# UNITYMAS-O: OPTIMIZING USER-DEFINED LLM MULTI-AGENT WORKFLOWS WITH REINFORCEMENT LEARNING

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Large language model (LLM) multi-agent systems are usually assembled from prompts, tools, and hand-written control logic, while reinforcement learning (RL) is applied to one policy at a time. This mismatch makes it difficult to optimize intermediate roles, compare parameter-sharing choices, or reuse training infrastructure across workflows. We present **UnityMAS-O**, a general RL optimization framework that treats a user-defined multi-agent workflow as the unit of optimization. The framework exposes four first-class objects: logical roles, graph-structured trajectories, explicit role-to-model mappings, and role-, turn-, or trajectory-level rewards. A Ray-based controller executes arbitrary workflows and routes invocations to model-local worker groups built on vLLM rollout and PPO training. The same interface supports fully shared, partially shared, and fully separated parameters. We train and evaluate multiple workflows on retrieval-augmented QA and executable code tasks, demonstrating the effectiveness of UnityMAS-O across diverse multi-agent settings. We further explore multi-agent optimization under shared and independent parameter settings, examining their differences in task performance and training dynamics. UnityMAS-O provides a reusable framework for turning manually designed LLM multi-agent systems into trainable multi-agent policies.[1]

## 1 INTRODUCTION

LLMs increasingly solve complex tasks through workflows of planners, generators, critics, verifiers, and answerers rather than a single response (Li et al., 2023; Wu et al., 2023; Qian et al., 2024). Such decomposition makes intermediate states explicit and lets each role use a different prompt, tool, or stopping rule. In practice, however, most multi-agent systems are still optimized by manual prompt engineering and fixed orchestration. When RL is used, it is commonly attached to one model or one final response, leaving the interaction protocol and the credit assignment across roles outside the training interface.

This creates a systems and learning gap. The quality of a final answer can depend on whether upstream roles create useful intermediate states, whether intermediate outputs preserve task-relevant information, whether feedback-producing roles expose actionable signals, and whether downstream agents consume those signals. A framework that only sees a single policy trajectory cannot naturally express these dependencies. Conversely, multi-agent RL implementations are often tied to one benchmark-specific workflow, making it hard to compare shared and independent parameters or to reuse the runtime for a different task.

We address this gap with **UnityMAS-O**, a general optimization interface for LLM-based multi-agent systems. UnityMAS-O treats the complete workflow as the RL object while keeping the distinction between a *logical role* and a *physical model instance*. Users define the roles and graph, choose a mapping from roles to models, and provide reward functions. The framework records graph-structured trajectories, attributes each reward to the role invocation that produced the relevant action, and routes the resulting data to the worker group for the mapped model. A single workflow can therefore be trained with full sharing, partial sharing, or independent models without rewriting its control logic.

[1]The anonymous code is available at http://anonymous.4open.science/r/UnityMAS-O-8EAE.



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.

Our design has three goals. First, it makes the workflow graph and its intermediate states visible to optimization. Second, it makes role-to-model assignment an explicit experimental variable rather than an implementation detail. Third, it separates workflow control from model-local tensors so heterogeneous models and distributed resources can be trained under one coordinator. We implement these ideas with a Ray-based star topology and PPO-style updates compatible with modern LLM RL infrastructure (Moritz et al., 2018; Sheng et al., 2024; Schulman et al., 2017).

Across workflow instances spanning parallel retrieval, iterative search, aggregation, and feedback-driven generation, RL improves the same hand-specified workflows before and after training. Sharing several logical roles in one model retains most of the independent-model score on the simpler QA workflow.

**Contributions.** We make the following contributions:

- We formulate workflow-level RL for LLM multi-agent systems with explicit logical roles, graph trajectories, role-to-model mappings, and reward attribution.
- We present a Ray + `verl`-style runtime with a central controller and model-local worker groups for rollout, buffering, advantage computation, and PPO-style updates.
- We support full, partial, and independent parameter sharing under one interface, and demonstrate before/after gains on tool-augmented and reflective workflows against published baselines.

**Related work.** We build on single-agent reasoning/tool use, role-based orchestration, workflow-specific MARL, and general LLM–MARL optimizers (Yao et al., 2023; Press et al., 2023; Trivedi et al., 2023; Shinn et al., 2023; Li et al., 2023; Wu et al., 2023; Qian et al., 2024; Chen et al., 2025a;b; 2026b;c;a; Feng et al., 2026; Zhao et al., 2026; Zhang et al., 2026a;b). Unlike these lines, which expose different parts of a workflow to the learner, UnityMAS-O keeps the user DAG, role-to-model map, invocation buffer, and node/turn/trajectory credit configurable in one interface. Detailed descriptions and the modeling comparison appear in Appendices B and G.

## 2 UnityMAS-O

### 2.1 Workflow and role abstractions

We model every user-defined workflow as a finite directed acyclic graph (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. A node $v \in \mathcal{V}$ is either a trainable LLM role or a non-trainable tool/environment operation. Each node specifies its input state, prompt or tool schema, parser, and output state. An edge $(u, v) \in \mathcal{E}$ is a data/control dependency: $v$ becomes ready only after the required predecessors have completed. Consequently, an arbitrary finite workflow can express sequential chains, parallel branches, conditional paths, and joins while retaining an acyclic execution contract.

Some user workflows are described by bounded repetition in pseudocode. UnityMAS-O compiles such a loop by unrolling role copies $v^{(0)}, \ldots, v^{(T_{\max}-1)}$ in the episode DAG. A stop action, terminal predicate, or environment success signal masks the unused suffix. Repeated role types are therefore represented by stage-indexed nodes, while each sampled execution remains a finite DAG with a well-defined dependency partial order. This convention makes graph structure explicit without restricting the user to a single fixed chain.

For execution level $\ell$, the controller maintains the ready frontier $\mathcal{F}_\ell = \{v : \mathrm{Pred}(v) \subseteq \mathcal{D}_\ell,\ v \notin \mathcal{D}_\ell\}$, where $\mathcal{D}_\ell$ is the set of completed nodes and branch masks remove inactive successors. Nodes in the same frontier can run concurrently. For task $x$, the primary trajectory object is a structured graph trace rather than an ordered chain:

$$\mathcal{T}_x = \left(\mathcal{V}_x^{\mathrm{exec}}, \mathcal{E}_x, \{z_v\}_{v \in \mathcal{V}_x^{\mathrm{exec}}}\right), \qquad z_v = (s_v, a_v, o_v, r_v, d_{\mathcal{G}}(v)). \tag{1}$$

Here $\mathcal{V}_x^{\mathrm{exec}}$ is the set of realized nodes, $\mathcal{E}_x \subseteq \mathcal{E}$ is the active dependency relation, $s_v$ is the node input state, $a_v$ is an LLM output or tool request, $o_v$ is the parsed output or tool observation, and $r_v$ may be assigned immediately or after downstream nodes finish. The trace preserves the partial order and frontier masks, so concurrent nodes have no artificial precedence. A topological order can still be materialized when an algorithm needs a batch or an ordered view, but that order is a storage and computation view rather than the workflow semantics.

The trace and rollout buffer are two synchronized views of the same execution record. The graph-level buffer stores node states, active edges, frontier/stop decisions, observations, and rewards. Once the trainable-node set $\mathcal{V}_{\text{LLM}}$ and role-to-model map $\phi$ are selected, the PPO buffer for each physical model $m_i$ is the projection

$$\mathcal{B}_i(x) = \{b_v : v \in \mathcal{V}_x^{\text{exec}} \cap \mathcal{V}_{\text{LLM}},\ \phi(v) = m_i\}, \tag{2}$$

where $b_v$ retains the token-level rollout tensors together with the node identity, graph depth, reward, and references to predecessor outputs. In the implementation, graph metadata remains with the controller-side trajectory store while tensor-heavy entries are sharded by model; logically they form one invocation-indexed buffer. Tool-only nodes therefore remain in $\mathcal{T}_x$ but contribute no PPO entry.

The distinction is deliberate: $\mathcal{G}$ is the semantic object that defines execution dependencies, branch eligibility, and which downstream outcomes can be attributed to an invocation, whereas the PPO update consumes the materialized entries $b_v$. After predecessor context, stop masks, delayed rewards, and advantages have been resolved, a worker can update its model from token-level entries without traversing or tensorizing the DAG again. The graph is therefore essential to specifying and producing training data, but it is not an additional structural input required by the optimizer.

### 2.2 Role-to-model mappings

Let $\mathcal{V}_{\text{LLM}} \subseteq \mathcal{V}$ be the trainable LLM nodes, let $\mathcal{M} = \{m_1, \ldots, m_K\}$ be physical trainable models, and let

$$\phi : \mathcal{V}_{\text{LLM}} \to \mathcal{M} \tag{3}$$

map each trainable role instance to a physical model. An unrolled copy inherits its role-template mapping, $\phi(v^{(t)}) = \phi(\text{type}(v))$. The mapping gives three regimes without changing the workflow:

- **Full sharing:** $\phi(v) = m_1$ for all trainable roles. Role prompts and graph position distinguish invocations while parameters are shared.
- **Partial sharing:** related roles share a model and other role groups use different models.
- **Full separation:** each role has its own model, maximizing specialization at greater memory and compute cost.

For a trainable node $v$, the action is sampled from $a_v \sim \pi_{\theta_{\phi(v)}}(\cdot \mid s_v, v)$, so the role identity and graph context remain visible even when parameters are shared. The mapping is therefore a learning variable as well as a systems variable. Holding $\mathcal{G}$, prompt and parser schemas, tool interfaces, and reward contracts fixed, full sharing measures parameter efficiency and cross-role transfer; full separation measures the benefit of role specialization under higher resource cost; and partial sharing tests whether groups of related roles can share general capabilities while retaining specialization where it matters.

### 2.3 Role-aware rewards and objective

For each trainable node invocation $v$, UnityMAS-O exposes three complementary reward scales:

$$r_v = \lambda_{\text{node}} r_v^{\text{node}} + \lambda_{\text{turn}} r_v^{\text{turn}} + \lambda_{\text{traj}} r_v^{\text{traj}}. \tag{4}$$

The node-level term $r_v^{\text{node}}$ checks properties of the current invocation, such as a valid parser format, a required schema, or a task-specific rubric score for that node. The turn-level term $r_v^{\text{turn}}$ measures progress after a local execution stage or frontier, such as the change in an intermediate answer or environment score between two successive stages. The trajectory-level term $r_v^{\text{traj}}$ is a delayed outcome of the completed workflow, such as a final answer, verification, or executable-success score. The coefficients are workflow-configurable and may be zero, so the same interface supports dense local feedback, sparse final rewards, or mixtures of the three. The concrete workflows in Sec. 4 and App. F instantiate these terms with parser/format rewards, progress deltas, and task-level scores.

