpha	8 / 32
Gradient checkpointing	enabled
Train batch size	4
Gradient accumulation	1
Max prompt tokens	2048
Max new tokens	2048
Temperature / top-p	0.6 / 0.9
Role pool variant	`baseline`
Node cache	enabled

Each reward call evaluates four tasks with four sampled specifications per task, yielding sixteen candidates. LoRA and gradient checkpointing reduce memory usage during policy optimization; execution caching is described in Appendix H.

Training-dynamics statistics. The training-dynamics comparison uses four random seeds: 42, 1701, 67, and 0. Shaded regions in Figure 5(a,b) indicate ± 1 standard deviation across these runs.

I.3 GRPO Objective

Let $\mathcal{D}_{\text{train}}$ denote the training dataset, from which tasks are sampled. For each task x, the rollout policy samples K specifications $y_i \sim \pi_{\theta_{\text{old}}}(\cdot \mid x)$, for $i = 1, \ldots, K$, and assigns each specification the reward $R_i = R_{\text{orch}}(x, y_i)$. Define the group mean, standard deviation, and normalized advantage

as

$$\overline{R}_x = \frac{1}{K}\sum_{i=1}^{K} R_i, \quad \sigma_x = \sqrt{\frac{1}{K}\sum_{i=1}^{K}(R_i - \overline{R}_x)^2}, \\ \widehat{A}_i = \frac{R_i - \overline{R}_x}{\sigma_x + \epsilon_{\text{adv}}}, \tag{20}$$

where $\epsilon_{\text{adv}} > 0$ ensures numerical stability. Following standard GRPO Guo et al. (2025), the token-wise importance ratio is

$$r_{i,t}(\theta) = \frac{\pi_\theta(y_{i,t} \mid x, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t} \mid x, y_{i,<t})}, \tag{21}$$

where $y_{i,t}$ is token t of the serialized specification y_i and $y_{i,<t}$ is its preceding token sequence. The clipped surrogate in Eq. 10 is

$$\ell_{\text{clip}}(r, A) = \min(rA, \text{clip}(r, 1-\epsilon_{\text{clip}}, 1+\epsilon_{\text{clip}})A)\,, \tag{22}$$

where ϵ_{clip} is the clipping coefficient.

I.4 Reward Hyperparameters

Table 11 reports the reward weights used throughout the main experiments. Appendix D.2 examines their effect on accuracy and worker-token usage.

Table 11: Reward hyperparameters.

Hyperparameter	Value
`efficiency_weight` (λ_{tok})	1.5
`structure_weight` (λ_{str})	0.1
`execution_weight`	1.0

I.5 Localized Counterfactual Training

Table 12 lists the settings for counterfactual sampling and optimization. Each pair changes one dependency, role, or capacity field, as defined in Section 3.3.

Table 12: Localized counterfactual training hyperparameters.

Hyperparameter	Value
Localized counterfactual objective	enabled
Mutation families	dependency deletion, role rollback, capacity downgrade
Dependency deletion count	1
Role rollback count	1
Capacity downgrade count	1
Sampling mode	`adaptive_rotation`
Reward contrast in the CF loss	Raw ΔR (Eq. 11)
EMA alpha	0.1
Rotation temperature	1.0
Minimum type probability	0.05
Counterfactual objective weight (λ_{cf})	0.05
Preference beta (β_{cf})	0.1
Delta cap ($\delta_{\max}$)	0.5
Minimum absolute delta ($\delta_{\min}$)	0.01

The mutation counts are set to one so that each counterfactual pair changes only one local orchestration field. This keeps the reward contrast attributable to a specific dependency, role, or capacity decision. The adaptive rotation sampler adjusts the probability of choosing each mutation family according to its recent reward contrasts, while the minimum type probability preserves exploration over all

mutation types. The delta cap and minimum absolute delta stabilize the counterfactual signal by clipping very large contrasts and filtering negligible reward differences. The counterfactual objective weight controls the contribution of localized counterfactual learning relative to the orchestration-level GRPO objective.

Field-span construction. The factual and counterfactual scoring spans correspond to the complete serialized values of the selected field. For dependency deletion, both spans include the brackets of the `ref` list: deleting its only reference compares `[A]` with `[]`. The empty list therefore retains a nonempty token span, so the average log-probability remains well-defined. Tokens outside the selected field values provide conditioning context but are excluded from the span score.

Reward contrast. The localized loss uses the raw orchestration-reward difference ΔR in Eq. 11. Its sign determines the preference direction, while $w_{\mathrm{cf}}(\Delta R)$ and $\delta_{\min}$ control the contrast weight and filtering condition, respectively.

