FlexMARL: Graph-Driven Rollout-Training Co-Design for LLM-Based Multi-Agent Reinforcement Learning

ATC 2026 Submission #2173

Abstract

Despite algorithm-level innovations for multi-agent reinforcement learning (MARL), the underlying infrastructure for large-scale MARL remains underexplored. Existing training frameworks primarily optimize for single-agent scenarios and fail to capture the graph-structured nature of MARL. This fundamental mismatch leads to three MARL-specific challenges: topology-induced rollout skew, critical path synchronization barriers, and dynamic training demand. To bridge this gap, we propose FlexMARL, a graph-driven rollout-training co-designed framework for large-scale LLM-based MARL. FlexMARL introduces the multi-agent execution graph (MEG) and materializes the intermediate representation as the *MEG Store*, which provides decentralized multi-table storage and serves as the data hub for system components. Built around the MEG Store, the *Rollout Engine* develops dependency parallel sampling with hierarchical load balancing, which reduces inference latency and balances intra-agent/inter-agent loads. The *Joint Orchestrator* exploits an agent-level asynchronous pipeline to mitigate the critical path synchronization barriers while maintaining consistency. The *Training Engine* achieves on-demand resource allocation by collectively managing training processes for individual agents. The training states of different agents are swapped via location-agnostic communication APIs. Empirical results on a production cluster demonstrate that FlexMARL achieves up to 7.3× speedup and improves hardware utilization by up to 5.6× compared to existing frameworks.

CCS Concepts: • **Computing methodologies** → **Machine learning**; **Distributed computing methodologies**.

Keywords: Multi-Agent Reinforcement Learning, Large Language Models, Distributed Training

ACM Reference Format:
ATC 2026 Submission #2173. 2026. FlexMARL: Graph-Driven Rollout-Training Co-Design for LLM-Based Multi-Agent Reinforcement Learning. In *Proceedings of The 2026 ACM SIGOPS Annual Technical Conference (ATC 2026).* ACM, New York, NY, USA, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Recent advances in large language models (LLMs) have facilitated the development of multi-agent systems that leverage specialized roles and distributed reasoning to solve complex tasks [16, 21, 45]. Concurrently, reinforcement learning (RL) has emerged as a powerful paradigm for improving the training performance of LLMs [13, 51]. To harness RL's policy optimization capabilities for strengthening multi-agent systems, multi-agent reinforcement learning (MARL) enables a collective of agents to learn coordinated policies through shared experiences [37, 50]. Such a learning paradigm continuously collects experience generated by multi-agent interactions, integrates feedback from reward signals, and refines models to unleash collaborative intelligence [14, 53, 59].

Despite the impressive performance gains brought by MARL, most research pays more attention to algorithm-level innovations, such as communication protocol [43, 56], coordination optimization [25, 32], and reward function refinement [3, 42]. The underlying networked infrastructure that supports large-scale MARL training remains underexplored. Mainstream RL frameworks (e.g., OpenRLHF [18], veRL [35], and AReaL [10]) only support single-agent scenarios, and even recent MARL frameworks represented by MARTI [53] are customized for specialized MARL algorithms and incremental extensions of single-agent RL. They naively optimize the rollout or training phase and lack end-to-end optimization considering the characteristics of MARL. Besides, their implementation fails to support cross-node placement for individual agents, and LLM-based agents are confined to limited parameters (e.g., 3B), impeding large-scale deployment.

Through profiling of real-world MARL workloads, we identify three MARL-specific challenges that differ fundamentally from those of general agentic RL. *(1) Topology-induced rollout skew.* Rollout request scheduling exhibits a severe imbalance among agents during parallel trajectory generation. Core agents appearing on the critical paths are repeatedly involved and incur request queuing, while auxiliary agents remain underutilized. *(2) Critical path synchronization barriers.* MARL amplifies long-tail latency through multi-agent interaction trajectories. Policy training has to wait for the slowest path to complete, leaving expensive compute resources idle. *(3) Dynamic training demand.* Different subsets of agents could be activated for policy training based on the collected experience. The computing and memory resources occupied by inactive agents are wasted under static allocation strategies, resulting in poor resource utilization.

Our key insight is that these challenges are not independent issues but stem from the graph-structured nature of MARL. The fundamental execution unit is not an isolated request or agent, but a dependency structure formed by agent invocations, outputs, and policy-versioned experiences [47]. Existing RL frameworks rely on uniform scheduling and static resource allocation during both rollout and training phases, which do not jointly optimize the above

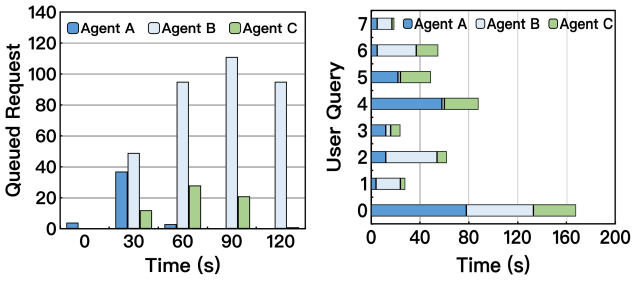
MARL-specific challenges based on the dynamic and collaborative characteristics of MARL. This motivates us to design an intermediate representation that makes the multi-agent dependency structure explicit and uses it to guide rollout, orchestration, and training of MARL.

To fill this gap, we propose an end-to-end MARL training framework, termed FlexMARL, that co-designs rollout, training, and their orchestration for large-scale LLM-based MARL. FlexMARL introduces the multi-agent execution graph (MEG) and materializes the graph representation as the *MEG Store*. Each vertex represents an agent invocation, and each edge represents a data dependency. During rollout, FlexMARL uses these dependencies to decide which downstream agents can be executed. During training, FlexMARL groups vertices by agent and policy version, and dispatches them for independent policy optimization. The MEG Store provides decentralized multi-table storage and serves as the data hub for the following three components.

The *Rollout Engine* employs dependency parallel sampling that enables concurrent trajectory generation as soon as upstream dependencies are satisfied. To address topology-induced rollout skew, the Rollout Engine monitors multi-agent workloads and then adopts hierarchical load balancing to flexibly adjust the number of inference instances across agents. The *Joint Orchestrator* adopts the rollout-training disaggregated architecture and decomposes the global batch at the granularity of agents. The agent-level asynchronous pipeline hides critical path rollout behind training through incremental gradient caching and unified policy updating, preserving multi-agent policy version consistency. The *Training Engine* exploits sparse training activation of MEG by collectively managing all training processes for an individual agent. Hardware resources are allocated to active agents only where and when needed, thus adapting to dynamic training demands and improving resource utilization. We abstract multi-tier pathways into a location-agnostic `Set/Get` API, which are used to efficiently swap the training states of suspended and resumed agents.

During each step, the Rollout Engine generates multi-agent trajectories via parallel sampling, with inference instances elastic scaling. These trajectories are collected into the MEG Store, where the Joint Orchestrator dispatches micro batches asynchronously to the Training Engine. The Training Engine allocates resources on demand and swaps states for policy training. The updated models are synchronized back to inference instances to start the next step. Overall, we make the following contributions in this paper:

- We identify three MARL-specific challenges: topology-induced rollout skew, critical path synchronization barriers, and dynamic resource demand, which can be attributed to the graph-structured nature of MARL. Based on this insight, we introduce the MEG and materialize the graph representation as the MEG Store.



(a) Queued Request (b) Query Response Time

Figure 1. Preliminary experiments demonstrating the MARL characteristics.

- We propose FlexMARL, the first framework to provide end-to-end optimization based on unique characteristics of large-scale LLM-based MARL. Built around the MEG Store, FlexMARL designs dependency parallel sampling with hierarchical load balancing for rollout, agent-level asynchronous pipeline for orchestration, and on-demand resource allocation for training.
- We implement and deploy FlexMARL on a production cluster. Experimental results for real-world e-commerce multi-agent assistants show that FlexMARL provides up to 7.3× speedup and improves hardware utilization by up to 5.6× compared to existing MARL frameworks.

2 Background and Motivation

2.1 Multi-Agent Reinforcement Learning

Traditional non-LLM MARL is largely restricted to well-structured environments with predefined action sets [1, 28, 54]. In contrast, LLM-based MARL represents a paradigm shift from low-level control optimization to knowledge-driven, high-level collaboration [48]. The agents are instantiated as LLMs with inherent natural language understanding and reasoning capabilities. These LLM agents process unstructured information and generate token sequences as actions. As a result, LLM-based MARL effectively decomposes task complexity, enabling agents to learn collaboratively through interactions with other agents [26, 63]. Recent advancements have highlighted its effectiveness in solving long-horizon tasks such as code generation [31, 60], mathematical reasoning [23, 55], and network management [40, 44], which are challenging for conventional neural network policies.

The typical workflow of LLM-based MARL contains repeated cycles across rollout and training phases. In the rollout phase, each LLM agent generates responses based on its current policy parameters, role-specific prompts, and environmental observations [59]. Multi-agent interactions produce the trajectories that capture their reasoning processes. These experiences are then evaluated using a rule-based or learned reward model that reflects task correctness, collaboration effectiveness, and alignment with global objectives. After credit assignment across agents, the collected experiences

Table 1. Comprehensive comparison: FlexMARL vs. Existing RL/MARL frameworks.

Frameworks	Basic Features			End-to-End Optimization		
	Multi-Agent Support	Algorithm Decoupling	Large-Scale Deployment	Topology-Induced Rollout Skew	Critical Path Sync Barriers	Dynamic Training Demand
General-Purpose RL Frameworks (e.g., OpenRLHF [18], veRL [35], AReaL [10])	✗	✓	✓	✗	✓	✗
Specialized MARL Frameworks (e.g., MARTI [53])	✓	✗	✗	✗	✗	✗
FlexMARL (Ours)	✓	✓	✓	✓	✓	✓

are organized into batches and dispatched to corresponding agent policies for training [19]. Each agent independently optimizes its policy model using RL algorithms (e.g., GRPO [34]) to maximize the expected cumulative reward. The updated models are synchronized via device-to-device (D2D) communication for the next step of rollouts.