With $\Theta = \{\theta_m : m \in \mathcal{M}\}$, and an optional discount $\gamma \in (0, 1]$, the workflow objective is the expected weighted sum of these composite rewards over the trainable nodes that actually execute:

$$\max_{\Theta}\ \mathbb{E}_{x \sim \mathcal{D},\, \mathcal{T}_x \sim p_\Theta(\cdot \mid x, \mathcal{G}, \phi)} \left[ \sum_{v \in \mathcal{V}_x^{\text{exec}} \cap \mathcal{V}_{\text{LLM}}} w_{\mathcal{G}}(v) r_v \right], \qquad w_{\mathcal{G}}(v) = \gamma^{d_{\mathcal{G}}(v)} \text{ or } 1. \tag{5}$$

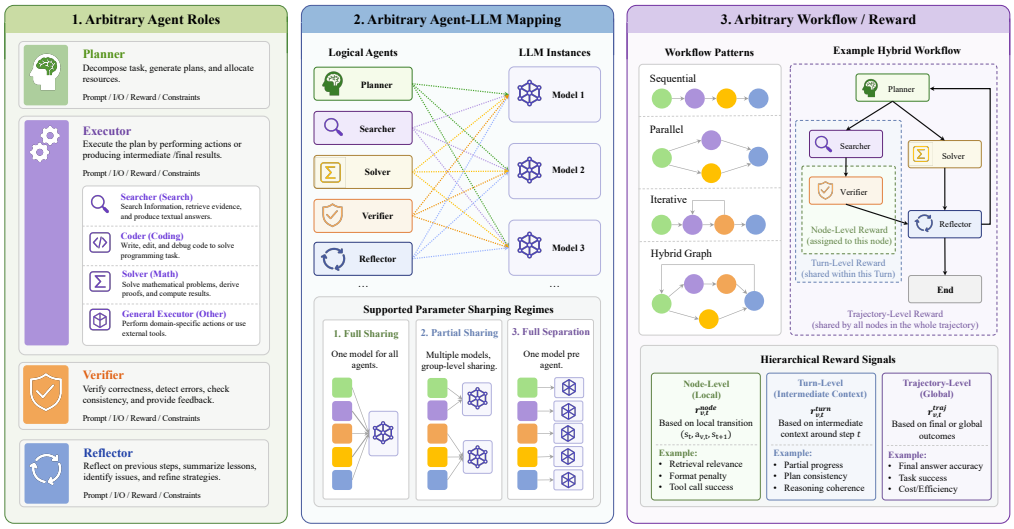


Figure 1: UnityMAS-O exposes four objects to the user: logical roles, a workflow graph, a role-to-model mapping, and role-aware rewards. These objects are independent of the physical worker layout.

The controller caches node outputs and any immediately available terms during asynchronous rollout. When a task rollout terminates, it finalizes each realized node's composite reward, including delayed turn- and trajectory-level terms, and writes that value back to the corresponding model buffer entry. Tool transitions remain in the graph trace but are excluded from the trainable-node sum. The optional graph-depth weight keeps any discount invariant to the arbitrary ordering of parallel nodes.

Figure 1 summarizes the four objects that remain fixed when we compare parameterizations: the logical roles, the DAG, the role-to-model map, and the reward contract. The system below turns these objects into asynchronous rollout records and model-local update batches.

## 3 System and Training

### 3.1 Star-topology runtime

UnityMAS-O has a central controller and one model-local worker group $W_i$ for each $m_i$. As shown in Figure 2, the controller owns workflow state, scheduling, tool and environment calls, stopping conditions, and reward commitment, while each worker group owns generation and optimization for one physical model. Ray provides remote execution and placement (Moritz et al., 2018). A worker group generates outputs for the trainable roles mapped to its model and owns the heavy tensors needed for training, including token log probabilities, masks, values, and rollout buffers. For a trainable LLM node $v \in \mathcal{V}_{\text{LLM}}$, the route is

$$(s_v, v) \longrightarrow W_{\phi(v)}. \tag{6}$$

Only lightweight outputs and metadata return to the controller; this avoids moving model-specific tensors through the workflow coordinator.

### 3.2 Training loop

For a batch of tasks, the controller asynchronously advances executable nodes, while different tasks and same-frontier nodes run concurrently. Each worker group receives the projection of trainable records mapped to its model, stores token-level fragments locally, aligns delayed rewards, computes returns and generalized advantages (Schulman et al., 2015), and performs PPO-style actor/critic updates without reconstructing the DAG. Updated weights are synchronized to the rollout backend (e.g., vLLM or SGLang (Kwon et al., 2023; LMSYS Organization, 2025)); groups may use different parallelism, optimizers, or model sizes.

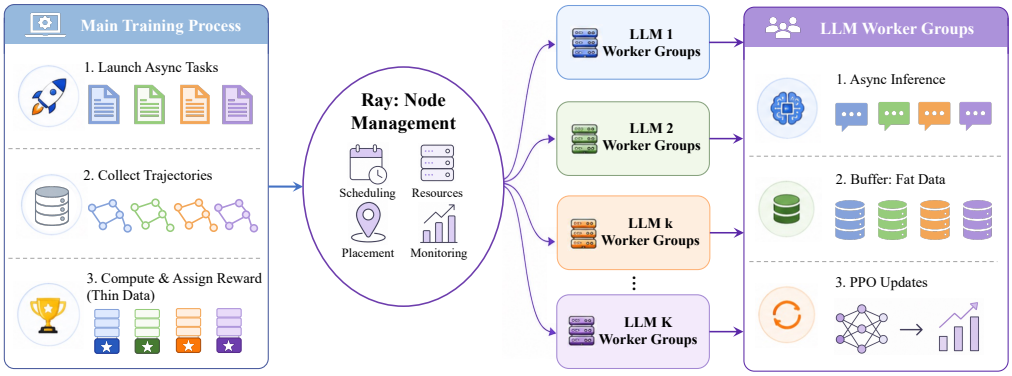


Figure 2: Runtime architecture. A central controller executes the graph and commits rewards; Ray schedules remote work; model-local worker groups generate, buffer, and update the physical LLMs assigned by $\phi$.

The complete controller–worker protocol is given in Appendix D. Inference for newly ready nodes, buffer writes, and updates from earlier reward-complete batches can overlap. After termination, the controller finalizes all node rewards and backfills the matching entries, so each trainable record contains the input, output, and reward needed by the next update.

Data ownership is explicit: a shared model receives fragments from all mapped roles, whereas separated models update only from the roles they serve, exposing the resource–specialization trade-off in Section 5.

The mapping and placement choices are operational configuration points rather than changes to the workflow code. Appendix E gives a concrete launch command, the corresponding role-to-model YAML fragments, and examples of assigning different numbers of distributed nodes to different LLM worker groups. This makes the framework's extensibility reproducible: a new sharing regime or model size can be selected by configuration while the controller, workflow graph, and reward allocator remain unchanged.

## 4 Tasks, Workflows, and Reward Contracts

We evaluate two task families that stress different parts of a multi-agent trajectory. Retrieval QA uses Natural Questions (NQ) (Kwiatkowski et al., 2019) and HotpotQA (Yang et al., 2018). NQ mainly tests query decomposition and evidence selection for short answers, whereas HotpotQA requires evidence from multiple documents and exposes whether search and update roles preserve a useful reasoning state. We report answer F1 on the validation split. The code task follows the released `agentica-org/DeepCoder-14B-Preview` data setting (Luo et al., 2025; Agentica, 2025). Training uses its approximately 24K unique problem–test pairs compiled from TACO-Verified, PrimeIntellect SYNTHETIC-1, and LiveCodeBench (LCB) v5 problems dated 2023-05-01–2024-07-31. Evaluation uses a held-out pool combining LCB v5 and Codeforces problems from the corresponding evaluation collections. Every problem is executed with the same verifier, and the reported strict metric is the fraction of pooled evaluation problems for which all available tests pass; we also report verifier pass rate and average verification turns.

### 4.1 Workflow Instances

The workflow is the experimental unit. QD–Answer decomposes a question into one to three sub-queries, retrieves passages for the sub-queries in parallel, and sends the accumulated evidence to an answerer. QD–Evidence–Answer adds one evidence extractor per sub-query and a context-builder join before the answerer. M-ASK (Chen et al., 2026b) is an unrolled search DAG: a planner initializes the knowledge state, then each turn contains search, retrieval, summary, and update nodes before a temporary answer is evaluated. The reflective code DAG contains planner, coder, verifier, and reflector copies for up to three turns. Figure 5 gives a compact overview of these four templates; their

Table 1: QA benchmark results for RQ1. Average is the arithmetic mean of NQ and HotpotQA when both entries are validation F1. Bold and underlined entries are the best and second-best F1 values within the Qwen2.5-7B comparison; Dr. MAS reports avg@16 and is shown for context.

| Method | Family | Backbone | Metric | NQ | HotpotQA | Average |
|---|---|---|---|---|---|---|
| LLM w/o RAG | single-agent | Qwen2.5-7B | F1 | 17.53 | 17.76 | 17.65 |
| Vanilla RAG | single-agent | Qwen2.5-7B | F1 | 40.60 | 28.33 | 34.47 |
| RRR (Ma et al., 2023) | static RAG/RL | Qwen2.5-7B | F1 | 54.60 | 46.21 | 50.41 |
| BGM (Ke et al., 2024) | static RAG/RL | Qwen2.5-7B | F1 | 54.21 | 46.85 | 50.53 |
| MMOA-RAG (Chen et al., 2025a) | workflow MARL | Qwen2.5-7B | F1 | 55.44 | 49.21 | 52.33 |
| Search-R1 (Jin et al., 2025) | single-agent RL | Qwen2.5-7B | F1 | 52.57 | 46.87 | 49.72 |
| MAO-ARAG (Chen et al., 2025b) | adaptive MARL | Qwen2.5-7B | F1 | 36.82 | 46.65 | 41.74 |
| M-ASK (Chen et al., 2026b) | workflow MARL | Qwen2.5-7B | F1 | 57.40 | **58.31** | 57.86 |
| JADE (Chen et al., 2026c) | adaptive MARL | Qwen2.5-7B | F1 | <u>59.45</u> | 57.02 | <u>58.24</u> |
| MAPPO (MHGPO source) (Chen et al., 2026a) | general MARL | Llama3.1-8B | F1 | – | 46.40 | – |
| MHGPO-FoF(os) (Chen et al., 2026a) | general MARL | Llama3.1-8B | F1 | – | 50.86 | – |
| Dr. MAS (shared) (Feng et al., 2026) | general MARL | Qwen2.5-7B | avg@16 | 47.4 | 42.5 | – |
| Dr. MAS (non-shared) (Feng et al., 2026) | general MARL | Qwen2.5-7B | avg@16 | 47.7 | 44.0 | – |
| UnityMAS-O (Ours) | general MARL | Qwen2.5-7B | F1 | **62.10** | <u>57.30</u> | **59.70** |

Table 2: Code-workflow effectiveness with independent planner, coder, and reflector models (3×). Direct and trained workflows use the same graph, verifier, and pooled LCB v5 + Codeforces problems. Scores are strict all-passed rates; gains are percentage points (pp).

| Setting | Backbone | All-passed | Gain (pp) |
|---|---|---|---|
| Direct workflow (untrained) | 3×Qwen3-4B | 0.255 | – |
| UnityMAS-O (trained) | 3×Qwen3-4B | **0.686** | **+43.1** |
| Direct workflow (untrained) | 3×Qwen3-8B | 0.290 | – |
| UnityMAS-O (trained) | 3×Qwen3-8B | **0.738** | **+44.8** |

concrete node contracts, stopping conditions, and reward allocations are specified in Appendix F. In every case, repeated role types are distinct DAG nodes after unrolling, and a stop condition masks later nodes.

