

# ROLLMUX: Phase-Level Multiplexing for Disaggregated RL Post-Training

Tianyuan Wu<sup>†</sup>, Lunxi Cao<sup>†</sup>, Yining Wei<sup>‡</sup>, Wei Gao<sup>†</sup>, Yuheng Zhao<sup>†</sup>, Dakai An<sup>†</sup>, Shaopan Xiong<sup>§</sup>,  
Zhiqiang Lv<sup>§</sup>, Ju Huang<sup>§</sup>, Siran Yang<sup>§</sup>, Yinghao Yu<sup>§</sup>, Jiamang Wang<sup>§</sup>, Lin Qu<sup>§</sup>, Wei Wang<sup>†</sup>  
<sup>†</sup>Hong Kong University of Science and Technology, <sup>‡</sup>UIUC, <sup>§</sup>Alibaba Group

## Abstract

Rollout-training disaggregation is emerging as the standard architecture for Reinforcement Learning (RL) post-training, where memory-bound rollout and compute-bound training are physically disaggregated onto purpose-built clusters to maximize hardware efficiency. However, the strict synchronization required by on-policy algorithms introduces severe *dependency bubbles*, forcing one cluster to idle while the dependent phase is running on the other. We present ROLLMUX, a cluster scheduling framework that reclaims these bubbles through cross-cluster orchestration. ROLLMUX is built on the insight that the structural idleness of one job can be effectively utilized by the active phase of another. To realize this, we introduce the *co-execution group* abstraction, which partitions the cluster into isolated locality domains. This abstraction enables a *two-tier scheduling architecture*: an *inter-group scheduler* that optimizes job placement using conservative stochastic planning, and an *intra-group scheduler* that orchestrates a provably optimal round-robin schedule. The group abstraction also imposes a *residency constraint*, ensuring that massive model states remain cached in host memory to enable “warm-start” context switching. We evaluate ROLLMUX on a production-scale testbed with 328 H20 and 328 H800 GPUs. ROLLMUX improves cost efficiency by  $1.84\times$  over standard disaggregation and  $1.38\times$  over state-of-the-art co-located baselines, all while achieving 100% SLO attainment.

## 1 Introduction

The focus of Large Language Model (LLM) development has shifted from pre-training to *Reinforcement Learning (RL) post-training* [1, 3, 6], a critical technique for unlocking reasoning capabilities in complex domains such as mathematics [2], coding [31], and tool use [8, 51]. To achieve optimal performance and model stability, production practices have converged on *synchronous, on-policy algorithms* [6, 35, 39]. This paradigm mandates a strict, iterative learning process comprising three phases with distinct resource bottlenecks: (1)

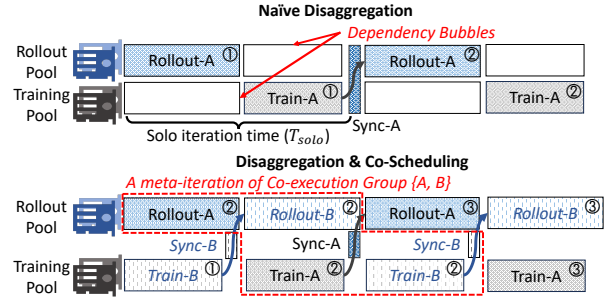


Figure 1: Comparison between existing disaggregated RL architecture and ROLLMUX’s co-scheduling paradigm.

**rollout**, a *memory-bandwidth-bound* inference stage where the model generates tokens as experience trajectories; (2) **training**, a *compute-intensive* stage where model parameters are optimized based on the rewards; and (3) **synchronization**, a *network-bound* stage where updated model parameters are propagated back to inference workers.

To accommodate the divergent resource demands of these phases, production deployment increasingly employs a *disaggregated architecture* [10, 44, 49, 58]. Unlike monolithic provisioning, this architecture separates the rollout and training phases onto purpose-built clusters (see Figure 1): a *rollout pool* consisting of cost-effective, inference-optimized GPUs (e.g., NVIDIA H20) and a *training pool* of high-performance, compute-optimized GPUs (e.g., NVIDIA H800). By aligning hardware capabilities with specific phase characteristics, disaggregation resolves the resource mismatches inherent in a monolithic setup. Consequently, it achieves superior cost efficiency and comparable throughput despite the introduction of cross-cluster synchronization overheads [10, 44, 49, 58].