2.2 Observations of MARL Characteristics

Despite the advancements in MARL algorithms, the underlying infrastructure bottlenecks hinder large-scale training efficiency. To reflect the characteristics of LLM-based MARL, we conduct extensive profiling of real-world multi-agent merchant assistant workloads. The setup of preliminary experiments is similar to that in Section 9. Three observations are as follows.

Observation #1: MARL exhibits imbalanced rollout load distributions across agents. A salient characteristic of MARL is that agents are typically assigned different roles due to task-specific collaboration mechanisms. During the experience collection phase, MARL needs to handle a large number of continuous rollout requests, exhibiting different interaction patterns based on functional roles. The rollout load among distinct agents shows a highly skewed distribution. We select three representative agents and track the number of their queued requests over time. As illustrated in Figure 1(a), a small portion of core agents are repeatedly invoked, handling over 76% of the total rollout requests, whereas other auxiliary agents process less than 24% of the requests. Such an imbalanced workload causes the core agents' requests to continuously queue, which prolongs the trajectory sampling time and becomes a bottleneck.

Observation #2: MARL suffers from long-tail multi-agent interaction latency. Due to differences in task complexity and agent decision paths, multi-agent interaction latency exhibits a pronounced long-tail distribution [22]. As shown in Figure 1(b), most multi-agent interactions are completed in a short time, but the longest response time of user queries can reach nearly 170s, which dominates the overall experience collection time. Under synchronous training paradigms, policy updating is blocked by "stragglers" due to strict serial dependency. Policy training is triggered until all trajectories are collected, causing rollout-training synchronization barriers.

Observation #3: MARL demonstrates dynamic resource demands during policy training phases. The policy training phase may activate different subsets of agents for gradient computation based on the collected samples. If we allocate dedicated resources to each agent using a fixed mapping strategy, the computing and memory resources occupied by inactive agents are wasted rather than reallocated to active agents with high current demand. Our preliminary experiments also confirm that the static allocation strategy lacks a global perspective, resulting in average hardware utilization of only 18.8% during policy training.

2.3 Key Insights

The above three observations reveal that large-scale LLM-based MARL introduces new system challenges that cannot be addressed by naively extending single-agent RL infrastructure. We realize that the fundamental execution unit of MARL is not an isolated request or an isolated agent, but a set of agent invocations (vertices) connected by data dependencies (edges). This graph dependency structure exposes and amplifies the observed challenges.

Revisiting Observation #1: Rollout load imbalance is induced by graph topology. Load is unevenly distributed across different graph vertices. Core agents correspond to high-degree vertices that appear on the critical paths of many trajectories, while auxiliary agents correspond to low-degree vertices invoked by only a few paths. Request-based scheduling cannot resolve inherent skew caused by graph topology.

Revisiting Observation #2: Synchronization barriers stem from graph critical paths. The long-tail distribution reported in Figure 1(b) is explained as the dispersion of path lengths across the graph. The overall rollout time of multi-agent interactions is determined by the longest path in the graph (critical path), not by individual agent invocations. Under batch-level synchronization paradigms, policy optimization has to wait for the slowest dependency path to complete, inducing severe pipeline bubbles.

Revisiting Observation #3: Low resource utilization arises from sparse subgraph activation. Only subgraphs corresponding to agents with sufficient accumulated experience are activated for policy optimization. The active vertex groups are sparse and change per training step. Static allocation strategy wastes resources on agents that are present in MARL workloads but inactive in the current subgraph.

2.4 Limitations of Existing Works

Existing RL frameworks generally fall into two categories, neither of which performs end-to-end optimization based on the graph structure nature of MARL. On the one hand, general-purpose RL frameworks (such as OpenRLHF [18], AReaL [10], and veRL [35]) are primarily designed for a single model and cannot capture cross-agent dependencies, agent load skew, or dynamically activated training subsets in LLM-based MARL. On the other hand, specialized MARL frameworks [8, 53, 59] only provide basic multi-agent training capabilities but treat the underlying infrastructure as ad-hoc extensions of single-agent RL, without end-to-end and MARL-specific optimization. As a result, they rely on uniform scheduling and static resource allocation during both rollout and training phases. Such designs suffer from load imbalance when rollout workloads are skewed across heterogeneous agents, induce synchronization barriers when long-tail dependency paths prolong batch completion, and waste compute/memory resources when only a subset of agents accumulate sufficient training samples. Furthermore, their implementations lack cross-node deployment support and are limited to small-scale models. For instance, MARTI [53] is deployed on a 3-node cluster, and the largest model participating in MARL is Qwen2.5-3B. Such constrained configurations prevent the exploration of performance gains from larger agent ensembles and more powerful models, thereby limiting the scalability of MARL.

As summarized in Table 1, existing RL frameworks fail to simultaneously address the challenges exhibited by large-scale MARL training, including topology-induced rollout skew (Observation #1), critical path synchronization barriers (Observation #2), and dynamic training demand (Observation #3). This reveals the gaps between algorithmic requirements and existing infrastructure support.

3 System Overview

Motivated by the graph-structured nature of MARL, we propose FlexMARL, a graph-driven rollout-training co-designed framework that provides end-to-end optimization for large-scale LLM-based MARL. As illustrated in Figure 2, FlexMARL consists of four tightly integrated components:

• **MEG Store (§4).** FlexMARL uses the MEG as an intermediate representation and materializes the graph in the MEG Store, which organizes agent invocations, data dependencies, policy versions, and training status. As the concrete runtime materialization, the MEG Store employs decentralized multi-table storage and serves as the data hub connecting system components by supporting incremental experience generation during rollout and efficient retrieval during training.

• **Rollout Engine (§5).** To improve inference efficiency, our engine employs a dependency parallel sampling scheme that schedules rollout requests according to satisfied upstream dependencies, allowing different trajectory branches

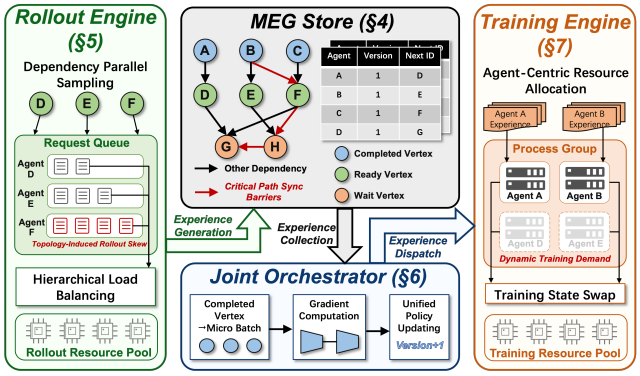


Figure 2. Overview of FlexMARL.

to progress concurrently. Based on request queues, hierarchical load balancing elastically adjusts the number of intra-agent/inter-agent inference instances via D2D transmission, thus alleviating topology-induced rollout skew.

• **Joint Orchestrator (§6).** The orchestrator adopts the disaggregated architecture to allocate dedicated resource pools and overlaps rollout and training at the agent granularity. Instead of blocking training until the rollout of MEG completes, it decouples gradient computation from policy update. Policy training is triggered incrementally for each agent as soon as part of the experience is available in the MEG Store, thereby eliminating the synchronization barriers caused by the long-tail critical path.

• **Training Engine (§7).** Since the agents' experiences become ready at non-deterministic times, different subgraphs of MEG are activated during training. FlexMARL exploits this sparsity by agent-centric resource allocation. When an agent requires policy optimization, its process group is dynamically scheduled onto available resources. We further implement efficient state swap mechanisms that transfer training states (e.g., weights and optimizer states) between device and host memory, adapting to large-scale scenarios where a large number of agents need to be updated.

The collaboration process of the four components is described as follows. The Rollout Engine continuously retrieves agent invocation tasks whose all upstream dependencies have been satisfied, dispatches these tasks to elastic inference instances, and writes back the output responses to the MEG Store. This process triggers new downstream agents incrementally until the complete trajectory is generated. The agent-level experiences are also collected in the MEG Store with version tracking. The Joint Orchestrator dispatches micro batches of experiences to the Training Engine with ongoing rollouts. Based on available cluster capacity, the Training Engine concurrently allocates hardware resources for active agents to perform policy optimization. After processing a full batch, accumulated gradients are aggregated for unified parameter updates. Finally, new policy models are synchronized to inference instances via D2D interconnects.

4 MEG Store

4.1 Graph Representation

Existing RL systems typically organize experience as flat trajectories or batches. However, LLM-based MARL differs from single-agent RL in a structural way. MARL contains multiple role-specialized agents whose invocations depend on outputs from other agents, and some branches may proceed in parallel. A flat trajectory representation makes it difficult to exploit the topology-induced rollout skew, critical path synchronization barriers, and dynamic training demand. To this end, FlexMARL models MARL workloads using the MEG, which is a directed graph $(\mathcal{N}, \mathcal{E})$. Each vertex $n \in \mathcal{N}$ is one agent invocation with specific policy version V_n, input prompt I_n, output response O_n, and reward R_n. Each edge $e_{n,m} \in \mathcal{E}$ represents a data dependency, meaning that the downstream agent m can be executed only after the upstream agent n has produced the response. The directed property follows the execution semantics of MARL workloads. An agent invocation consumes previously generated context and produces a new output for downstream agents. MEG handles learning feedback across training steps through updated policy versions.

The MEG supports two views over the same set of vertices. In the *rollout view*, vertices execute in topological order, and edges enforce data dependencies between agents. They determine when a agent becomes ready and which downstream agents can be invoked. Different vertices have different out-degrees and in-degrees, indicating a skewed rollout load. The critical path dominates the rollout latency of multi-agent interactions. In the *training view*, the completed vertices are grouped by agent identifier and policy version. The edges no longer impose execution dependencies because policy optimization is performed independently for each agent after experience collection and reward assignment. Training activation is sparse and dynamic since only agents that have sufficient experience perform policy optimization.