### 4.2 Reward contracts

For every trainable node invocation $v$, the controller combines three optional reward scales,

$$r_v = \lambda_{\text{node}} r_v^{\text{node}} + \lambda_{\text{turn}} r_v^{\text{turn}} + \lambda_{\text{traj}} r_v^{\text{traj}}. \tag{7}$$

The node term checks the current output (for example, a parser/schema constraint or a role-specific rubric); the turn term measures progress after a local stage or frontier; and the trajectory term is a delayed outcome of the completed task. Coefficients may be zero, so a workflow can use dense local feedback, sparse terminal feedback, or a mixture. In QD–Answer and QD–Evidence–Answer, JSON/evidence validity supplies the node term and final answer F1 is the trajectory term shared by realized trainable nodes. M-ASK adds the turn-level marginal F1 used by its search, summary, and update roles (Chen et al., 2026b); trajectory-level final-outcome allocation follows the shared terminal-reward pattern used by MMOA-RAG (Chen et al., 2025a); and the reflective code workflow uses parser/tag checks and verifier pass-rate deltas, with all-passed serving as an early-stop predicate. Appendix F gives the concrete role matrix, state fields, mapping configurations, and reward allocation.

## 5 Experiments

We organize the experiments around three questions. **RQ1 (effectiveness):** does UnityMAS-O compare favorably with single-agent prompting/RL, workflow-specific MARL, and general LLM-MARL systems? **RQ2 (co-training):** after the workflow and reward contract are fixed, does role-aware MARL improve every task family? **RQ3 (sharing):** when does mapping several logical roles to one policy help or hurt?

Table 3: Published training-gain context grouped by dataset. The Method column lists the system name; the training comparison and workflow are described in the surrounding text and Appendix I. Before, After, and Δ report the corresponding scores and percentage-point gain. "LCB+CF" denotes the pooled LCB v5 + Codeforces set; "4o-mini" denotes GPT-4o-mini. Source metrics and evaluation sets are retained for external rows; only the UnityMAS-O gains are bold.

| Method | Task / metric | Backbone | Before | After | Δ |
|---|---|---|---|---|---|
| *Retrieval QA: NQ* | | | | | |
| JADE | NQ / F1 | Qwen2.5-7B | 36.06 | 59.45 | +23.39 |
| MAO-ARAG | NQ / F1 | Qwen2.5-7B + 4o-mini | 50.57 | 54.50 | +3.93 |
| Dr. MAS | NQ / avg@16 | Qwen2.5-7B | 27.1 | 47.7 | +20.6 |
| UnityMAS-O | NQ / F1 | Qwen2.5-7B | 26.0 | 62.1 | **+36.1** |
| *Retrieval QA: HotpotQA* | | | | | |
| MHGPO | HotpotQA / F1 | Llama3.1-8B | 21.452 | 50.858 | +29.406 |
| JADE | HotpotQA / F1 | Qwen2.5-7B | 46.20 | 57.02 | +10.82 |
| MAO-ARAG | HotpotQA / F1 | Qwen2.5-7B + 4o-mini | 49.68 | 53.80 | +4.12 |
| Dr. MAS | HotpotQA / avg@16 | Qwen2.5-7B | 24.4 | 44.0 | +19.6 |
| UnityMAS-O | HotpotQA / F1 | Qwen2.5-7B | 25.4 | 57.3 | **+31.9** |
| *Executable code* | | | | | |
| STRONGER-MAS | LiveCodeBench / pass@1 | Qwen3-8B | 28.0 | 33.1 | +5.1 |
| STRONGER-MAS | APPS / pass@1 | Qwen3-8B | 44.4 | 46.5 | +2.1 |
| STRONGER-MAS | CodeContests / pass@1 | Qwen3-8B | 17.6 | 18.1 | +0.5 |
| UnityMAS-O | LCB+CF / all-passed | 3×Qwen3-4B | 25.5 | 68.6 | **+43.1** |
| UnityMAS-O | LCB+CF / all-passed | 3×Qwen3-8B | 29.0 | 73.8 | **+44.8** |

### 5.1 Evaluation protocol

We evaluate retrieval QA on NQ and HotpotQA (Kwiatkowski et al., 2019; Yang et al., 2018) and executable code on the pooled held-out LCB v5–Codeforces set described in Sec. 4. QA uses normalized F1; code uses strict all-passed, with verifier pass rate and verification turns as auxiliary metrics. Cross-method QA uses Qwen2.5-7B, while RQ2 adds Qwen2.5 sizes and code uses Qwen3. We report the best checkpoint under each stated protocol; RQ2 compares controlled before–after changes under fixed workflow contracts.

### 5.2 RQ1: Cross-method benchmark effectiveness

**Retrieval QA.** Table 1 compares single-agent RL, workflow-specific MARL, adaptive agentic search, and general LLM-MARL. Our Qwen2.5-7B result uses the iterative-search (M-ASK) template in Figure 5. Among the reported Qwen2.5-7B F1 results, UnityMAS-O achieves the highest NQ score (62.10) and two-dataset average (59.70). Against single-agent RL, it improves average F1 by 9.98 points over Search-R1; it also exceeds RRR and BGM by 9.29 and 9.17 points. Against multi-agent RL, it exceeds MMOA-RAG by 7.37 average points and JADE, the strongest baseline average, by 1.46 points. The gains therefore extend beyond adding retrieval or a manually assembled multi-agent workflow.

By dataset, UnityMAS-O improves over JADE by 2.65 points on NQ and reaches 57.30 on HotpotQA, behind M-ASK (58.31) but ahead of JADE (57.02). It therefore achieves the highest mean while retaining competitive multi-hop reasoning; its HotpotQA score is 6.44 points above MHGPO-FoF(os) with a similarly sized Llama3.1-8B backbone.

**Executable code.** For executable code, Table 2 compares the same manually specified reflective workflow before training with its best UnityMAS-O checkpoint on the pooled LiveCodeBench (LCB) v5–Codeforces evaluation set. Each 3× row uses separate planner, coder, and reflector models. The graph, verifier, held-out problems, and Qwen3 backbone are fixed within each pair, so the difference isolates the effect of role-aware MARL. We use strict all-passed: a problem counts as solved only when every available test passes.

Training raises the 4B workflow from 0.255 to 0.686 (+43.1 points) and the 8B workflow from 0.290 to 0.738 (+44.8). The 4B-to-8B backbone change adds only 3.5 points before training, whereas co-

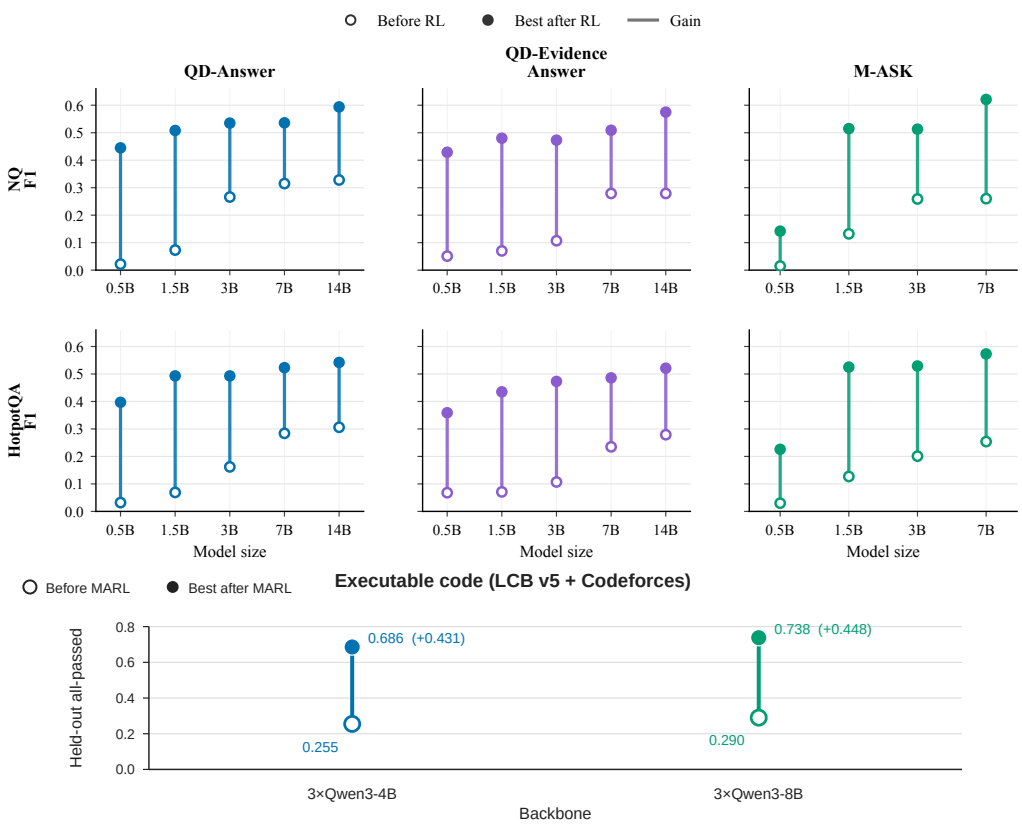


Figure 3: RQ2 co-training evidence across retrieval QA and DeepCoder-derived executable code. The six upper panels show untrained (open markers) and best-after-MARL (filled markers) validation F1 for QD–Answer, QD–Evidence–Answer, and M-ASK on NQ and HotpotQA. The lower panel applies the same before/after encoding to strict all-passed results on the pooled LiveCodeBench v5–Codeforces problems; annotations report the exact values and gains for 3×Qwen3-4B and 3×Qwen3-8B.

training adds more than 43 at either size, showing that role-aware optimization drives the improvement within the fixed reflective graph.

### 5.3 RQ2: Workflow co-training effectiveness

We hold each workflow, reward contract, and data protocol fixed and compare its initialization with the best MARL checkpoint. Appendix Table 6 provides the full model-size sweep. To relate these gains to existing training approaches, Table 3 includes heterogeneous-group optimization for multi-agent search (MHGPO (Chen et al., 2026a)), joint planning and execution training (JADE (Chen et al., 2026c)), planner optimization with frozen executors (MAO-ARAG (Chen et al., 2025b)), agent-wise reward normalization (Dr. MAS (Feng et al., 2026)), and agent- and turn-wise GRPO (STRONGER-MAS (Zhao et al., 2026)). Appendix I describes each baseline.

For MHGPO, JADE, MAO-ARAG, STRONGER-MAS, and UnityMAS-O, Before and After measure the effect of training an initially unoptimized workflow or planner. Dr. MAS instead compares two training algorithms under the same non-shared configuration: vanilla GRPO and agent-wise normalization. Its delta measures an optimizer improvement rather than the full gain from an untrained workflow.