I.6 Supervised Warm Start

We construct 500 supervised examples from questions randomly selected from the training set, using a 32B teacher to generate executable YAML specifications. These examples initialize the orchestrator's ability to generate role, capacity, and dependency fields before execution-based reinforcement learning. We fine-tune the orchestration policy with LoRA using LLaMA-Factory for 32 global steps and use the resulting checkpoint to initialize GRPO training. The training-stage comparison is reported in Appendix D.1.

I.7 Evaluation Protocol

For RL training, we sample at most 480 examples from the training split of each dataset when an official training split is available. For HumanEval, which does not provide a separate training split, we use 40 examples for RL training or task-level adaptation by methods that require it; all methods are evaluated on the remaining 124. For MMLU, we evaluate on the validation split because the official test split is substantially larger and more costly.

Evaluation is conducted on MMLU, GSM8K, AQuA, MultiArith, SVAMP, and HumanEval. For each evaluation instance, the orchestrator generates one executable orchestration specification. The validator checks schema validity and dependency validity, and the compiled orchestration graph is executed once to produce the final answer. Table 13 summarizes the evaluation split and task type for each benchmark.

Table 13: Evaluation datasets.

Dataset	Evaluation size	Task type
MMLU (val)	1531	multiple-choice reasoning
GSM8K (test)	1319	mathematical reasoning
AQuA (test)	254	mathematical reasoning
MultiArith	600	mathematical reasoning
SVAMP	1000	mathematical reasoning
HumanEval (held-out)	124	code generation

Evaluation metrics are defined in Appendix B. The 124 held-out HumanEval examples do not overlap with the 40 examples used for RL training or adaptation.

J Compute Resources

Table 14 reports the compute resources used for training and evaluation. The main RL run takes 24 h 50 min; the counterfactual execution and caching procedure is described in Appendix H.

Table 14: Compute resources used in our experiments.

Item	Value
Compute platform	Amazon EC2
Instance type	`g6e.12xlarge`
GPU type	NVIDIA L40S
Number of GPUs	4
GPU memory	4×48 GB
CPU type	AMD EPYC 7R13
vCPUs	48
System memory	384 GiB
Storage used	1000 GB
Main RL training time	24 h 50 min

K Broader Impacts

COLLATOR aims to improve the effectiveness and efficiency of LLM-based multi-agent orchestration. Its potential positive impacts include reducing redundant worker-model calls, improving the efficiency of multi-agent reasoning systems, and enabling more systematic construction of task-adaptive orchestration specifications. These properties may make multi-agent systems easier to study, compare, and deploy under controlled resource budgets. As with other LLM-based systems, practical deployment should still include task-specific validation and appropriate human oversight.

L Existing Assets and Licenses

We use existing datasets, baseline implementations, and LLM backbones for research evaluation. We cite the original papers or repositories for these assets and follow their licenses or terms of use. For assets whose license is not explicitly specified in the local snapshot, we report the license status as not found and use the asset only for research comparison without redistributing it. Table 15 summarizes the main existing assets used in this work.

Table 15: Existing assets used in this work.

Asset	Type	Source / Version	License / Terms
MMLU	Dataset	Hendrycks et al.; local snapshot	MIT License
GSM8K	Dataset	Cobbe et al.; local snapshot	MIT License
AQuA	Dataset	Ling et al.; local snapshot	Apache License 2.0
MultiArith	Dataset	Roy and Roth; local snapshot	Not explicitly specified in local snapshot; research evaluation only
SVAMP	Dataset	Patel et al.; local snapshot	MIT License
HumanEval	Dataset	Chen et al.; local snapshot	MIT License
AFlow	Baseline / Code	Original repository; local snapshot	MIT License
G-Designer	Baseline / Code	Original repository; local snapshot	License not found in local snapshot; research evaluation only
OFA-MAS	Baseline / Code	Original repository; local snapshot	License not found in local snapshot; research evaluation only
ARG-DESIGNER	Baseline / Code	Original repository; local snapshot	License not found in local snapshot; research evaluation only
Qwen2.5-7B-Instruct	Model	`Qwen/Qwen2.5-7B-Instruct`	Apache License 2.0
Qwen2.5-14B-Instruct	Model	`Qwen/Qwen2.5-14B-Instruct`	Apache License 2.0
Qwen2.5-32B-Instruct	Model	`Qwen/Qwen2.5-32B-Instruct`	Apache License 2.0