4.2 Multi-Table Materialization

FlexMARL materializes the above graph representation through the MEG Store, which supports MARL's requirements for data isolation, cross-agent dependency, and policy version tracking. As illustrated in Figure 3, the MEG Store is organized by the decentralized multi-table storage at the granularity of graph vertices $n \in \mathcal{N}$, avoiding single-point bottlenecks. Each row contains three categories of columns: meta-information columns, data columns, and status columns. Specifically, meta-information columns store essential metadata for each sample, including the agent identifier n, policy version V_n, next agent identifier m, sample identifier, and a processing flag indicating whether a sample has been read but not yet updated. The next agent identifier m represents the dependency edge $e_{n,m} \in \mathcal{E}$ in MEG. The sample identifier ensures global uniqueness and traceability of

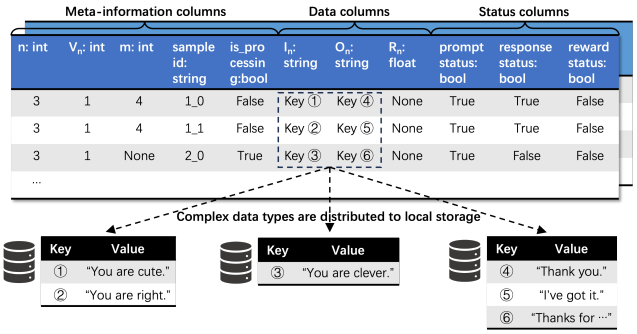


Figure 3. Multi-table organization of the MEG Store.

the form "`sample_id = {input_id}_{trajectory_id}`", where `input_id` denotes a globally incrementing identifier for each sample and `trajectory_id` is the index of the response within a trajectory. Data columns contain user-defined fields that store sample content, such as input prompt I_n, output response O_n, and reward R_n. Each data column is paired with a boolean status column that specifies whether its corresponding value has been fully generated.

To support lightweight querying while accommodating extensible user-defined fields across MARL algorithms, the MEG Store adopts type-aware hybrid storage that selects value-based or reference-based storage based on data type. All meta-information and status fields use simple data types (including int, float, and bool), which are stored by value for fast in-table access. Complex data types (including strings, lists, and tensors) are stored by reference, and the table only records the location key. FlexMARL fetches the keys from the table and then retrieves the corresponding values directly.

5 Rollout Engine

5.1 Dependency Parallel Sampling

The typical processing of rollout requests follows a strictly sequential execution model. Each agent may generate additional candidate samples for the same prompt. These samples are then further propagated through downstream agents to generate complete multi-agent trajectories. The next trajectory can be generated only after the entire rollout of the current trajectory has finished [57]. While this design preserves clear execution semantics, it severely limits hardware utilization and system throughput, especially in large-scale MARL settings.

To eliminate these inefficiencies, our Rollout Engine introduces dependency parallel sampling. Once the upstream dependencies of a rollout request have been satisfied, it can be dispatched to an inference instance of downstream agents immediately. Other queries or branches are independent of the completion state of the current query. In other words, multiple queries and/or multiple trajectory generation for the same query are allowed to execute rollout in parallel. After the agent n generates a response, the Rollout Engine writes

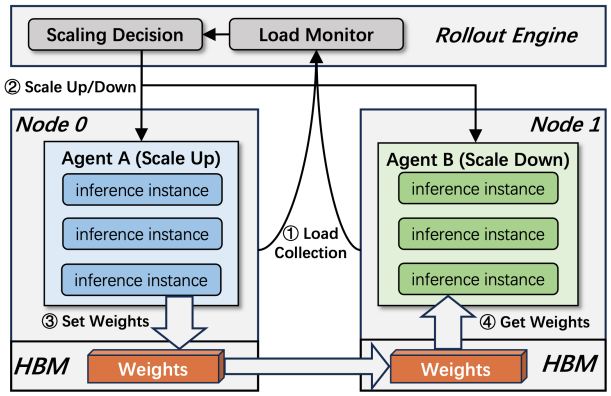


Figure 4. Inter-agent load balancing in the Rollout Engine.

the output O_n and the corresponding status fields back to the MEG Store. The downstream agents are notified through the next agent identifier m. The MEG Store is used not only to collect the final trajectories, but also to mediate intermediate data movement between agents. This procedure continues until complete multi-agent interaction trajectories are generated. Dependency parallel sampling converts the experience generation from a query-level blocking model into a fully parallelized scheduling model. All rollout requests compete dynamically for inference resources subject to data dependencies without unnecessary waiting, thereby significantly improving rollout efficiency.

5.2 Hierarchical Load Balancing

Dependency parallel sampling increases available query parallelism, but it also makes topology-induced load skew more visible. Due to agents' functional specialization, core agents may appear on many dependency paths and exhibit large in-degree/out-degree in the MEG. In contrast, auxiliary agents are rarely invoked, resulting in persistent queuing delays at overloaded agents and underutilization at others [47]. To mitigate load imbalance, our Rollout Engine provides a two-tier elastic scaling that balances rollout load both within and across agents.

Intra-Agent Load Balancing. For each agent, we deploy multiple inference instances to handle its concurrent rollout requests. The Rollout Engine employs a min-heap data structure of request queues to track the instantaneous load of backend inference instances. A new rollout request will be dispatched to the instance with the lowest current load. This greedy scheduling scheme achieves balanced request distribution within inference instances of individual agents. Moreover, the Rollout Engine ensures fault tolerance by removing completed requests, canceling timed-out ones, and re-queuing unfinished ones.

Inter-Agent Load Balancing. While intra-agent balancing mitigates contention within a single agent, it cannot resolve resource starvation across agents with heterogeneous demands. As shown in Figure 4, the Rollout Engine polls the request queue lengths across all agents. We analyze these load metrics and then compute the load disparity between the highest and lowest load agents. When this disparity exceeds a configurable threshold Δ, a scaling operation is triggered to migrate inference capacity from underutilized agents to overloaded ones [17]. Specifically, the number of migrated instances equals the difference in queue lengths, subject to the constraint that each agent retains at least one active inference instance to preserve liveness. The conservative strategy prevents transient load oscillation while ensuring service availability for all agents. Migrating inference capacity requires rapid model weight transfer to newly allocated instances. FlexMARL abstracts multi-tier pathways into a unified and location-agnostic `Set/Get` API, which hides the complexities of D2D, H2D, and D2H communication. When a scaling operation is triggered, the agents requiring scaling up (Agent A) publish model weights via `Set` API, while the reallocated instances of scale-down agents (Agent B) overwrite their original weights via `Get` API. The implementation of `Set/Get` API will be detailed in §8. Finally, the migrated instance detaches from the donor Agent B and registers with the target Agent A to accept new rollout requests.

6 Joint Orchestrator

Existing training frameworks for LLM-based MARL typically employ the colocated architecture, where the rollout and training phases share the identical resource pool. In fact, the rollout and training phases have different task characteristics. The rollout phase is memory-bound, while the performance of the training phase is limited by computational power. It is suboptimal to allocate the same resources to two distinct phases, leading to resource underutilization. Beyond resource conflicts, the colocated architecture also degrades MARL's training efficiency because the rollout and training phases alternate for time-division multiplexing of resources. The shared devices cannot transition to the training state due to critical path synchronization barriers [11, 38, 46, 64]. This problem is exacerbated in MARL compared to single-agent RL, as multi-agent interactions amplify long-tail latency. In light of the above discussion, the Joint Orchestrator advocates the disaggregated architecture for LLM-based MARL, which allocates dedicated, physically separate resource pools for the rollout and training phases. This decoupling design avoids resource conflicts and allows independent scaling according to phase characteristics.

To fully unlock the potential of disaggregation, the orchestrator needs to determine the collaborative paradigm to overlap the rollout and training phases. Although asynchronous pipelines have been widely adopted in single-agent RL [10, 15, 61], they operate on a single policy model and flat trajectories. Applying them directly to MARL introduces two limitations. Firstly, these schemes require the entire

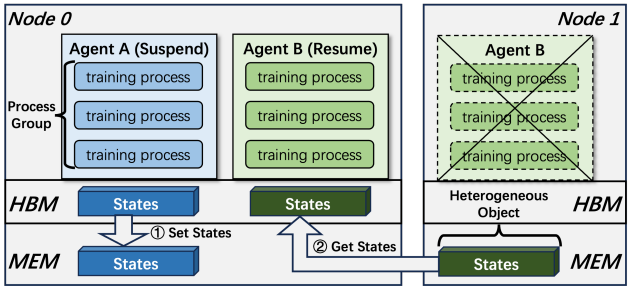


Figure 5. Efficient swap-in/out of training states for agent-centric resource allocation.

multi-agent interactions to complete before any experience can be used for training. Secondly, single-agent asynchronous pipelines only maintain consistency for a single policy, while MARL training involves multiple policy models. In fact, MARL is more sensitive to policy staleness compared with single-agent RL because the bias propagates along the dependency paths rather than remaining local to one agent invocation [11]. If different vertices of the MEG observe incompatible policy versions, the collected experience no longer corresponds to consistent multi-agent interaction, ultimately causing significant accuracy degradation.

To this end, the Joint Orchestrator adopts an agent-level asynchronous pipeline, which decomposes the global batch at the granularity of agents. Concretely, completed vertices become visible to the training view of MEG as soon as part of the experience is available. Thanks to the multi-table organization, once the agent-level accumulated experience in the MEG Store exceeds a predefined micro batch size, the Joint Orchestrator dispatches them to the corresponding training process for independent gradient computation on heterogeneous policy models. In this way, policy training is triggered incrementally in a fine-grained manner without waiting for the entire batch [6]. This allows training to start much earlier and execute in parallel with the remaining rollout requests, thereby alleviating pipeline bubbles [9] and synchronization barriers caused by the long-tail critical path.