On comparable retrieval-QA rows, UnityMAS-O gains $+36.1$ F1 on NQ and $+31.9$ on HotpotQA, exceeding MHGPO's largest reported gain (29.406). On code, its $+43.1/+44.8$ all-passed gains exceed STRONGER-MAS's largest source gain (5.1 pass@1 points). Figure 3 shows that every QA workflow improves; the code checkpoints reach 0.686/0.738 all-passed and use 1.76/1.68 verification

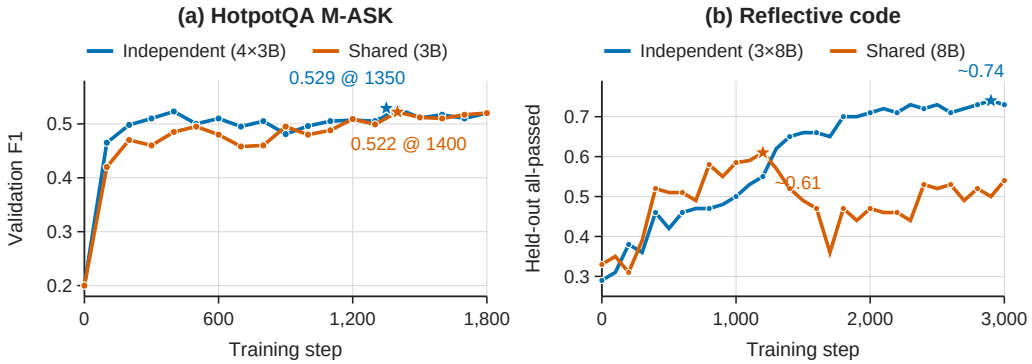


Figure 4: RQ3 parameter-sharing exploration. Left: HotpotQA M-ASK with four independent Qwen2.5-3B models versus one shared Qwen2.5-3B model. Right: the Qwen3-8B DeepCoder-derived reflective-code trace, shown through 3,000 steps. Shared parameters converge more slowly on M-ASK but finish near the independent score, whereas independent code models retain a clear quality advantage.

turns. Thus the same reward-backfill and worker-local update interface transfers across graph shapes, output lengths, and evaluators.

### 5.4 RQ3: Parameter-sharing exploration

We explore multi-agent optimization under two parameterization settings: independent policies for each role and a single policy shared across roles. Figure 4 compares these settings on HotpotQA M-ASK and reflective code generation.

On HotpotQA M-ASK, four independent Qwen2.5-3B models reach 0.529 at step 1350, compared with 0.522 at step 1400 for one shared model. The small gap suggests that sharing is effective when roles solve related, short-horizon subproblems, although convergence is slightly slower. Code exhibits the opposite pattern: near step 2900, independent Qwen3-8B reaches about 0.74, while the shared model peaks near 0.61 and later fluctuates between 0.40 and 0.53. Heterogeneous role objectives and delayed verifier credit create stronger interference under sharing, making independent policies more stable but more resource-intensive.

## 6 Conclusion

UnityMAS-O turns manually designed LLM workflows into trainable multi-agent systems through asynchronous DAG execution, explicit role-to-model mappings, and graph-aware credit assignment. It integrates node, turn, and trajectory rewards into model-local buffers, enabling PPO updates across heterogeneous roles without same-state candidate groups. Across retrieval QA and executable code tasks, training consistently improves performance: validation F1 rises from 26.0 to 62.1 on NQ and from 25.4 to 57.3 on HotpotQA, while strict all-passed code success increases from 25.5% to 68.6% with Qwen3-4B and from 29.0% to 73.8% with Qwen3-8B. These gains across parallel retrieval, iterative search, and reflective code workflows demonstrate the effectiveness and generality of the proposed optimization interface. Parameter-sharing studies further reveal a workload-dependent trade-off: sharing remains competitive for related short-horizon QA roles, whereas independent policies are more stable for heterogeneous code roles with delayed verifier feedback. Overall, UnityMAS-O provides a general framework for optimizing diverse LLM multi-agent workflows under shared, partially shared, and independent policy settings.

## REFERENCES


Agentica. DeepCoder-14B-Preview model card. Hugging Face, 2025. URL https://huggingface.co/agentica-org/DeepCoder-14B-Preview.

Guanzhong Chen, Shaoxiong Yang, Chao Li, Wei Liu, Jian Luan, and Zenglin Xu. End-to-end optimization of llm-driven multi-agent search systems via heterogeneous-group-based reinforcement learning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, 2026a. URL https://aclanthology.org/2026.acl-long.1399/.

Yiqun Chen, Lingyong Yan, Weiwei Sun, Xinyu Ma, Yi Zhang, Shuaiqiang Wang, Dawei Yin, Yiming Yang, and Jiaxin Mao. Improving retrieval-augmented generation through multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, 2025a. doi: 10.48550/arXiv.2501.15228. URL https://arxiv.org/abs/2501.15228.

Yiqun Chen, Erhan Zhang, Lingyong Yan, Shuaiqiang Wang, Jizhou Huang, Dawei Yin, and Jiaxin Mao. MAO-ARAG: Multi-agent orchestration for adaptive retrieval-augmented generation. *arXiv preprint arXiv:2508.01005*, 2025b. doi: 10.48550/arXiv.2508.01005. URL https://arxiv.org/abs/2508.01005.

Yiqun Chen, Lingyong Yan, Zixuan Yang, Erhan Zhang, Jiashu Zhao, Shuaiqiang Wang, Dawei Yin, and Jiaxin Mao. Beyond monolithic architectures: A multi-agent search and knowledge optimization framework for agentic search. *arXiv preprint arXiv:2601.04703*, 2026b. URL https://arxiv.org/abs/2601.04703.

Yiqun Chen, Erhan Zhang, Tianyi Hu, Shijie Wang, Zixuan Yang, Meizhi Zhong, Xiaochi Wei, Yan Gao, Yi Wu, Yao Hu, and Jiaxin Mao. JADE: Bridging the strategic-operational gap in dynamic agentic RAG. *arXiv preprint arXiv:2601.21916*, 2026c. doi: 10.48550/arXiv.2601.21916. URL https://arxiv.org/abs/2601.21916.

Lang Feng, Longtao Zheng, Shuo He, Fuxiang Zhang, and Bo An. Dr. MAS: Stable reinforcement learning for multi-agent LLM systems. *arXiv preprint arXiv:2602.08847*, 2026. doi: 10.48550/arXiv.2602.08847. URL https://arxiv.org/abs/2602.08847.

Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Zilin Zhu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, Weikai Fang, Xianyu, Yu Cao, Haotian Xu, and Yiming Liu. OpenRLHF: An easy-to-use, scalable and high-performance RLHF framework. *arXiv preprint arXiv:2405.11143*, 2024. doi: 10.48550/arXiv.2405.11143. URL https://arxiv.org/abs/2405.11143.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025. URL https://arxiv.org/abs/2503.09516.

Zixuan Ke, Weize Kong, Cheng Li, Mingyang Zhang, Qiaozhu Mei, and Michael Bendersky. Bridging the preference gap between retrievers and LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 10438–10451, 2024. URL https://aclanthology.org/2024.acl-long.562/.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019. doi: 10.1162/tacl_a_00276. URL https://aclanthology.org/Q19-1026/.

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*, pp. 611–626, 2023. doi: 10.1145/3600006.3613165. URL https://arxiv.org/abs/2309.06180.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Systems*, 2023. doi: 10.48550/arXiv.2303.17760. URL https://arxiv.org/abs/2303.17760.

LMSYS Organization. slime: An SGLang-native post-training framework for RL scaling. https://lmsys.org/blog/2025-07-09-slime/, 2025.

Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. DeepCoder: A fully open-source 14b coder at o3-mini level. https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-mini-Level-1cf81902c14680b3bee5eb349 2025. Notion Blog.

Xinbei Ma, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. Query rewriting in retrieval-augmented large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 5303–5315, 2023. URL https://aclanthology.org/2023.emnlp-main.322/.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pp. 561–577, 2018. URL https://www.usenix.org/conference/osdi18/presentation/moritz.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A. Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03350. Self-Ask prompting baseline; arXiv:2210.03350.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 15174–15186, 2024. URL https://aclanthology.org/2024.acl-long.810/.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015. doi: 10.48550/arXiv.1506.02438. URL https://arxiv.org/abs/1506.02438.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. doi: 10.48550/arXiv.1707.06347. URL https://arxiv.org/abs/1707.06347.

Zhihong Shao, Peiyi Wang, Qihao Zhu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. *arXiv preprint arXiv:2409.19256*, 2024. doi: 10.48550/arXiv.2409.19256. URL https://arxiv.org/abs/2409.19256.

Noah Shinn, Federico Cassano, Ashay Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2303.11366.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, pp. 10014–10037, 2023. URL https://aclanthology.org/2023.acl-long.557/.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouedec. TRL: Transformer reinforcement learning. https://github.com/huggingface/trl, 2020.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023. doi: 10.48550/arXiv.2308.08155. URL https://arxiv.org/abs/2308.08155.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

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

Kaiyan Zhang, Kai Tian, Runze Liu, Sihang Zeng, Xuekai Zhu, Guoli Jia, Yuchen Fan, Xingtai Lv, Yuxin Zuo, Che Jiang, Yuru Wang, Jianyu Wang, Ermo Hua, Xinwei Long, Junqi Gao, Youbang Sun, Zhiyuan Ma, Ganqu Cui, Ning Ding, Biqing Qi, and Bowen Zhou. MARTI: A framework for multi-agent LLM systems reinforced training and inference. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=E7jZqo0A50.

Zhixia Zhang, Zixuan Huang, Xin Xia, Deqing Wang, Fuzhen Zhuang, Shuai Ma, Ning Ding, Yaodong Yang, Jianxin Li, and Yikun Ban. Heterogeneous agent collaborative reinforcement learning. *arXiv preprint arXiv:2603.02604*, 2026b. doi: 10.48550/arXiv.2603.02604. URL https://arxiv.org/abs/2603.02604.

Yujie Zhao, Lanxiang Hu, Yang Wang, Minmin Hou, Hao Zhang, Ke Ding, and Jishen Zhao. STRONGER-MAS: Multi-agent reinforcement learning for collaborative LLMs. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=IdF6JqXWzx.

# A SUBMISSION STATEMENTS

**AI use statement.** AI tools were used for language editing and code-organization assistance. The authors verified the technical content, experimental numbers, citations, and final manuscript.

**Reproducibility statement.** The anonymous implementation is available at http://anonymous.4open.science/r/UnityMAS-O-8EAE. The repository provides the workflow definitions, runtime and reward components, training configurations, and scripts needed to reproduce the reported experiments. The main paper specifies the workflow graphs, reward rules, sharing regimes, evaluation metrics.

# B RELATED WORK

## B.1 SINGLE-AGENT AND MULTI-AGENT SYSTEMS

ReAct interleaves reasoning and tool actions in one agent (Yao et al., 2023); Self-Ask decomposes questions into follow-up queries (Press et al., 2023); IRCoT interleaves retrieval with multi-step reasoning (Trivedi et al., 2023); and Reflexion uses verbal feedback to improve later attempts (Shinn et al., 2023). CAMEL, AutoGen, and ChatDev show the complementary value of role-based orchestration (Li et al., 2023; Wu et al., 2023; Qian et al., 2024). These methods motivate our workflow choices, but generally optimize prompts or inference-time control rather than a shared RL interface over all roles.

Workflow-specific LLM–MARL systems include MMOA-RAG and MAO-ARAG for retrieval pipelines (Chen et al., 2025a;b), and M-ASK and JADE for agentic search (Chen et al., 2026b;c). General multi-agent training systems include MHGPO, MARTI, Dr. MAS, STRONGER-MAS, and HACRL (Chen et al., 2026a; Zhang et al., 2026a; Feng et al., 2026; Zhao et al., 2026; Zhang et al., 2026b). Search-R1 provides a strong single-agent RL reference for tool-augmented QA (Jin et al., 2025), while TRL, OpenRLHF, slime, and HybridFlow provide scalable single-policy post-training infrastructure (von Werra et al., 2020; Hu et al., 2024; LMSYS Organization, 2025; Sheng et al., 2024).