However, disaggregation introduces a fundamental efficiency challenge caused by *dependency bubbles*. Due to the strict synchronization required by on-policy learning, the training cluster must remain idle during rollout, and vice versa (Figure 1-top), leading to severe cluster underutilization. While systems such as AReaL [10], StreamRL [58], and AsyncFlow [16] attempt to eliminate these bubbles by adopt-

ing *asynchronous, off-policy algorithms*, they achieve high utilization only by relaxing the synchronization requirements. This relaxation introduces *sample staleness* that often compromises model accuracy and convergence stability, rendering it unsuitable for tasks that demand strict on-policy performance.

We present ROLLMUX, a cross-cluster scheduling framework for disaggregated RL post-training that mitigates dependency bubbles by elevating the optimization scope from the individual job to *cluster-level orchestration*. Our key insight is that the dependency bubbles inherent to one job can be effectively utilized to serve another. ROLLMUX exploits this by orchestrating multiple RL jobs into a *co-execution group*, tightly “weaving” their rollout and training phases across the two resource pools (Figure 1-bottom). This *co-scheduling* approach enables efficient *time-multiplexing* of both rollout and training GPUs, maximizing utilization while preserving the synchronous dependencies required for on-policy learning.

While intuitive, realizing this co-scheduling benefit in production is non-trivial due to three primary challenges. (C1) First, production RL jobs exhibit *extreme workload heterogeneity* in model sizes (3B–32B), response lengths, and interaction patterns (Figure 2), leading to highly diverse phase durations and resource demands. Naive time-multiplexing of such diverse workloads results in severe interference: for instance, pairing two rollout-heavy jobs creates a bottleneck that substantially delays both (Figure 3). Identifying an optimal, interference-free schedule for these heterogeneous workloads formulates a Job Shop Scheduling problem [23], which is known to be NP-hard. (C2) Second, unlike standard deep learning workloads with stable iteration times, RL rollouts are *highly stochastic*: LLM generation follows a *long-tailed distribution* [12, 17, 59], where a few straggler requests can *unpredictably prolong* phase durations, rendering static plans obsolete. (C3) Third, efficient time-multiplexing is fundamentally constrained by context-switching costs. RL post-training is inherently *stateful*, requiring the management of hundreds of gigabytes of model weights and optimizer states (Table 2). Repeatedly loading these massive states over a bandwidth-limited cross-cluster network induces prohibitive *cold-start latencies*—up to 80 seconds per switch (Figure 4)—which can easily offset the throughput gains of co-scheduling.

ROLLMUX addresses these challenges via a holistic algorithm-system co-design. At the core is a *near-optimal* scheduling algorithm. The goal is to *minimize the total resource provisioning cost*—thereby minimizing dependency bubbles—while adhering to job-specified SLOs, defined as the acceptable slowdown relative to *solo execution* (Figure 1-top). To tackle the intractability of heterogeneous scheduling (C1), ROLLMUX decomposes the global optimization problem into two tractable decisions: (1) *inter-group scheduling*, which identifies jobs for group co-execution and, (2) *intra-group scheduling*, which orchestrates execution sequences within a co-execution group. When a new job arrives, the *inter-group scheduler* scans for an existing co-execution group where the

job can be placed without violating the SLOs of any group member. Among all SLO-compliant placement options, it selects the one that yields the *minimum marginal provisioning cost*; if no such group exists, it provisions a *new, isolated group*. Within each group, the *intra-group scheduler* employs a round-robin schedule, a policy we prove is *optimal* for minimizing dependency bubbles in this context (§4.3).

To handle runtime stochasticity (C2), ROLLMUX adopts a two-pronged strategy combining *conservative admission control* with *long-tail migration*. For inter-group placement, the scheduler assumes a worst-case scenario where all responses reach the maximum token length, ensuring that SLO guarantees hold even under maximum load (§4.2). At runtime, the intra-group scheduler dynamically adapts to the observed response distribution. It employs *long-tail migration*, opportunistically moving long responses to a small subset of devices to free up the majority of the rollout pool, thereby allowing the next job to begin pipelined execution immediately (§4.3).