More importantly, we carefully manage policy versions by decoupling *when gradients are computed* from *when policy updates are committed*. Each agent computes its gradients in advance, but will not immediately update its policy models, which creates an implicit frozen window for parallelizing rollout and training without introducing multi-agent policy version inconsistency. Policy model parameters remain frozen during all micro batch gradient computation, and there is no additional policy version. Therefore, different vertices leverage consistent and fresh policy models for the ongoing rollouts, avoiding the cross-agent staleness issue. After processing a full batch, all agents aggregate the cached gradients to perform unified policy updating, and policy versions are also updated in the MEG store. The updated parameters are synchronized to all inference instances via high-bandwidth D2D interconnects, ensuring that the next rollout also uses the up-to-date policy model. We also provide a theoretical consistency analysis of our agent-level asynchronous pipeline in Appendix A.

7 Training Engine

7.1 Agent-Centric Resource Allocation

In the training view of MEG, completed vertices are grouped by agent and policy version, and only agents with sufficient experience in the MEG Store need to be activated. As a result, the set of agents undergoing policy optimization and their entry is non-deterministic. Static allocation inevitably leads to resource waste, as inactive agents monopolize memory and computing resources throughout training. To accommodate the sparse subgraph activation pattern, our Training Engine integrates agent-centric allocation to bind training resources only where and when needed. In particular, we implement time-division multiplexing for training resources via process group abstraction, which encapsulates all training processes associated with an individual agent. The Training Engine initializes a process group for each agent and obtains handles for all required processes. The training processes are activated, suspended, and resumed using a gang-scheduling strategy [7], enabling collective lifecycle management.

Once the MEG Store accumulates sufficient training samples for a specific agent, the corresponding process group activates the training processes and dynamically schedules them to available resources within the cluster. After an agent completes training for the current micro batch or when no new experiences are available, the process group is terminated. Unlike naive suspension, which retains process contexts in memory, our Training Engine terminates processes and immediately releases all associated hardware resources (including computing cores and HBM) back to the cluster pool. When the Joint Orchestrator detects new experiences for the agent, the process group is re-created on available resources, and policy training resumes from the last checkpoint. This "suspend-to-destroy" design ensures that storage and computation resources are never locked by inactive agents. In this way, the training resource allocation in FlexMARL is aligned with agent-specific workloads rather than being preassigned to fixed hardware, maintaining high training throughput even with a massive number of potential agents.

7.2 Training State Swap

With dynamic reallocation of hardware resources, training states (including weights and optimizer states) cannot remain resident in HBM after process group suspension. Retaining the states for inactive agents would quickly exhaust device memory capacity and cause out-of-memory (OOM) failures, especially when the number of potential agents far exceeds available hardware resources. Consequently, on-demand resource binding necessitates efficient state swap between device and host memory. Herein, we again leverage `Set/Get`

API (§8) to decouple logical state references from physical storage, enabling the Training Engine to track and access agent states consistently across distributed nodes.

As presented in Figure 5, the process group for the suspended agent (Agent A) triggers a checkpoint. Upon a `Set` operation, the current training states are offloaded from device memory to host memory as heterogeneous objects. For the resumed agent (Agent B), the Training Engine exploits locality-aware deployment to minimize state migration latency. The process group is prioritized for scheduling to its previously deployed node. Finally, the Training Engine restores the states into the destination device memory through the `Get` API and creates the corresponding process group.

8 Implementation and Deployment

We implement FlexMARL in 8k lines of C/C++ code and 12k lines of Python code. vLLM v0.9.1 is integrated as the rollout backend to ensure low-latency inference. Policy training is built on DeepSpeed v0.16.9 with ZeRO-3 enabled [33]. We deploy FlexMARL on a production 48-node cluster. Each node is equipped with 16 Ascend NPUs (64GB memory) and interconnected via HCCS [62]. At the foundational layer, the vendor-specific SDK and hardware abstraction library provide the necessary compute kernels, interfaced through a PyTorch adapter.

8.1 Location-Agnostic `Set`/`Get` API

To relieve developers from the complexities of manual memory management and transmission, FlexMARL introduces heterogeneous objects for data in device and host memory. Each object is accessed through key-value (KV) semantics, where the key provides a stable logical reference and the runtime resolves the corresponding physical location. The host and device memory are *logically* unified to facilitate seamless data transfer across storage hierarchies. Each node in the cluster maintains a dedicated resident daemon, which is responsible for managing the distributed metadata of heterogeneous objects. Based on this metadata service, hierarchical load balancing and training state swap leverage a `Set`/`Get` API for intra-domain communication (D2D) and cross-tier communication (H2D/D2H).

D2D transfer is used when rollout capacity is migrated across different agents, where newly allocated inference instances must rapidly load the scale-up agent's model weights. The resident daemon utilizes a publish-subscribe coordination pattern where the invocation of the `Set` API triggers the registration of the device memory segment. This process records critical location metadata, including physical device address, memory offset, and node-level identifiers. Data retrieval via the `Get` API initiates a multi-stage resolution process. The receiver queries the resident daemon to obtain the physical location and device ID of heterogeneous objects. Once the peer identity is established, the sender/receiver instantiates or reuses a specialized communication domain to execute point-to-point primitives , thereby completing the D2D data transfer.

H2D/D2H transfer is primarily used by the Training Engine to swap weights and optimizer states for suspended and resumed agents. The `Set` API instantiates a buffer in the host memory. The resident daemon then registers metadata that links the device memory extents to the corresponding host-side buffer. Following metadata registration, heterogeneous objects are asynchronously migrated from the device to the host memory via D2H transfer. When a receiver initiates a `Get` request, the location metadata can be queried from the resident daemon using the user-defined key. The heterogeneous objects are directly copied from the local host memory to the target device. In the case of cross-node retrieval, the resident daemon synchronizes the remote sender and the local receiver to facilitate a high-throughput, zero-copy migration via an RDMA-enabled network [27]. Once the data is staged within the local host domain, it is finalized through a remote Host-to-Device (RH2D) operation.

8.2 Practical Deployment Lessons

Building FlexMARL and deploying it on a production cluster provided unique insights into the LLM-based MARL training. We summarize our experiences into the following key lessons during MARL system development.

Hardware-Aware Abstraction. Modern distributed frameworks usually abstract away physical device details to simplify development. However, the absence of stable UUIDs in the NPU driver stack led to failures in high-level zero-copy mechanisms. FlexMARL addresses this by enforcing process-level determinism and mapping internal PIDs to logical ranks. This design provides an explicit addressing scheme for heterogeneous objects without depending on unstable driver-level identifiers.

Rollout-Training Parameter Synchronization. We find that existing fine-grained, parameter-by-parameter synchronization schemes are costly. The task scheduling and kernel launching overhead accounts for over 99% of synchronization latency when iterating over billions of parameters. This suggests that the cost of invoking communication primitives could dwarf the actual time spent on transferring data. FlexMARL aggregates weights into a single contiguous memory buffer, reducing synchronization complexity from $O(Num\_of\_Parameters)$ to $O(1)$ and achieving a 200× speedup in practical deployment. These experiences reveal that system software for accelerators must aggressively batch control-plane operations to amortize scheduling costs.

Cross-Node Agent Deployment. Agent deployment in existing frameworks (e.g., MARTI [53]) fails to scale effectively across multiple nodes, revealing critical performance bottlenecks. The reason is that all resources across nodes are managed via a single placement group (PG) using a "PACK"

Table 2. Overall training performance of different MARL frameworks on the MA and CA datasets.

Dataset	Framework	E2E Time	Speedup	Throughput
MA	MAS-RL	914.4s	1.0×	119.0tps
	DistRL	293.8s	3.1×	401.0tps
	MARTI	174.1s	5.3×	642.8tps
	FlexMARL	126.1s	7.3×	910.2tps
CA	MAS-RL	438.6s	1.0×	265.5tps
	DistRL	130.0s	3.4×	571.6tps
	MARTI	112.8s	3.9×	655.9tps
	FlexMARL	78.8s	5.6×	821.4tps

scheduling strategy[1]. Both training and inference processes are deployed in the same PG. By generating a bundle list derived from a predefined device set, each process is bound to its corresponding device bundle. However, since the logical bundle ordering is not equivalent to physical device IDs, the "PACK" strategy fails in multi-agent settings. Training/inference processes of the same agent may be scheduled to different nodes, leading to additional communication overhead and potential network failure risk [30]. By shifting scheduling granularity from a cluster level to individual nodes, our framework instantiates an independent PG for each node using a "STRICT_PACK" strategy[1]. By establishing a deterministic one-to-one mapping between logical bundles and physical device IDs, FlexMARL effectively eliminates unintended cross-node communication and enhances system stability during large-scale MARL deployment.

9 Performance Evaluation

Our evaluation aims to answer the following six research questions (RQ):

- **RQ1**: How does FlexMARL compare with existing MARL frameworks in end-to-end training performance?
- **RQ2**: Can FlexMARL mitigate topology-induced load imbalance in multi-agent interactions?
- **RQ3**: Can FlexMARL improve hardware utilization for dynamic resource demands?
- **RQ4**: Are the key components of FlexMARL necessary for improving MARL training efficiency?
- **RQ5**: What is the control plane overhead of FlexMARL?
- **RQ6**: How does FlexMARL perform with increasing agent counts and model sizes?

9.1 Experimental Setup

Models and Datasets. We validate the effectiveness and scalability of FlexMARL on two real-world industrial datasets. For the Merchant Assistant (MA) dataset, multiple agents collaborate to help merchants perform complex store management tasks, including sales performance analysis, marketing strategy optimization, and after-sales service. We employ the Qwen2.5-14B model as the backbone, in which each agent does not share parameters and independently optimizes its policy. For the Category Assistant (CA) dataset, MAS is responsible for order querying, pricing strategies, and inventory management suggestions. The policy models involve two different sizes, i.e., Qwen2.5-14B and Qwen2.5-32B. These datasets are constructed for specific e-commerce tasks to simulate multi-agent collaboration requirements. Each sample comprises structured user queries, historical interaction context, and ground-truth agent responses. The detailed information about multi-agent assistant datasets is hidden due to business and confidentiality concerns.