The distinction from UnityMAS-O is the optimization interface. Single-agent methods expose one policy trajectory and usually attach feedback to a response or episode; prompt-oriented multi-agent systems expose role interactions but leave workflow and reward assignment outside the learner. Existing MARL systems provide outcome optimization for particular pipelines, yet their role graphs, model groups, or credit rules are commonly coupled to that pipeline. UnityMAS-O instead keeps the user DAG, role-to-model map, tool transitions, and reward allocator configurable in one runtime. This separation enables matched studies of sharing and makes one PPO-style training loop applicable to parallel retrieval, iterative search, and executable code. Appendix G gives the structured comparison with general LLM–MARL systems, including topology, rollout grouping, reward/credit, and policy-resource interfaces.

# C LIMITATIONS

Our evidence is limited to two QA datasets and one executable-code construction, and the public baseline protocols are not fully matched. The DAG contract requires bounded unrolling for workflows described with loops; dynamic per-sample branch selection is supported, while unbounded cyclic policies remain outside the current interface. The framework currently uses PPO-style updates; other algorithms, larger heterogeneous model pools, and additional environments remain to be evaluated. Independent role models improve the complex code workflow in the reported study, at the cost of additional inference and optimizer state; shared policies remain a useful resource-efficient option for shorter, more homogeneous roles. The current snapshot contains the workflow definitions, figures, and full before/after tables; configuration and reward code are not included in this submission snapshot.

## D ASYNCHRONOUS ROLLOUT AND JOINT UPDATE

Algorithm 1 instantiates the general controller–worker protocol described in Sec. 3. The controller keeps the graph-level trace and advances all enabled frontiers from all active tasks asynchronously. A worker group performs three operations concurrently: it generates outputs with its inference replica, appends the token-level record and a reward placeholder to the model-local buffer, and consumes reward-complete minibatches for PPO updates. Tool nodes only return observations to the controller. When a task terminates, the controller evaluates the configured node, turn, and trajectory terms and backfills the composite reward into the exact trainable entries that generated the trajectory. Consequently, each model-local buffer contains complete $\langle \mathrm{input}, \mathrm{output}, \mathrm{reward} \rangle$ records before the joint update phase.

**Algorithm 1** UnityMAS-O Asynchronous Rollout and Joint Update

**Input:** Task distribution $\mathcal{D}$, workflow DAG $\mathcal{G}$, role-to-model map $\phi$
**Input:** Reward contract $\mathcal{R}$, workers $\{W_i\}_{i=1}^{K}$, rollout size $B$
**Output:** Updated model parameters $\{\theta_{m_i}\}$ and synchronized inference replicas

Initialize graph trace store $\mathcal{T}$ and model-local buffers $\{\mathcal{B}_i\}_{i=1}^{K}$
**for** iteration $j = 1, \ldots, M$ **do**
  Set the rollout counter $n_{\mathrm{collected}} \leftarrow 0$
  *// Phase I: asynchronous data collection*
  **while** $n_{\mathrm{collected}} < B$ **do**
    Admit tasks $x \sim \mathcal{D}$ into the active set and initialize each state and ready frontier
    **while** some active task has an enabled node **do**
      Controller dispatches all ready nodes across active tasks and frontiers asynchronously
      *// Workers overlap inference, buffer writes, and updates from earlier ready batches*
      **for all** dispatched nodes $v$ **do**
        **if** $v \in \mathcal{V}_{\mathrm{LLM}}$ **then**
          Send $(s_v, v)$ to $W_{\phi(v)}$; asynchronously infer with vLLM/SGLang
          Append $\langle \mathrm{input}, \mathrm{output}, \square \rangle$ and token tensors to $\mathcal{B}_i$, where $\phi(v) = m_i$
        **else**
          Execute tool/environment transition and return observation $o_v$
        **end if**
        Merge outputs into $\mathcal{T}_x$; evaluate edge predicates, masks, and the next frontier
      **end for**
    **end while**
    Finalize $r_v = \mathcal{R}_v(s_v, a_v, o_v, h_v, \mathcal{T}_x)$ for every realized trainable node and backfill its buffer entry
    $n_{\mathrm{collected}} \leftarrow n_{\mathrm{collected}} + |\mathcal{V}_{\mathrm{LLM}}(\tau_x)|$ (drained entries do not count twice)
    **if** a worker buffer contains a reward-complete minibatch **then**
      Enqueue it for an asynchronous PPO update while collection continues
    **end if**
  **end while**
  *// Phase II: drain remaining joint updates*
  **for all** worker groups $W_i$ in parallel **do**
    Form ready minibatches of $\langle \mathrm{input}, \mathrm{output}, \mathrm{reward} \rangle$ from $\mathcal{B}_i$
    Compute returns and advantages; update $\theta_{m_i}$ and its critic with PPO
    Synchronize updated weights to the inference replica of $W_i$
  **end for**
  Clear consumed entries while retaining unfinished task traces
**end for**

The first phase is intentionally task-asynchronous: admitting a new task does not wait for older tasks, and each task advances whenever one of its graph frontiers becomes ready. A tool completion can therefore release a downstream role while another task is still generating an upstream output. The controller stores only lightweight state and tensor references; the worker that owns $\phi(v)$ keeps the token-level tensors. The second phase drains reward-complete minibatches in parallel across worker groups, computes GAE locally, updates the actor and critic, and synchronizes the new weights to inference replicas. This separation preserves the DAG for execution and diagnostics while allowing the optimization step to operate directly on model-local buffers.

## E RUNNING AND CONFIGURING UNITYMAS-O

This section documents the concrete interfaces used by the implementation. The training entry point is `verl/experimental/star_ppo/main_ppo.py`; `ray_trainer.py` owns controller-side workflow execution and reward commitment, `star_fsdp_workers.py` owns model-local rollout and PPO work, and `trajectory_buffer.py` stores the model-local records. Workflow runners and reward allocators are registered under `verl/experimental/star_ppo/workflows/` and `verl/experimental/star_ppo/reward_allocators/`, respectively. The

examples below use the repository's Hydra configurations and the launcher in `examples/star_ppo/common/run_per_node_background.sh`.

### E.1 Launching a run and the main controls

The environment is prepared once per cluster and the same launcher is invoked once on every node. The rank-zero process starts the Ray head and launches training after all ranks have joined; other ranks start Ray workers. The following is the minimal pattern (the values are placeholders and must be replaced for a local cluster):

```
cd /path/to/UnityMAS-O
conda activate verl
export UNITYMAS_ROOT=/path/to/storage
export HEAD_IP=<ray-head-ip>

RANK=0 HEAD_IP=${HEAD_IP} WORLD_SIZE=3 \
  CONFIG_NAME=star_code_iterative_plan_code_reflect_trainer \
  bash examples/star_ppo/common/run_per_node_background.sh
```

The command is repeated with `RANK=1` and `RANK=2` on the other nodes. `CONFIG_NAME` selects a YAML file in `verl/experimental/star_ppo/config/` (without the `.yaml` suffix). Dataset locations are supplied with `TRAIN_JSONL`/`VAL_JSONL` for executable code or `TRAIN_PARQUET`/`VAL_PARQUET` and `DATASET_NAME` for QA/search. `PROJECT_NAME` and `EXPERIMENT_NAME` control tracking only.

The most important runtime controls fall into four groups. `GEN_BATCH_SIZE`, `VAL_BATCH_SIZE`, `ACTOR_PPO_MINI_BATCH_SIZE`, and the actor/critic micro-batch variables determine update batches. `STAR_MAX_INFLIGHT_QUERIES` and `STAR_MAX_PARALLEL_ROLLOUTS_PER_MODEL` control asynchronous request overlap; the two `STAR_LLM_MICROBATCH_*` variables control worker-side request microbatching.

`ROLLOUT_PROMPT_LENGTH`, `ROLLOUT_RESPONSE_LENGTH`, `ROLLOUT_MAX_MODEL_LEN`, `ROLLOUT_TENSOR_MODEL_PARALLEL_SIZE`, and `ROLLOUT_GPU_MEMORY_UTILIZATION` control the vLLM/SGLang replica. `CODE_MAX_TURNS` and `CODE_STOP_ON_ALL_PASSED` control the reflective-code loop, while `MASK_MAX_TURNS` and `MASK_STOP_ON_SEARCH_END` control M-ASK.

Timeout variables such as `STAR_QUERY_TIMEOUT_SECONDS`, `STAR_WORKFLOW_BATCH_TIMEOUT_SECONDS`, and `STAR_LLM_TIMEOUT_SECONDS` bound remote calls. `STAR_WORKFLOW_DEBUG` and `STAR_VAL_DEBUG` expose trace samples without changing the optimization target.

### E.2 Configuring full, partial, and independent sharing

The YAML configuration has two linked parts. Under `trainer.llm_engines`, each physical model is declared by a unique `model_id`, a checkpoint path, and its placement. Under `star.workflow`, each trainable role names the `model_id` that should serve it. The controller uses this string as the map $\phi$ and sends both inference and the resulting buffer entry to that worker group. The code configuration contains the following role declarations (irrelevant prompt fields are omitted):

```
# star_code_iterative_plan_code_reflect_trainer.yaml
trainer:
  llm_engines:
    - model_id: planner_llm       # model_path: ${PLANNER_MODEL_PATH}
    - model_id: coder_llm         # model_path: ${CODER_MODEL_PATH}
    - model_id: reflection_llm    # model_path: ${REFLECTION_MODEL_PATH}
star:
  workflow:
    code:
      planner:    {model_id: planner_llm,    agent_id: planner_agent}
      coder:      {model_id: coder_llm,      agent_id: coder_agent}
      reflection: {model_id: reflection_llm, agent_id: reflection_agent}
```

This is the fully separated setting: changing one of the three model paths changes that role's physical model but leaves the DAG and prompts intact. The shared code configuration replaces the three engine

identifiers and all three role fields by `shared_agent_llm`, and sets `SHARED_MODEL_PATH`. The same pattern is used by the M-ASK shared configuration, whose role fields all contain `model_id: shared_agent_llm`.

Partial sharing is expressed by reusing an identifier for only a subset of roles. The released M-ASK configuration illustrates this directly: planning and answer use `reasoning_agent_llm`, while search, summary, and update use separate identifiers.

```
# star_iterative_plan_search_summary_update_answer_f1_trainer.yaml
plan:    {model_id: reasoning_agent_llm, agent_id: planning_agent}
answer:  {model_id: reasoning_agent_llm, agent_id: answer_agent}
search:  {model_id: search_agent,       agent_id: search_agent}
summary: {model_id: summary_agent,      agent_id: summary_agent}
update:  {model_id: update_agent,       agent_id: update_agent}
```

Thus full sharing, partial sharing, and full separation are all edits to the role-to-`model_id` map. The runner, reward allocator, and graph execution do not branch on the sharing choice. A new workflow follows the same extension contract: implement a `WorkflowRunner.run_batch`, implement a `RewardAllocator.allocate`, and add a Hydra YAML file that declares the engines, role map, tools, and data paths.