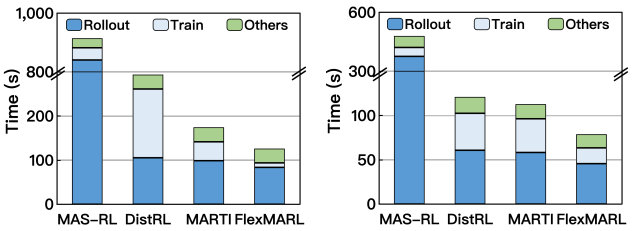
Training Configurations. For dependency parallel sampling, inter-query and intra-query parallelism are set to 4 and 16, respectively, to accelerate the processing of user queries. The maximum length of response tokens is 8192. The load-balancing disparity threshold Δ is set to 5. The batch size and micro batch size in our agent-level asynchronous pipeline are set to 64 and 16, respectively. We adopt the GRPO algorithm [34] for policy training and use a learning rate of 1e-6 with the Adam optimizer. All experiments are run with a fixed random seed of 2048 for reproducibility. Unless otherwise specified, all frameworks share the same hyperparameters to ensure fair evaluation.

Baselines. Most existing RL frameworks [10, 18, 35] are designed for a single model and cannot support MARL. Under such circumstances, we compare our proposed framework against the following MARL baselines:

- **MAS-RL** extends the single-agent RL training framework to multi-agent settings. The naive MARL implementation adopts the *colocated* architecture and the same optimization pipeline as agentic RL to perform uniform scheduling and static resource allocation during both rollout and training phases.
- **DistRL** enables flexible resource allocation through the *disaggregated* architecture, avoiding frequent onload/offload operations. The rollout and training phases of MARL are deployed on dedicated resource pools and coordinated under a synchronous paradigm.
- **MARTI** (ICLR 2026) [53] is the state-of-the-art MARL framework that supports centralized multi-agent interactions and distributed policy training. Asynchronous rollouts are designed to further improve the training efficiency of MARL.

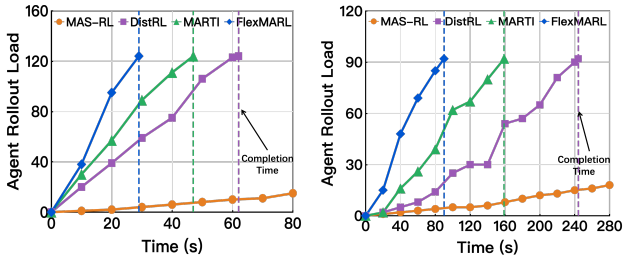
Evaluation Metrics. The design goal of FlexMARL is to facilitate efficient training of MARL algorithms on large-scale infrastructure. Our experiments focus on training efficiency metrics to comprehensively evaluate performance. (1) E2E time records the average latency of all phases (rollout, training, and others) for a single sample. (2) Speedup is calculated as the ratio of the E2E time between frameworks and MAS-RL, quantifying the relative performance improvement. (3)

[1] https://docs.ray.io/en/latest/ray-core/scheduling/placement-group.html



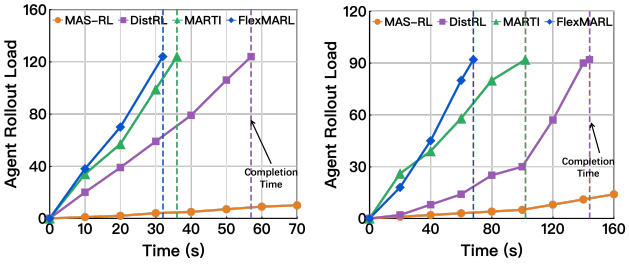
(a) MA Dataset (b) CA Dataset

Figure 6. E2E time breakdown of different MARL frameworks on the MA and CA datasets.



(a) Agent A (b) Agent B

Figure 7. Processed rollout load of representative agents on the MA dataset.



(a) Agent A (b) Agent B

Figure 8. Processed rollout load of representative agents on the CA dataset.

Throughput is denoted by the number of generated tokens per second. (4) Task accuracy is used to verify whether our solution preserves the benefit of MARL policy optimization. (5) Agent load is reflected by the total number of rollout requests that each agent has processed. (6) Hardware utilization rate represents the percentage of time that AI cores remain active within a given training period.

9.2 RQ1: Overall Performance

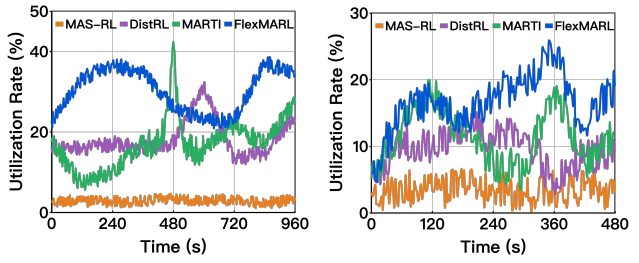
We first evaluate the overall performance of FlexMARL and three representative baselines. Table 2 summarizes the overall training efficiency, and Figure 6 shows the performance breakdown during rollout and training phases on the MA and CA datasets. FlexMARL consistently delivers the highest training speed and throughput. We highlight three observations from the experimental results. Firstly, our proposed framework outperforms MAS-RL by a large margin. For example, FlexMARL simultaneously reduces rollout and training time, ultimately achieving end-to-end speedups of 7.3× and 5.6× on the MA and CA datasets. The corresponding throughput improvements are 7.6× and 3.1×, respectively. The naive adaptation in MAS-RL cannot address MARL-specific challenges and suffers from severe queuing delays. FlexMARL leverages parallel sampling and elastic scaling to eliminate such bottlenecks, reducing rollout latency by 86% on average. Secondly, even with the disaggregated architecture, the E2E speed of DistRL is only 42%–61% of ours. Compared to DistRL, FlexMARL reduces policy optimization time from 155.9s to 10.2s on the MA dataset. This is because DistRL relieves resource contention but introduces significant pipeline bubbles, resulting in resource waste. Our agent-level asynchronous pipeline overlaps rollout and training phases, hiding tail latency. Thirdly, FlexMARL improves training speed and throughput by 1.4× and 1.3× over the state-of-the-art MARTI framework on average. Although asynchronous rollouts can speed up experience collection, their fixed resource allocation strategy fails to accommodate skewed and dynamic workloads. The results above demonstrate that our co-design of the rollout, training, and orchestration phases provides substantial performance gains.

9.3 RQ2: Agent Rollout Load

To validate the rollout efficiency, we select two representative agents from two datasets and monitor the number of processed requests within a single step. Figures 7 and 8 plot the load patterns of different frameworks. We find that FlexMARL processes rollout requests much faster than the baselines for all datasets. For instance, FlexMARL completes all pending requests within 90s for Agent B on the MA dataset, while DistRL and MARTI require about 244s and 159s. The corresponding speedups are 2.7× and 1.8×, respectively. In particular, MAS-RL maintains a constant workload throughout training, as it processes all queries serially without parallelization, resulting in the longest completion time. These baselines cannot mitigate topology-induced rollout skew. The core agents experience persistent queueing of rollout requests, which blocks the MARL training process. Compared to other methods, the queued requests of FlexMARL exhibit a faster increase followed by quicker decrease. The explanation for this phenomenon is that our Rollout Engine combines dependency parallel sampling with elastic scaling, achieving balanced workloads across multiple agents. Upstream agents also adjust the number of inference instances to efficiently process their queued requests, thereby accelerating rollout along the MEG.

9.4 RQ3: Resource Utilization

We measure the overall hardware utilization of FlexMARL and three baselines throughout the MARL training process.



(a) MA Dataset (b) CA Dataset

Figure 9. Resource utilization rates of different MARL frameworks on the MA and CA datasets.

The results on the MA and CA datasets are presented in Figure 9. We observe that FlexMARL maintains higher resource utilization rates compared to baselines, achieving an average of 32.4% on MA and 19.8% on CA. Other frameworks struggle with inefficient resource usage. For example, MAS-RL and MARTI exhibit severe utilization gaps with an average of only 3.6% and 12.3% on the CA dataset, respectively. This inefficiency stems from the colocated architecture that forces training and rollout to use the same number of resources. Heterogeneous requirements of different phases result in inherent resource conflict. Moreover, DistRL shows noticeable utilization drops, i.e., 16.9% on MA and 10.2% on CA. The rollout and training phases are executed alternately, leaving training resources idle. In contrast, our Training Engine dynamically binds resources to active agents through on-demand resource allocation. The above results imply that our proposed framework can accommodate dynamic training demand, effectively improving resource utilization.

9.5 RQ4: Ablation Study

We conduct ablation studies to quantify the individual contributions of rollout load balancing, agent-level asynchronous pipeline, and on-demand resource allocation. As summarized in Table 3, removing load balancing reduces the training throughput by 19.8%-25.5%, and the speedup drops to 6.0× and 4.6×. This indicates that our Rollout Engine elastically reallocates inference capacity and prevents overloaded agents from becoming persistent bottlenecks. For the variant without asynchronous pipeline, the performance degradation is even more pronounced due to severe pipeline bubbles. The E2E time increases by 103.2% for MA and 57.5% for CA. The Joint Orchestrator allows training to begin as soon as part of experiences is available, rather than waiting for the entire batch. It effectively hides the long-tail latency of critical paths. Agent-centric resource allocation in Training Engine is also important for adapting to dynamic and heterogeneous agent workloads. Without allocation component, idle resources cause a decrease in training speed. The above results highlight that these core components are necessary for achieving the full performance of FlexMARL.

Table 3. Ablation study of core components in FlexMARL.

Dataset	Variant	E2E Time	Speedup	Throughput
MA	w/o balancing	152.2s	6.0×	729.9tps
	w/o async	256.2s	3.6×	444.0tps
	w/o allocation	132.2s	6.9×	784.7tps
	FlexMARL	126.1s	7.3×	910.2tps
CA	w/o balancing	95.9s	4.6×	611.8tps
	w/o async	124.1s	3.5×	608.4tps
	w/o allocation	83.6s	5.2×	759.1tps
	FlexMARL	78.8s	5.6×	821.4tps

Table 4. Task accuracy comparison under synchronous and asynchronous pipelines.

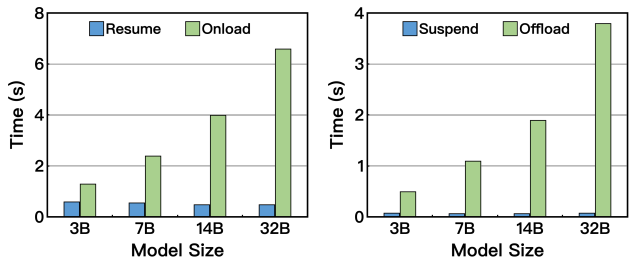
Dataset	SFT-only	Sync MARL	FlexMARL
MA	86.3%	90.3%	90.1%
CA	86.4%	90.4%	90.6%

We further verify that our training efficiency gains do not come at the cost of consistency. Table 4 reports the final task accuracy of FlexMARL, synchronous MARL baseline, and supervised fine-tuning (SFT) baseline. We observe that FlexMARL maintains task accuracy nearly identical to the synchronous baseline. Meanwhile, both synchronous MARL and FlexMARL achieve approximately 4% accuracy improvement compared with the SFT-only baseline, demonstrating the benefit of MARL policy optimization.

9.6 RQ5: Control Plane Overhead

As a distributed system, FlexMARL incurs overhead from state communication and control plane operations. We conduct high-load tests to verify that they do not become system bottlenecks. We first quantify the state swap overhead across four representative model sizes. The swap-in overhead includes resuming the process group and loading states back into device memory via `Get` API. The swap-out overhead includes suspending the process group and offloading training states to host memory via `Set` API. As shown in Figure 10, both suspend and resume latencies are minimal and stay nearly constant regardless of model scale. The onload and offload overhead grow gradually with model size. Nevertheless, after integrating the location-agnostic `Set`/`Get` API, total swap overhead is only 11s for the largest model, which can be effectively hidden behind rollout latency.

We also measure the latency of other core operations under production workload conditions in Table 5. Load monitoring in the Rollout Engine involves periodic polling of request queue lengths across all agents, incurring a latency of 380~610μs. For balancing decision-making, which performs load disparity calculation and elastic scaling decisions, the latency is only 50~100μs even for 1,000 agents. For the



(a) Swap-in Overhead (b) Swap-out Overhead

Figure 10. Swap-in/out overhead of training states across different model sizes.

Table 5. Control plane overhead breakdown of FlexMARL.

Operation	Control Plane Overhead
Load Monitoring	380~610μs
Scaling Decision-making	50~100μs
Experience Retrieval of MEG Store	30.324ms

overhead of experience retrieval in the MEG store, we observe an average latency of 30.324ms for 32 concurrent requests each querying 1,024 keys. These results demonstrate that additional overheads introduced by FlexMARL are acceptable compared to the E2E training time, enabling robust large-scale deployments of LLM-based MARL.

9.7 RQ6: Scalability

To evaluate the scalability of FlexMARL, we investigate system performance under large-scale heterogeneous scenarios. Table 6 reports the E2E time and training throughput on the MA dataset across varying agent counts and model sizes. FlexMARL demonstrates robust scalability, achieving a peak throughput of 754.2tps with 15 agents. Critically, FlexMARL successfully supports complex heterogeneous configurations, such as a mixed multi-agent setting of 3×32B and 7×14B. In contrast, existing MARL frameworks lack the cross-node agent placement capabilities required for such heavy, uneven workloads, which typically results in OOM errors within one node. Our end-to-end optimization enables FlexMARL to maintain high throughput even when involving multiple 32B agents, demonstrating its unique capability to support industrial-scale MARL workloads.

10 Related Work

Frameworks for LLM Post-Training. RL has emerged as a critical paradigm for enhancing LLM capabilities [13, 51], and it relies heavily on scalable training infrastructure [12, 29, 58]. Early frameworks like ColossalChat [20] and DeepSpeed-Chat [49] adopt the colocated architecture where the rollout and training phases share the same resource pool, leading to resource contention. To improve throughput, recent RL frameworks such as OpenRLHF [18], veRL [35], and AReaL [10] enable asynchronous pipeline and leverage distributed runtimes [5, 30] to orchestrate the different phases. AgentRL [52] and SkyRL-Agent [2] further extend these pipelines to support multi-turn and long-horizon agentic RL training. Other solutions aim to improve inference efficiency by using quantized actors [22] or adaptive rollout allocation [57]. However, these general-purpose RL frameworks are primarily designed for a single policy model and cannot capture cross-agent dependencies, agent load skew, or dynamically activated training subsets inherent in LLM-based MARL.

Table 6. Performance evaluation of FlexMARL in large-scale heterogeneous MARL scenarios.

Configuration	Rollout	Training	E2E Time	Throughput
5×32B	96.2s	54.5s	160.3s	265.9tps
3×32B + 7×14B	96.7s	15.6s	132.5s	334.8tps
15×14B	35.4s	5.3s	41.9s	754.2tps

Frameworks for Multi-Agent Systems. Although prior works achieve algorithm-level optimization on communication protocols [56], coordination strategies [14, 25, 32, 39], and reward functions [3, 36, 42], the infrastructure bottlenecks that support these MARL algorithms are overlooked. Regarding system-level works, some libraries (such as AutoGen [45], MetaGPT [16], and MARFT [24]) simplify deployment through high-level abstractions without collaborative multi-agent training support. Dr. MAS [8] identifies gradient-norm instability in GRPO-style multi-agent training and stabilizes optimization with agent-wise normalization. Other MARL frameworks [41, 59] only provide basic training capabilities for MARL and treat the training infrastructure as a black box. MARTI [53] integrates asynchronous rollouts to enhance training efficiency, but still employs static resource allocation for different agents, leading to synchronization barriers and resource underutilization, especially when multi-agent workloads are imbalanced [4]. In summary, existing frameworks are incremental extensions of single-agent RL. They naively optimize the rollout or training phase and do not jointly optimize all MARL-specific challenges based on the dynamic and collaborative characteristics of MARL, thus hindering large-scale training efficiency.

11 Conclusion

In this paper, we present FlexMARL, a rollout-training co-designed framework that bridges the gap between LLM-based MARL algorithms and infrastructure. We identify three MARL-specific challenges that hinder large-scale training efficiency. Building upon the MEG Store, FlexMARL designs dependency parallel sampling with load balancing for rollout, agent-level asynchronous pipeline for orchestration, and on-demand resource allocation for training. The evaluation results on a production cluster confirm the superior scalability and efficiency of our proposed framework.

References

[1] Lorenzo Canese, Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Daniele Giardino, Marco Re, and Sergio Spanò. 2021. Multi-agent reinforcement learning: A review of challenges and applications. *Applied Sciences* 11, 11 (2021), 4948.

[2] Shiyi Cao, Dacheng Li, Fangzhou Zhao, Shuo Yuan, Sumanth R Hegde, Connor Chen, Charlie Ruan, Tyler Griggs, Shu Liu, Eric Tang, et al. 2025. Skyrl-agent: Efficient rl training for multi-turn llm agent. *arXiv preprint arXiv:2511.16108* (2025).

[3] James L Carter, Yuxuan Liu, and Thomas K Lee. 2026. Multi-Agent Post-Co-Training of Large Language Models via Reinforcement Learning. *Frontiers in Applied Physics and Mathematics* 3, 2 (2026), 10–16.

[4] Guanzhong Chen, Shaoxiong Yang, Chao Li, Wei Liu, Jian Luan, and Zenglin Xu. 2025. Heterogeneous Group-Based Reinforcement Learning for LLM-based Multi-Agent Systems. *arXiv preprint arXiv:2506.02718* (2025).

[5] Qiong Chen, Jianmin Qian, Yulin Che, Ziqi Lin, Jianfeng Wang, Jie Zhou, Licheng Song, Yi Liang, Jie Wu, Wei Zheng, et al. 2024. Yuanrong: A production general-purpose serverless system for distributed applications in the cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 843–859.

[6] Tiancheng Chen, Ales Kubicek, Langwen Huang, and Torsten Hoefler. 2025. {CrossPipe}: Towards Optimal Pipeline Schedules for {Cross-Datacenter} Training. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 1089–1108.

[7] Dror G Feitelson and Morris A Jettee. 1997. Improved utilization and responsiveness with gang scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 238–261.

[8] Lang Feng, Longtao Zheng, Shuo He, Fuxiang Zhang, and Bo An. 2026. Dr. mas: Stable reinforcement learning for multi-agent llm systems. *arXiv preprint arXiv:2602.08847* (2026).

[9] Weiqi Feng, Yangrui Chen, Shaoyu Wang, Yanghua Peng, Haibin Lin, and Minlan Yu. 2025. Optimus: Accelerating {Large-Scale}{Multi-Modal}{LLM} Training by Bubble Exploitation. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 161–177.

[10] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, et al. 2025. AReaL: A Large-Scale Asynchronous Reinforcement Learning System for Language Reasoning. *arXiv preprint arXiv:2505.24298* (2025).

[11] Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, et al. 2025. Rollpacker: Mitigating long-tail rollouts for fast, synchronous rl post-training. *arXiv preprint arXiv:2509.21009* (2025).

[12] Hao Ge, Junda Feng, Qi Huang, Fangcheng Fu, Xiaonan Nie, Lei Zuo, Haibin Lin, Bin Cui, and Xin Liu. 2025. ByteScale: Communication-Efficient Scaling of LLM Training with a 2048K Context Length on 16384 GPUs. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 963–978.