### E.3 Allocating different numbers of nodes to different LLMs

Each entry under `trainer.llm_engines` also exposes `nnodes` and `n_gpus_per_node`. The corresponding environment variables are role-specific, for example `PLANNER_NNODES`/`PLANNER_GPUS_PER_NODE`, `CODER_NNODES`/`CODER_GPUS_PER_NODE`, and `REFLECTION_NNODES`/`REFLECTION_GPUS_PER_NODE`; M-ASK uses the analogous `REASONING_AGENT_*`, `SEARCH_AGENT_*`, `SUMMARY_AGENT_*`, and `UPDATE_AGENT_*` variables. A typical declaration is:

```
trainer:
  llm_engines:
    - model_id: planner_llm
      nnodes: 2
      n_gpus_per_node: 8
      model_path: ${PLANNER_MODEL_PATH}
    - model_id: coder_llm
      nnodes: 2
      n_gpus_per_node: 8
      model_path: ${CODER_MODEL_PATH}
    - model_id: reflection_llm
      nnodes: 2
      n_gpus_per_node: 8
      model_path: ${REFLECTION_MODEL_PATH}
```

The three entries above reserve six 8-GPU nodes in total. Replacing each `nnodes: 2` by `nnodes: 3` reserves nine such nodes, as in the 3×Qwen3-8B code run. The 3×Qwen3-4B run uses two nodes per role (six nodes total). QA runs use one 8-GPU node per independently trained LLM. Different entries may use different checkpoint paths and different node counts, so a workflow can combine small and large models or allocate additional replicas only to a bottleneck role. Subject to available Ray resources, `nnodes` is an independent positive integer for each physical model; the controller does not impose a one-node-per-role assumption. `ROLLOUT_TENSOR_MODEL_PARALLEL_SIZE`, FSDP settings, and micro-batch sizes can then be tuned per run to fit the chosen placement.

The separation between logical roles and these physical engine declarations is the extensibility point: changing model size, node count, or sharing changes the worker layout and data routing, while the workflow DAG, tool calls, and reward allocator remain the same. This is also why the same rollout/update pseudocode applies to all experiments.

## F Workflow specifications

This appendix gives the concrete workflow semantics used by the experiments and mirrors the public implementation's runner, configuration, and reward-allocator interfaces. A workflow state contains the task input, role outputs, tool results, and a structured graph trace with one record per realized node. The runner keeps this trace in an invocation-indexed buffer; each model-local PPO buffer is its

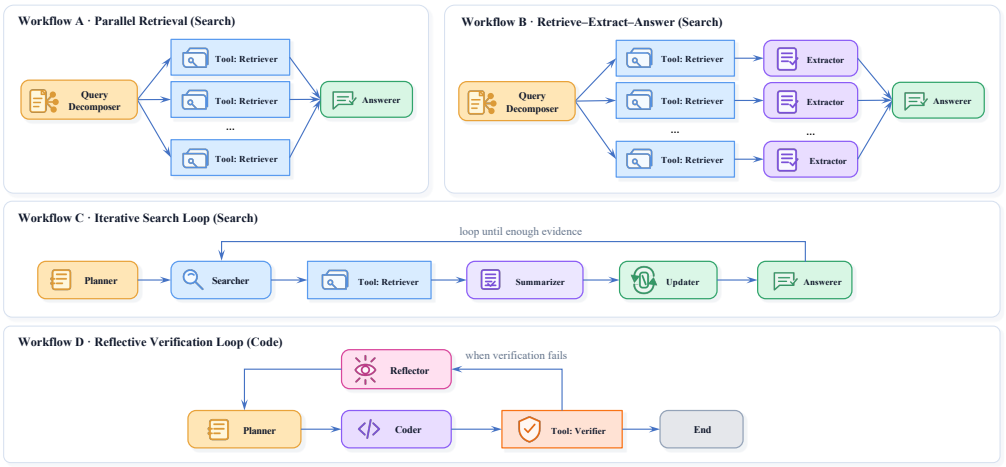


Figure 5: Workflow templates used in the experiments: parallel retrieval, retrieve–extract–answer, iterative search, and reflective verification.

projection onto trainable nodes and retains the token-level tensors needed for updating that model. Each trainable node also emits a format reward; invalid tags or malformed structured output receive a negative format reward. For every trainable invocation, `total_reward = task_reward + format_weight * format_reward`. Tool nodes produce observations and never enter the PPO buffer.

Figure 5 summarizes the four templates instantiated below: parallel retrieval (QD–Answer), retrieve–extract–answer (QD–Evidence–Answer), iterative search (M-ASK), and reflective verification (the code workflow). The figure is a structural overview; the following subsections give the state fields, trainable and tool nodes, stopping rules, and reward terms for each template.

### F.1 Graph execution and query-decomposition RAG

The configurable graph runner accepts a finite DAG with start nodes, end nodes, node configurations, and conditional edges. Nodes on the same frontier are executed concurrently. Each node writes its parsed output to the state and evaluates its outgoing edges; edge conditions may inspect existing node values. The controller advances only to enabled successors, so the realized execution is a topological prefix of the DAG. Execution stops when the frontier is empty, an end node is reached with `stop_on_end=true`, or the maximum graph step count is reached. A retrieval tool can batch sub-queries, retry failed calls, and return top-$k$ passages. In the reported QD workflows, retrieval uses top-$k = 4$ passages per sub-query; nodes on a common frontier are awaited with an asynchronous gather before edge conditions select the next frontier.

In the QD–Answer workflow, the decomposer receives the question and must return a JSON object with a non-empty `sub_queries` list containing one to three strings. The retriever tool executes these queries in parallel. The answerer receives the original question, sub-queries, and retrieved passages and emits a tagged `final_answer` (or "Unknown" when evidence is insufficient). In QD–Evidence–Answer, one evidence extractor is activated for each available sub-query; each extractor reads its own passages and emits a tagged `sub_evidence` span. A context-builder tool concatenates the evidence before the answerer runs. The outcome F1 is computed from the final answer and shared among every trainable node instance that participated in the trajectory, with each node additionally receiving its own format reward. The same graph runner also supports summarization, rewrite, selector, and two-agent RAG variants through configuration changes.

These two workflows correspond to the first two rows of Table 4. The table makes explicit that retrieval and context construction are tool transitions, while the decomposer, evidence extractors, and answerer are the trainable nodes that receive format checks and the delayed answer-F1 outcome. Neither workflow defines a turn-level progress term, because the graph terminates after the answer node and has no iterative frontier.

### F.2 M-ASK ITERATIVE SEARCH

M-ASK has five logical roles: planning, search, summary, update, and answer. In the independent configuration, planning and answer use one reasoning model while search, summary, and update use separate model groups; the shared configuration maps all five roles to one physical LLM. The planner initializes a knowledge state and emits paired sub-questions and answers together with an initial final answer. The answer role then normalizes this initial answer.

Each search turn emits either a tagged query or `<end>`. A query invokes the retriever (top-$k = 5$, with up to five retrieval attempts), the summary role compresses the returned passages into tagged evidence, and the update role either attaches evidence to an existing reasoning step or adds a new step. The answer role produces a new temporary answer after every update. The loop stops at `<end>` or the configured maximum number of turns (three in the reported experiments; the runner supports five). Planning and answer receive absolute F1. Search, summary, and update receive the marginal improvement $F_1(a_t) - F_1(a_{t-1})$ for their turn; a search end action receives zero task reward. Format penalties are added per node.

This is the M-ASK row in Table 4: the node term covers role tags and the legal end action, the turn term is the marginal F1 assigned to search, summary, and update, and the trajectory term is the absolute F1 assigned to planning and answer. The table also records that retrieval is a tool with retries and that either the end action or the turn limit masks later copies of the loop.

### F.3 REFLECTIVE CODE GENERATION

**Data and evaluation.** The code experiments use the released `agentica-org/DeepCoder-14B-Preview` data setting (Luo et al., 2025; Agentica, 2025). The training pool contains approximately 24K unique problem–test pairs compiled from TACO-Verified, PrimeIntellect SYNTHETIC-1, and LiveCodeBench (LCB) v5 problems dated 2023-05-01–2024-07-31. Our evaluation pool combines held-out LCB v5 and Codeforces problems from the corresponding evaluation collections. We execute both subsets with the same verifier and report the pooled all-passed rate: a problem is correct only when every available test passes. This is our workflow metric and should be distinguished from the DeepCoder model card's benchmark-specific Pass@1 and Codeforces rating.

The code workflow makes the role contract, model mapping, and delayed credit explicit. A sample initializes a state $S_0$ containing the problem, optional starter code, visible tests, and an empty iteration history. At stage $t$, the planner emits `<pseudocode>...</pseudocode>`, the coder consumes the problem and plan and emits `<code>...</code>`, and the verifier tool executes the candidate. Its observation contains the number of passed tests, total tests, pass rate $s_t$, error text, and an all-passed predicate. If the predicate is false and another stage remains, the reflector consumes the candidate and verifier feedback and emits `<reflection>...</reflection>`; that reflection is appended to $S_{t+1}$ and becomes part of the next planner prompt. Thus the bounded loop is represented by stage-indexed copies of these roles in the episode DAG, while early termination masks the unused suffix.

The independent configuration maps planner, coder, and reflector to three worker groups, whereas the shared configuration maps all three logical roles to one worker group. This choice is made in the YAML role-to-model map and leaves prompts, parsers, verifier calls, and stopping predicates unchanged. The verifier is always a non-trainable tool node. The code runner uses `CODE_STOP_ON_ALL_PASSED=true`: an all-passed candidate terminates before reflection; otherwise the maximum is three stages in the reported run. Figure 6 shows the per-task state transition and the two mapping regimes, and Figure 7 shows how many tasks and worker groups overlap during training.

Let $s_t$ be the verifier pass rate in stage $t$. The initial planner and coder receive the absolute score $s_0$; later planner/coder pairs receive the progress signal $s_t - s_{t-1}$; and the reflector that prepares stage $t + 1$ receives the same progress signal when that next stage is evaluated. A reflector at the final stage or immediately before early stopping receives zero task progress. Every trainable invocation additionally receives a node-level format/schema term: malformed tags or missing required fields receive the configured negative format score, while a legal but non-improving output receives zero task progress. The resulting composite reward is written back to the exact planner, coder, or reflector

entry after the verifier result is known. This is the concrete code instance of the general reward decomposition in Eq. 4.

The reflective-code row of Table 4 summarizes this contract. Planner, coder, and reflector are trainable roles; the executable verifier is a tool and therefore contributes an observation but no PPO entry. Parser and schema checks form the node reward, verifier pass-rate deltas form the turn reward, and the all-passed predicate controls termination rather than adding a second terminal bonus. Figure 6 visualizes the same row as a single realized DAG, while Figure 7 shows how its realized node records are routed into model-local buffers.