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

[14] Ai Han, Junxing Hu, Pu Wei, Zhiqian Zhang, Yuhang Guo, Jiawei Lu, and Zicheng Zhang. 2025. JoyAgents-R1: Joint Evolution Dynamics for Versatile Multi-LLM Agents with Reinforcement Learning. *arXiv preprint arXiv:2506.19846* (2025).

[15] Zhenyu Han, Ansheng You, Haibo Wang, Kui Luo, Guang Yang, Wenqi Shi, Menglong Chen, Sicheng Zhang, Zeshun Lan, Chunshi Deng, et al. 2025. AsyncFlow: An Asynchronous Streaming RL Framework for Efficient LLM Post-Training. *arXiv preprint arXiv:2507.01663* (2025).

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

[17] Jinbin Hu, Wenxue Li, Xiangzhou Liu, Junfeng Wang, Bowen Liu, Ping Yin, Jianxin Wang, Jiawei Huang, and Kai Chen. 2025. {FLB}: Fine-grained Load Balancing for Lossless Datacenter Networks. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 365–380.

[18] Jian Hu, Xibin Wu, Zilin Zhu, Weixun Wang, Dehao Zhang, Yu Cao, et al. 2024. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. *arXiv preprint arXiv:2405.11143* (2024).

[19] Tianyi Hu, Zhiqiang Pu, Xiaolin Ai, Tenghai Qiu, and Jianqiang Yi. 2024. Measuring Policy Distance for Multi-Agent Reinforcement Learning. *arXiv preprint arXiv:2401.11257* (2024).

[20] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. 2023. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*. 766–775.

[21] Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. 2024. A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth* 1, 1 (2024), 9.

[22] Yuhang Li, Reena Elangovan, Xin Dong, Priyadarshini Panda, and Brucek Khailany. 2026. Qurl: Efficient reinforcement learning with quantized rollout. *arXiv preprint arXiv:2602.13953* (2026).

[23] Zhenghao Li, Zhi Zheng, Wei Chen, Jielun Zhao, Yong Chen, Tong Xu, and Enhong Chen. 2026. DynaDebate: Breaking Homogeneity in Multi-Agent Debate with Dynamic Path Generation. *arXiv preprint arXiv:2601.05746* (2026).

[24] Junwei Liao, Muning Wen, Jun Wang, and Weinan Zhang. 2025. Marft: Multi-agent reinforcement fine-tuning. *arXiv preprint arXiv:2504.16129* (2025).

[25] Shuo Liu, Tianle Chen, Ryan Amiri, and Christopher Amato. 2026. Learning Decentralized LLM Collaboration with Multi-Agent Actor Critic. *arXiv preprint arXiv:2601.21972* (2026).

[26] Shuo Liu, Tianle Chen, Zeyu Liang, Xueguang Lyu, and Christopher Amato. 2025. Llm collaboration with multi-agent reinforcement learning. *arXiv preprint arXiv:2508.04652* (2025).

[27] Haodi Lu, Haikun Liu, Yujian Zhang, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. 2025. Fast Distributed Transactions for {RDMA-based} Disaggregated Memory. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 943–958.

[28] Francisco Martinez-Gil, Miguel Lozano, and Fernando Fernández. 2014. MARL-Ped: A multi-agent reinforcement learning based framework to simulate pedestrian groups. *Simulation Modelling Practice and Theory* 47 (2014), 259–275.

[29] Qingkai Meng, Hao Zheng, Zhenhui Zhang, ChonLam Lao, Chengyuan Huang, Baojia Li, Ziyuan Zhu, Hao Lu, Weizhen Dang, Zitong Lin, et al. 2025. Astral: A datacenter infrastructure for large language model training at scale. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 609–625.

[30] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 561–577.

[31] Ruwei Pan, Hongyu Zhang, and Chao Liu. 2025. CodeCoR: An LLM-based self-reflective multi-agent framework for code generation. *arXiv preprint arXiv:2501.07811* (2025).

[32] Chuhao Qin and Evangelos Pournaras. 2025. Strategic Coordination for Evolving Multi-agent Systems: A Hierarchical Reinforcement and Collective Learning Approach. *arXiv preprint arXiv:2509.18088* (2025).

[33] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[34] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. 2024.

Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300* (2024).

[35] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*. 1279–1297.

[36] Haoran Su, Yandong Sun, and Congjia Yu. 2026. The End of Reward Engineering: How LLMs Are Redefining Multi-Agent Coordination. *arXiv preprint arXiv:2601.08237* (2026).

[37] Chuanneng Sun, Songjun Huang, and Dario Pompili. 2024. Llm-based multi-agent reinforcement learning: Current and future directions. *arXiv preprint arXiv:2405.11106* (2024).

[38] Weixun Wang, Shaopan Xiong, Gengru Chen, Wei Gao, Sheng Guo, Yancheng He, Ju Huang, Jiaheng Liu, Zhendong Li, Xiaoyang Li, et al. 2025. Reinforcement Learning Optimization for Large-Scale Learning: An Efficient and User-Friendly Scaling Library. *arXiv preprint arXiv:2506.06122* (2025).

[39] Xin Wang, Chen Zhao, Tingwen Huang, Prasun Chakrabarti, and Jürgen Kurths. 2023. Cooperative learning of multi-agent systems via reinforcement learning. *IEEE Transactions on Signal and Information Processing over Networks* 9 (2023), 13–23.

[40] Zhaodong Wang, Samuel Lin, Guanqing Yan, Soudeh Ghorbani, Minlan Yu, Jiawei Zhou, Nathan Hu, Lopa Baruah, Sam Peters, Srikanth Kamath, et al. 2025. Intent-driven network management with multi-agent LLMs: The confucius framework. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 347–362.

[41] Rui Wei, Hanfei Yu, Xikang Song, Jian Li, Devesh Tiwari, Ying Mao, and Hao Wang. 2025. Multi-Agent Reinforcement Learning with Serverless Computing. In *Proceedings of the 2025 ACM Symposium on Cloud Computing*. 225–239.

[42] Yuan Wei, Xiaohan Shan, Ran Miao, and Jianmin Li. 2025. Lero: Llm-driven evolutionary framework with hybrid rewards and enhanced observation for multi-agent reinforcement learning. In *International Conference on Intelligent Computing*. Springer, 15–26.

[43] Christoph Wittner. 2026. Communication Methods in Multi-Agent Reinforcement Learning. *arXiv preprint arXiv:2601.12886* (2026).

[44] Duo Wu, Xianda Wang, Yaqi Qiao, Zhi Wang, Junchen Jiang, Shuguang Cui, and Fangxin Wang. 2024. Netllm: Adapting large language models for networking. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 661–678.

[45] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*.

[46] LLM Xiaomi, Bingquan Xia, Bowen Shen, Dawei Zhu, Di Zhang, Gang Wang, Hailin Zhang, Huaqiu Liu, Jiebao Xiao, Jinhao Dong, et al. 2025. MiMo: Unlocking the Reasoning Potential of Language Model–From Pretraining to Posttraining. *arXiv preprint arXiv:2505.07608* (2025).

[47] Zhiqiang Xie, Hao Kang, Ying Sheng, Tushar Krishna, Kayvon Fatahalian, and Christos Kozyrakis. 2025. Ai metropolis: Scaling large language model-based multi-agent simulation with out-of-order execution. *Proceedings of Machine Learning and Systems* 7 (2025).

[48] John Yan, Michael Yu, Yuqi Sun, Alexander Duffy, Tyler Marques, and Matthew Lyle Olson. 2026. Data-Centric Interpretability for LLM-based Multi-Agent Reinforcement Learning. *arXiv preprint arXiv:2602.05183* (2026).

[49] Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, et al. 2023. Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales. *arXiv preprint arXiv:2308.01320* (2023).

[50] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. 2022. The surprising effectiveness of ppo in cooperative multi-agent games. *Advances in neural information processing systems* 35 (2022), 24611–24624.

[51] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476* (2025).

[52] Hanchen Zhang, Xiao Liu, Bowen Lv, Xueqiao Sun, Bohao Jing, Iat Long Iong, Zhenyu Hou, Zehan Qi, Hanyu Lai, Yifan Xu, et al. 2025. Agentrl: Scaling agentic reinforcement learning with a multi-turn, multi-task framework. *arXiv preprint arXiv:2510.04206* (2025).

[53] 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. 2026. MARTI: A Framework for Multi-Agent LLM Systems Reinforced Training and Inference. In *The Fourteenth International Conference on Learning Representations*. https://openreview.net/forum?id=E7jZqo0A50

[54] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. 2021. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of reinforcement learning and control* (2021), 321–384.

[55] Shaowei Zhang and Deyi Xiong. 2025. Debate4MATH: Multi-agent debate for fine-grained reasoning in math. In *Findings of the Association for Computational Linguistics: ACL 2025*. 16810–16824.

[56] Xinren Zhang, Jiadong Yu, and Zixin Zhong. 2025. Learning Efficient Communication Protocols for Multi-Agent Reinforcement Learning. *arXiv preprint arXiv:2511.09171* (2025).

[57] Zhi Zhang, Zhen Han, Costas Mavromatis, Qi Zhu, Yunyi Zhang, Sheng Guan, Dingmin Wang, Xiong Zhou, Shuai Wang, Soji Adeshina, et al. 2026. Train less, learn more: Adaptive efficient rollout optimization for group-based reinforcement learning. *arXiv preprint arXiv:2602.14338* (2026).

[58] Zili Zhang, Yinmin Zhong, Yimin Jiang, Hanpeng Hu, Jianjian Sun, Zheng Ge, Yibo Zhu, Daxin Jiang, and Xin Jin. 2025. Disttrain: Addressing model and data heterogeneity with disaggregated training for multimodal large language models. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 24–38.

[59] Yujie Zhao, Lanxiang Hu, Yang Wang, Minmin Hou, Hao Zhang, Ke Ding, and Jishen Zhao. 2025. Stronger-MAS: Multi-Agent Reinforcement Learning for Collaborative LLMs. *arXiv preprint arXiv:2510.11062* (2025).