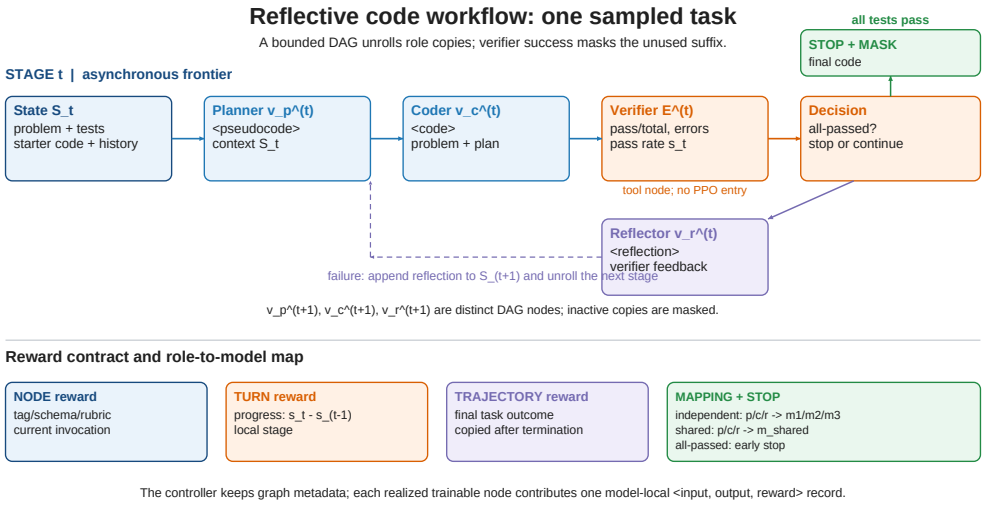


Figure 6: Reflective code workflow contract. Stage-indexed planner, coder, and reflector nodes form a bounded DAG around the executable verifier. The verifier supplies pass-rate progress and an early-stop predicate; parser/schema checks supply node-level rewards. The lower row shows the independent and shared role-to-model mappings.

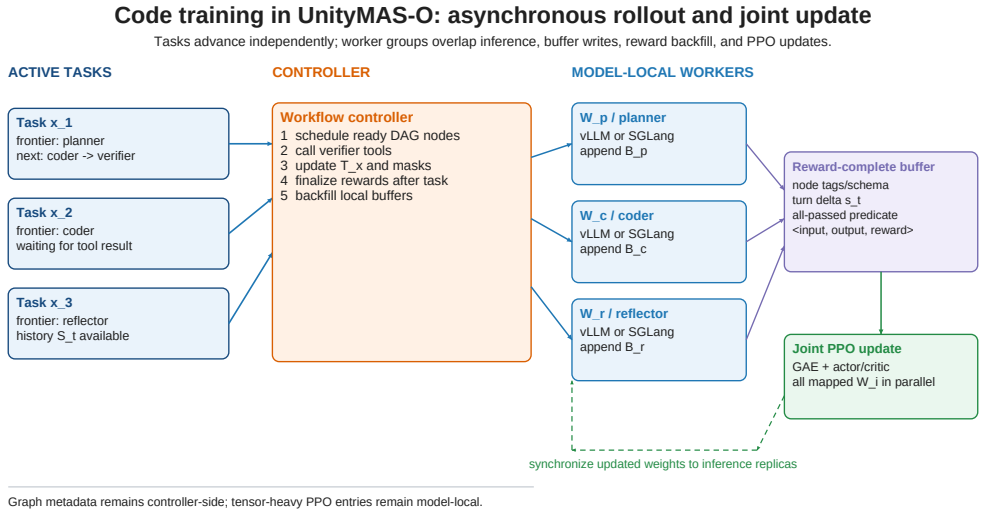


Figure 7: Concrete code-task rollout and update path. Independent tasks advance asynchronously through their ready frontiers; worker groups run vLLM/SGLang inference and append tensor-heavy entries to model-local buffers. After a task terminates, the controller backfills node and verifier-progress rewards before all worker groups perform PPO updates and synchronize inference replicas.

**Algorithm 2** Reflective Code Workflow Rollout and Joint Update

**Input:** Code-task stream $\mathcal{D}$, unrolled DAG $\mathcal{G}_{\text{code}}$, map $\phi$, verifier $E$, maximum stages $T_{\max}$
**Input:** Worker groups $\{W_i\}$, reward contract $\mathcal{R}_{\text{code}}$, rollout size $B$
**Output:** Completed model-local buffers and updated planner, coder, and reflector parameters

Initialize active task set $\mathcal{A}$, trace store $\mathcal{T}$, and buffers $\{\mathcal{B}_i\}$
Set $n_{\text{collected}} \leftarrow 0$
**while** $n_{\text{collected}} < B$ **do**
  Admit new tasks $x \sim \mathcal{D}$ into $\mathcal{A}$ and initialize $S_0(x)$ with problem, tests, starter code, and empty history
  **for all** active tasks $x \in \mathcal{A}$ asynchronously **do**
    Let $t$ be the next unmasked stage of $x$; dispatch planner $p^{(t)}$ to $W_{\phi(p)}$ with $S_t(x)$
    Dispatch coder $c^{(t)}$ to $W_{\phi(c)}$ with $S_t(x)$ and planner output; append both records and reward placeholders
    Execute verifier tool $E$; record pass rate $s_t$, errors, and all-passed flag in $\mathcal{T}_x$
    **if** all tests pass or $t = T_{\max} - 1$ **then**
      Mask unused role copies, retire $x$, and mark its trace reward-complete
    **else**
      Dispatch reflector $r^{(t)}$ to $W_{\phi(r)}$ and set $S_{t+1}(x) \leftarrow S_t(x) \cup \{r^{(t)}\}$
    **end if**
  **end for**
  For every retired $x$, compute node format terms, progress deltas, and terminal outcome with $\mathcal{R}_{\text{code}}$
  Backfill $\langle \text{input}, \text{output}, \text{reward} \rangle$ for each realized trainable node in its mapped $\mathcal{B}_i$ and increment $n_{\text{collected}}$ by its number of trainable nodes
  **if** a model-local buffer reaches a reward-complete minibatch **then**
    Enqueue asynchronous PPO work for that worker while other tasks continue rolling out
  **end if**
**end while**
**for all** worker groups $W_i$ in parallel **do**
  Form minibatches from $\mathcal{B}_i$; compute returns and GAE advantages
  Update actor and critic with PPO; synchronize updated weights to the inference replica
**end for**
Clear consumed entries and retain graph traces for diagnostics

### F.4 Configuration and implementation map

The concrete runners are implemented in `verl/experimental/star_ppo/workflows/`; reward allocators are in `verl/experimental/star_ppo/reward_allocators/`. The main experiment configurations are `star_query_decompose_retrieve{_evidence_answer}_f1_*`, `star_iterative_plan_search_summary_update_answer_f1_*`, and `star_code_iterative_plan_code_reflect_*.yaml`. These files specify the role-to-model mapping, turn limits, parser tags, retriever settings, timeouts, and early-stop switches. The public repository is intentionally not linked from the anonymous manuscript; it can be restored in the camera-ready version.

Table 4: Workflow contracts used by the experiments. The reward contract records node-level checks, turn-level progress, delayed trajectory outcomes, and the stopping rule. Tool nodes produce observations and never enter a PPO buffer.

| Workflow | Trainable roles | Tools and termination | Reward contract |
|---|---|---|---|
| QD–Answer | decompose; answer | Parallel top-$k$ retriever; stop at answer/end or empty frontier | Node: JSON sub-query and answer tags/schema. Turn: none. Trajectory: final answer F1 shared by realized roles. |
| QD–Evidence–Answer | decompose; evidence$_{1..3}$; answer | Retriever and context-builder join; stop at answer/end or empty frontier | Node: JSON, evidence-span, and answer tags/schema. Turn: none. Trajectory: final answer F1 shared by realized roles. |
| M-ASK | plan; search; summary; update; answer | Top-$k$ retriever with retries; stop on "end" or maximum turns | Node: role tags and valid end action. Turn: $F_1(a_t) - F_1(a_{t-1})$ for search/summary/update. Trajectory: absolute answer F1 for plan/answer. |
| Reflective code | planner; coder; reflector | Executable verifier; stop when all tests pass or the stage limit is reached | Node: pseudocode/code/reflection tags and required fields. Turn: verifier pass-rate delta $s_t - s_{t-1}$. Trajectory: no extra terminal bonus; all-passed is the early-stop predicate. |

# G POSITIONING AGAINST GENERAL LLM–MARL

The central distinction is the optimization contract. UnityMAS-O takes four objects as first-class inputs: an arbitrary user-defined DAG with asynchronous frontier scheduling, an explicit logical-role-to-physical-model map, node/turn/trajectory reward allocation, and model-local buffers containing every realized trainable invocation. The cited systems instead expose one or more of these through a fixed workflow, grouped rollout protocol, centralized multi-agent world, or optimizer-specific policy grouping. Table 5 compares these interfaces along topology and scheduling, optimization object, reward/credit, policy grouping, and rollout-to-update path. Concretely, graph-first execution preserves branches, joins, masks, and asynchronous frontiers; each realized invocation becomes one model-local buffer entry without requiring a same-state candidate group; $\phi$ independently controls sharing and physical placement; and node, turn, and delayed trajectory terms are backfilled to the invocations that produced them.

Table 5: Comparison with recent general LLM–MARL systems. Source evidence is listed as one benchmark/protocol item per line; "NR" means that no matched NQ/HotpotQA result is reported. These numbers are not cross-benchmark rankings.

| Method | Topology and optimized object | Reward / optimizer | Role–model relation and rollout grouping | Source evidence |
|---|---|---|---|---|
| MHGPO | Fixed multi-agent search (MASS); optimizes a shared backbone with heterogeneous-group updates | Global answer reward plus agent behavior rewards; critic-free IS/FoF/RR group-relative objectives | Agent/history groups retain cross-agent dependencies; reported Hotpot setup uses one Llama3.1-8B backbone | **HotpotQA (F1):** 21.45→50.86 (+29.41; FoF(os)) **MAPPO:** 46.40 |
| Dr. MAS | User-defined sequential, hierarchical, or conditional MAS; optimizes agent policies with agent-wise normalization | Per-agent mean/std normalization of outcome rewards addresses heterogeneous GRPO scales | Agent-aware groups with optional sharing/non-sharing and pooled actor backends | **NQ (avg@16):** shared: 45.2→47.4 non-shared: 27.1→47.7 **HotpotQA (avg@16):** shared: 40.3→42.5 non-shared: 24.4→44.0 |
| STRONGER-MAS | Fixed MAS workflows with single- or multi-policy training; optimizes AT-GRPO policies | Agent×turn grouped relative advantages combine global and local rewards | Role/turn groups route experience to policy batches | **LiveCodeBench:** 28.0→33.1 **APPS:** 44.4→46.5 **CodeContests:** 17.6→18.1 |
| MARTI | Centralized Multi-Agent World with distributed trainers; supports graph workflows and heterogeneous agents | Centralized reward allocation/shaping with PPO, GRPO, or REINFORCE++ variants | World coordinates agents while distributed trainers own policy resources | **AIME:** DeepScaleR-1.5B 53.5→65.0 (+11.5) |
| HACRL | Heterogeneous cooperative roles with verified rollouts; optimizes policies from cooperative outcomes | Verified rollout feedback is shared during training; deployment can remain independent | Training shares verified data while execution preserves independent agents | **Math-only v2:** Qwen3-4B-base: 57.8→59.7 (+1.9) Llama3.2-3B-Instruct: 35.1→39.0 (+3.9) |
| UnityMAS-O | Arbitrary user DAG with asynchronous frontiers, branches, joins, conditionals, and bounded loop unrolling; optimizes mapped role policies | Node, turn, and trajectory rewards are committed to the exact realized node; PPO/GAE uses model-local buffers | Explicit $\phi$ supports full, partial, or separate models; one local record per realized trainable invocation | **QA (F1):** NQ 62.10; HotpotQA 57.30 Qwen2.5-7B **Code (all-passed):** 0.738 Qwen3-8B |

## G.1 GROUPED-ROLLOUT COMPLEXITY AND THE GRAPH-FIRST ALTERNATIVE

GRPO-style methods (Shao et al., 2024) form a relative baseline by sampling multiple candidates for a common prompt or state. In a multi-agent workflow, later prompts depend on earlier sampled outputs. Let $R_x = |\mathcal{V}_{\mathrm{LLM}}(\tau_x)|$ be the number of realized trainable invocations in one episode and let $G_j$ be the number of same-history candidates required at invocation $j$. A fully nested counterfactual rollout tree has

$$N_{\mathrm{leaf}} = \prod_{j=1}^{R_x} G_j, \qquad N_{\mathrm{leaf}} = G^{R_x} \text{ when } G_j = G. \tag{8}$$

For $N$ roles repeated over $T$ stages this gives the familiar worst-case upper bound $G^{NT}$, with multiplication only along branches whose downstream prompts depend on the sampled predecessor.

This is a worst-case credit-assignment accounting, not a claim that every GRPO implementation literally enumerates the full tree. Sampling only $G$ complete trajectories costs $O(GR_x)$, but then the candidates for an individual role generally have different upstream histories and no longer form a same-state group. AT-GRPO, MHGPO, and Dr. MAS address this trade-off by factorizing the comparison into agent/turn or heterogeneous groups; the factorization avoids explicit Cartesian enumeration but retains role–turn–candidate grouping and its associated storage and normalization state (Chen et al., 2026a; Feng et al., 2026; Zhao et al., 2026).

UnityMAS-O chooses a different contract. The controller samples one realized asynchronous DAG execution, records one token-level invocation for each trainable node, and backfills node, turn, and delayed trajectory rewards after the task terminates. The model-local buffers therefore contain $O(R_x)$ realized records per task, and a batch costs $O(\sum_x R_x)$ records under explicit DAG materialization. PPO/GAE does not require same-state candidate groups or a joint cross-role candidate tensor. This removes a source of rollout-dimension and memory growth while preserving exact graph dependencies in the trace; the trade-off is that the relative-group variance reduction of GRPO is not used. The design is therefore especially useful when roles have different prompts, output lengths, or stopping times, as in the code workflow.

The comparison therefore exposes three modeling consequences. First, GRPO-style rollout becomes difficult to scale when later prompts depend on earlier outputs: preserving same-state candidate groups across several roles and stages creates a counterfactual product of role, turn, and candidate choices, while sampling only complete trajectories breaks the common-state assumption needed for a clean relative advantage. Grouping or factorizing the candidates avoids explicit Cartesian enumeration, but introduces additional agent/turn bookkeeping and normalization state. Second, under the standard GRPO contract, the advantage is computed from a relative normalization of a trajectory-level final score. That delayed scalar cannot distinguish whether progress came from a particular node, turn, or model, and role- or turn-level shaping terms do not by themselves resolve the credit-assignment ambiguity when the useful contribution is visible only after later verification. Third, PPO-style updates can store every realized trainable invocation separately and backfill node-level, turn-level, and delayed trajectory rewards before computing advantages. The resulting model-local buffers can therefore optimize multiple trajectories from one task, including trajectories produced by different roles or physical models, while the graph trace preserves their dependencies. This comparison motivates the graph-first, buffer-local design of UnityMAS-O and explains why it is better aligned with asynchronous heterogeneous workflows than a single GRPO grouping contract.

## H Full co-training sweep and computational resources

Table 6 reports the complete independent-policy sweep and its hardware allocation. The before–after scores complement the effectiveness analysis in Sec. 5; here we focus on how computation is assigned to the physical LLMs.

**Compute allocation.** All experiments in Table 6 use independent agent policies, so the Agent/LLM column also gives the number of physical LLM worker groups. QD–Answer has two trainable roles and therefore uses two one-node LLM groups (two nodes, 16 H800 GPUs); QD–Evidence–Answer has three roles and uses three nodes (24 GPUs); M-ASK has four roles and uses four nodes (32 GPUs). The reflective-code workflow uses three independent LLMs for planner, coder, and reflector. In the 3×4B run, each LLM receives two nodes (16 GPUs), yielding six nodes and 48 GPUs overall. In the 3×8B run, each LLM receives three nodes (24 GPUs), yielding nine nodes and 72 GPUs overall.

**Configurable capacity per LLM.** The role-to-model map $\phi$ and the compute allocation are separate configuration choices. For $K$ physical models, UnityMAS-O accepts an independently configured node count $n_i$ for each worker group $W_i$, giving $N_{\text{nodes}} = \sum_{i=1}^{K} n_i$ and, on this cluster, $N_{\text{GPU}} = 8 \sum_{i=1}^{K} n_i$. Each LLM can therefore span an arbitrary configured number of compute nodes, subject to available cluster resources and the backend's parallelism requirements. The reported runs instantiate one-, two-, and three-node worker groups within the same controller–worker interface. Changing $n_i$ adjusts the resources available for that model's inference and parameter updates while

Table 6: Full co-training sweep and hardware allocation. All rows use independent agent policies, so the number of trainable agents equals the number of physical LLMs shown in the Agent/LLM column. QA reports validation F1 with Qwen2.5; code reports pooled LCB v5 + Codeforces all-passed with Qwen3. Each compute node contains eight NVIDIA H800 GPUs. Nodes is the total number of nodes assigned to the experiment; the per-LLM allocation is given in the text below. Scores and gains use the 0–1 scale; Relative is the gain divided by Before. QA gains retain source precision before rounding.

| Dataset | Workflow | Size | Agent/LLM | Before | After | Gain | Relative | Nodes |
|---|---|---|---|---|---|---|---|---|
| NQ | QD–Answer | 0.5B | 2 | 0.022 | 0.445 | +0.424 | +1943% | 2 |
| | | 1.5B | 2 | 0.073 | 0.508 | +0.435 | +599% | 2 |
| | | 3B | 2 | 0.266 | 0.535 | +0.269 | +101% | 2 |
| | | 7B | 2 | 0.315 | 0.536 | +0.221 | +70% | 2 |
| | | 14B | 2 | 0.328 | 0.594 | +0.267 | +81% | 2 |
| | QD–Evidence–Answer | 0.5B | 3 | 0.051 | 0.429 | +0.379 | +746% | 3 |
| | | 1.5B | 3 | 0.070 | 0.480 | +0.410 | +588% | 3 |
| | | 3B | 3 | 0.107 | 0.473 | +0.366 | +342% | 3 |
| | | 7B | 3 | 0.279 | 0.509 | +0.231 | +83% | 3 |
| | | 14B | 3 | 0.279 | 0.575 | +0.295 | +106% | 3 |
| | M-ASK | 0.5B | 4 | 0.015 | 0.142 | +0.127 | +842% | 4 |
| | | 1.5B | 4 | 0.132 | 0.515 | +0.383 | +290% | 4 |
| | | 3B | 4 | 0.259 | 0.513 | +0.253 | +98% | 4 |
| | | 7B | 4 | 0.260 | 0.621 | +0.361 | +139% | 4 |
| HotpotQA | QD–Answer | 0.5B | 2 | 0.032 | 0.397 | +0.366 | +1161% | 2 |
| | | 1.5B | 2 | 0.069 | 0.493 | +0.424 | +612% | 2 |
| | | 3B | 2 | 0.162 | 0.493 | +0.332 | +205% | 2 |
| | | 7B | 2 | 0.284 | 0.523 | +0.240 | +85% | 2 |
| | | 14B | 2 | 0.306 | 0.542 | +0.236 | +77% | 2 |
| | QD–Evidence–Answer | 0.5B | 3 | 0.068 | 0.359 | +0.291 | +430% | 3 |
| | | 1.5B | 3 | 0.071 | 0.435 | +0.364 | +514% | 3 |
| | | 3B | 3 | 0.107 | 0.473 | +0.366 | +342% | 3 |
| | | 7B | 3 | 0.235 | 0.486 | +0.251 | +107% | 3 |
| | | 14B | 3 | 0.279 | 0.521 | +0.243 | +87% | 3 |
| | M-ASK | 0.5B | 4 | 0.030 | 0.226 | +0.196 | +648% | 4 |
| | | 1.5B | 4 | 0.127 | 0.525 | +0.398 | +313% | 4 |
| | | 3B | 4 | 0.201 | 0.529 | +0.327 | +162% | 4 |
| | | 7B | 4 | 0.254 | 0.573 | +0.319 | +126% | 4 |
| Code | Reflective verification | 3×4B | 3 | 0.255 | 0.686 | +0.431 | +169% | 6 |
| | | 3×8B | 3 | 0.290 | 0.738 | +0.448 | +154% | 9 |

preserving the workflow DAG, reward allocation, and model-local buffer semantics. This separation lets users allocate capacity according to each agent's model size and workload.

# I BASELINE DESCRIPTIONS

**Single-agent and retrieval baselines.** The direct language-model and vanilla-RAG rows use one answer policy with either no retrieval or a fixed retriever. RRR is the retriever-oriented query-rewriting method of Ma et al. (Ma et al., 2023), while BGM bridges retriever and generator preferences during retrieval training (Ke et al., 2024). Search-R1 applies single-agent RL to tool-augmented search (Jin et al., 2025).

**MMOA-RAG and MAO-ARAG.** MMOA-RAG trains the roles of a fixed retrieval-and-answer pipeline (Chen et al., 2025a). MAO-ARAG trains a planner to select and compose retrieval, query-reformulation, document-selection, and answer-generation modules; its reported executor models remain frozen while the Qwen2.5-7B planner is optimized with an answer-and-cost objective (Chen et al., 2025b).

**M-ASK and JADE.** M-ASK optimizes repeated search, summary, update, and answer roles in an iterative search workflow (Chen et al., 2026b). JADE jointly trains a planner and specialized executors with a shared Qwen2.5-7B backbone; its Document Selector remains in the workflow in both the frozen and trained conditions (Chen et al., 2026c).

**MHGPO.** MHGPO optimizes a multi-agent search system with relative advantages over heterogeneous rollout groups and without a learned critic (Chen et al., 2026a). Its reported comparison contrasts the unoptimized search workflow with the FoF(os) optimizer under the source Llama3.1-8B protocol.

**Dr. MAS.** Dr. MAS normalizes each agent's reward statistics before forming relative advantages and supports shared or independent policies (Feng et al., 2026). The QA deltas in Table 3 keep the non-shared configuration fixed and replace vanilla GRPO with agent-wise normalization; both endpoints are trained systems and use the source avg@16 metric.

**STRONGER-MAS.** STRONGER-MAS introduces Agent- and Turn-wise GRPO (AT-GRPO), which forms local comparison groups for a role and interaction state and combines team rewards with role-specific feedback (Zhao et al., 2026). Its code workflow alternates coding and test-generation roles; the table compares its prompted MAS condition with trained per-role policies on the reported source benchmarks.

**MARTI and HACRL.** MARTI studies centralized multi-agent worlds with distributed trainers and supports several policy optimizers (Zhang et al., 2026a). HACRL trains heterogeneous cooperative roles with verified rollouts and shared training feedback while preserving independent execution policies (Zhang et al., 2026b). Neither reports a matched NQ/HotpotQA or pooled-code protocol in the comparison table.