[60] Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. 2025. Mage: A multi-agent engine for automated rtl code generation. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–7.

[61] Yinmin Zhong, Zili Zhang, Xiaoniu Song, Hanpeng Hu, Chao Jin, Bingyang Wu, Nuo Chen, Yukun Chen, Yu Zhou, Changyi Wan, et al. 2025. StreamRL: Scalable, Heterogeneous, and Elastic RL for LLMs with Disaggregated Stream Generation. *arXiv preprint arXiv:2504.15930* (2025).

[62] Yuhang Zhou, Zibo Wang, Zhibin Wang, Ruyi Zhang, Chen Tian, Xiaoliang Wang, Wanchun Dou, Guihai Chen, Bingqiang Wang, Yonghong Tian, et al. 2025. Accelerating Model Training on Ascend Chips: An Industrial System for Profiling, Analysis and Optimization. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 1387–1408.

[63] Guobin Zhu, Rui Zhou, Wenkang Ji, and Shiyu Zhao. 2025. Lamarl: Llm-aided multi-agent reinforcement learning for cooperative policy generation. *IEEE Robotics and Automation Letters* (2025).

[64] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, et al. 2025. Megascale-infer: Efficient mixture-of-experts model serving with disaggregated expert parallelism. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 592–608.

A Theoretical Consistency Analysis of Agent-Level Asynchronous Pipeline

A.1 Sequence-Mean Optimization Objective

Let $\mathcal{N}$ be the set of trainable agents. The complete multi-agent policy published at global step k is:

$$\Theta^k = \left(\theta_n^k\right)_{n\in\mathcal{N}}, \tag{1}$$

where θ_n^k denotes the policy parameters of agent n, and V_n^k is the corresponding global version identifier. A version-consistent rollout generation uses Θ^k for all of its agent invocations. At global step k, the MEG Store associates the current rollout generation with global version V_n^k. Every vertex admitted to this generation records V_n^k, and the Rollout Engine serves it using the corresponding component of Θ^k. Parameters computed during training remain unpublished and cannot affect the current rollout generation.

Consider a trainable GRPO group q of agent n. Starting from the same invocation context, branch sampling produces G candidate sequences. Each candidate is continued until its terminal reward and credit assignment become available. Since agent n is fixed in this subsection, we omit the agent subscript. Let $R_{q,j}$ denote the reward of candidate sequence j in group q. The group-relative advantage is:

$$\begin{aligned}
\mu_q &= \frac{1}{G}\sum_{j=1}^{G} R_{q,j},\\
\sigma_q &= \sqrt{\frac{1}{G}\sum_{j=1}^{G}\left(R_{q,j}-\mu_q\right)^2},\\
\widehat{A}_{q,j} &= \frac{R_{q,j}-\mu_q}{\sigma_q+\delta}.
\end{aligned} \tag{2}$$

The complete group and its advantages are materialized in the MEG Store before dispatch. Downstream agents may affect its reward, but their tokens and parameters are excluded from the loss.

FlexMARL may perform multiple local optimizer steps before publishing the next global policy. For the following aggregation proof, we fix one agent, one global step, and one local optimizer step. Let θ_0 denote the unpublished policy parameters at the beginning of this local step, and let θ_{old} denote the behavior-policy parameters that generated the candidate sequences. The policy parameters remain at θ_0 throughout the frozen window. For token t of candidate sequence (q, j), the importance ratio is:

$$r_{q,j,t}(\theta) = \frac{\pi_\theta\left(o_{q,j,t} \mid h_{q,j,t}\right)}{\pi_{\theta_{\text{old}}}\left(o_{q,j,t} \mid h_{q,j,t}\right)}. \tag{3}$$

The clipped per-token policy loss is:

$$\begin{aligned}
\ell_{q,j,t}(\theta) = -\min\Big(& r_{q,j,t}(\theta)\widehat{A}_{q,j},\\
& \text{clip}\left(r_{q,j,t}(\theta), 1-\epsilon, 1+\epsilon\right)\widehat{A}_{q,j}\Big).
\end{aligned} \tag{4}$$

Sequence-mean reduction first averages valid token losses within each candidate sequence and then averages the resulting per-sequence losses. Let $m_{q,j,t} \in \{0,1\}$ indicate whether token t is a valid training token. The loss of candidate sequence (q, j) is:

$$L_{q,j}(\theta) = \frac{\sum_t m_{q,j,t}\ell_{q,j,t}(\theta)}{\sum_t m_{q,j,t}}. \tag{5}$$

Let $\mathcal{Q}$ denote the complete set of GRPO groups assigned to the current logical accumulation window. Each element $q \in \mathcal{Q}$ identifies one trainable invocation with a distinct invocation context. For every group q, branch sampling produces exactly G candidate sequences indexed by $j \in \{1,\dots,G\}$. We flatten these groups into the candidate-sequence index set:

$$C = \{(q,j) \mid q \in \mathcal{Q},\ j \in \{1,\dots,G\}\}. \tag{6}$$

Each pair $(q, j) \in C$ uniquely identifies candidate sequence j from GRPO group q. Because every group contains exactly G candidate sequences, the total number of candidate sequences in the accumulation window is:

$$W = |C| = \sum_{q\in\mathcal{Q}} G = G|\mathcal{Q}|. \tag{7}$$

The Joint Orchestrator fixes $\mathcal{Q}$, and therefore C and W, when it closes the accumulation window. The reference synchronous objective is the mean of all per-sequence losses in the window:

$$L_{\text{ref}}(\theta) = \frac{1}{W}\sum_{(q,j)\in C} L_{q,j}(\theta). \tag{8}$$

A.2 Incremental Gradient Caching

The Joint Orchestrator may dispatch a training-ready group before the complete accumulation window has closed. The candidate sequences are partitioned into micro batches satisfying:

$$C = \bigcup_{m=1}^{M}\mathcal{B}_m, \qquad \mathcal{B}_m \cap \mathcal{B}_{m'} = \varnothing \quad (m \neq m'). \tag{9}$$

Thus, every candidate sequence in C belongs to exactly one execution micro batch. For micro batch $\mathcal{B}_m$, the Training Engine computes the unnormalized sum of its per-sequence losses:

$$S_m(\theta) = \sum_{(q,j)\in\mathcal{B}_m} L_{q,j}(\theta). \tag{10}$$

It then performs the backward pass on S_m and caches the resulting gradient:

$$h_m = \nabla_\theta S_m(\theta)|_{\theta=\theta_0}. \tag{11}$$

The accumulation window may remain open while early micro batches are processed. During this window, the policy parameters remain at θ_0, while the optimizer state, behavior-policy parameters θ_{old}, materialized advantages, and stochastic state associated with each candidate remain unchanged.

FlexMARL performs no optimizer step, gradient clipping, or dynamic loss-scale update between micro batches.

After the accumulation window closes, the Training Engine normalizes the sum of the cached micro batch gradients:

$$g_{\text{async}} = \frac{1}{W} \sum_{m=1}^{M} h_m. \tag{12}$$

Normalization is performed once after all candidate sequences in the window are known. The same gradient-clipping rule as the synchronous backend is then applied, followed by one optimizer step.

Proposition A.1 (Preservation of sequence-mean aggregation)**.** *If the asynchronous and reference executions use the same candidate set, per-sequence losses, and accumulation boundary, then Equation* (12) *equals the gradient of Equation* (8) *at* θ_0 *in exact arithmetic.*

Proof. Using Equations (10) and (11), together with the non-overlapping partition in Equation (9), we obtain:

$$\begin{aligned} g_{\text{async}} &= \frac{1}{W} \sum_{m=1}^{M} \nabla_\theta \sum_{(q,j)\in\mathcal{B}_m} L_{q,j}(\theta) \Bigg|_{\theta=\theta_0} \\ &= \nabla_\theta \left[\frac{1}{W} \sum_{(q,j)\in C} L_{q,j}(\theta) \right] \Bigg|_{\theta=\theta_0} \\ &= \nabla_\theta L_{\text{ref}}(\theta)|_{\theta=\theta_0}. \end{aligned} \tag{13}$$

Thus, asynchronous arrival changes only the order and timing of micro batch computation, not the logical sequence-mean gradient. □

A.3 Unified Policy Updating

FlexMARL maintains disjoint policy parameters, optimizer states, gradient caches, and accumulation windows for different agents. Rewards may depend on outputs from multiple agents, but rewards and advantages are materialized constants during optimization. The loss of one agent therefore contains no trainable parameter of another agent. Agents may perform training at different times, but their updated policies are not synchronized or published independently. Let $\widetilde{\theta}_n^{k+1}$ denote the unpublished policy produced for agent n during global step k. For an agent not scheduled for an update, we have $\widetilde{\theta}_n^{k+1} = \theta_n^k$. Only after all scheduled agent updates have completed does the Joint Orchestrator enter the global publication barrier. The complete unpublished multi-agent policy is:

$$\widetilde{\boldsymbol{\Theta}}^{k+1} = \left(\widetilde{\theta}_n^{k+1} \right)_{n\in\mathcal{N}}. \tag{14}$$

The Joint Orchestrator then synchronizes every component of $\widetilde{\boldsymbol{\Theta}}^{k+1}$ to its designated inference instances. After all synchronization operations acknowledge completion, the Joint Orchestrator atomically updates the policy version in the MEG Store and admits the next rollout.

Proposition A.2 (Global policy consistency)**.** *MEG rollout observes one complete policy vector,* $\boldsymbol{\Theta}^k$ *or* $\boldsymbol{\Theta}^{k+1}$*, and never a mixture of old and new agent policies.*

Proof. Before publication completes, only $\boldsymbol{\Theta}^k$ is rollout-visible. Candidate parameters remain unpublished, and no vertices of the next generation are admitted during synchronization. After every inference instance receives its designated parameters, the version advances to V_n^{k+1} and the next generation begins. Therefore, partially updated policies are never rollout-visible. □