Finally, to mitigate the prohibitive switching overheads (C3), ROLLMUX implements a *warm-start* mechanism. By rightsizing co-execution groups to fit within the host memory of the worker nodes, ROLLMUX ensures that all necessary job states, such as model weights, optimizer states, and execution contexts, remain *cached* in host memory. When a context switch is required, the worker simply loads the cached state from host memory to the GPU, rather than fetching it across the slow cluster interconnect or from disk. This optimization reduces context switching latency by two orders of magnitude (Figure 4), making fine-grained time-multiplexing practical.

To enforce these fine-grained schedules, ROLLMUX introduces a *phase-centric control model* that treats individual RL phases as first-class schedulable entities. This abstraction exposes the job’s internal dependency graph to the scheduler, transparently managing job state loading required for the warm-start mechanism. Furthermore, ROLLMUX optimizes cross-cluster model synchronization via a *topology-aware* broadcast scheme. It pipes a single model copy across the slow inter-cluster link through parallel point-to-point streams, then utilizes high-speed local fabrics (e.g., NVLink or InfiniBand) for intra-cluster broadcasting, effectively eliminating the synchronization bottleneck in disaggregated setups.

We implemented ROLLMUX atop ROLL [43] and evaluated it in a production-scale disaggregated testbed comprising a rollout pool of 328 H20 GPUs and a training pool of 328 H800 GPUs. End-to-end replays of production workloads on these two clusters reveal that ROLLMUX reduces total resource provisioning costs by up to  $1.84\times$  compared to naive disaggregation and  $1.38\times$  compared to the state-of-the-art veRL [41] baseline, all while maintaining 100% SLO attainment (§7.4). Large-scale trace-driven simulations further confirm that ROLLMUX’s inter- and intra-group scheduling combined operates within 6% of the theoretical optimum identified via brute-force search (§7.5).

## 2 Background and Motivation

**RL Post-Training Workload Characterization.** RL post-training has evolved into a cornerstone workload for modern AI infrastructure. In our production clusters, the volume of RL jobs nearly tripled within six months, growing from 5k monthly jobs in April to over 14k in September 2025, driven largely by the need to instill complex reasoning capabilities in LLMs [3, 6, 8, 39, 43]. Unlike LLM pre-training, which is a uniform compute stream, the standard RL post-training workflow comprises repeated cycles across three phases with distinct resource bottlenecks. (1) **Rollout:** The actor LLM generates responses for a batch of input prompts, which are subsequently evaluated to collect the reward feedback. This phase is characterized by *high memory bandwidth pressure* due to KV-cache operations but relatively *low arithmetic intensity*. On high-end training GPUs, this results in severe compute underutilization [10, 17, 58]. (2) **Training:** The actor LLM’s parameters are optimized based on the reward feedback. This phase is highly *compute-intensive* and requires massive floating-point throughput and high-bandwidth interconnects (e.g., NVLink and InfiniBand) for gradient aggregation [42]. (3) **Synchronization:** Updated parameters must be broadcast from training workers to rollout workers. This phase is *network-bound*, often becoming a bottleneck when workers are distributed across different physical domains.

**The Case for Disaggregated RL.** The divergent resource requirements of rollout and training create a fundamental inefficiency in traditional *monolithic, co-located deployments* [41], where rollout and training are time-multiplexed on a single cluster of homogeneous, compute-optimized GPUs (e.g., NVIDIA H100/H800). This forces the memory-bound rollout to run on expensive high-FLOPS hardware, leading to significant resource mismatches and increased total cost of ownership (TCO) [49, 58].

Disaggregation offers a promising solution to address this mismatch [10, 44, 49, 58]. In this setup, the RL workload is disaggregated across two purpose-specific resource pools (Figure 1-bottom): training is retained on a pool of costly, high-FLOPS GPUs (e.g., H100/H800), while rollout is offloaded to a cluster of cost-effective, inference-optimized GPUs (e.g., H20), which offer high HBM capacity and bandwidth at only a fraction of the cost (Table 1). Compared to monolithic provisioning, disaggregation aligns hardware capabilities with phase characteristics, offering superior TCO *in theory*.

**Dependency Bubbles.** While disaggregation addresses hardware mismatches, its efficiency can be significantly undermined by *dependency bubbles*. State-of-the-art RL post-training relies on *synchronous, on-policy* algorithms to ensure training stability and model quality [6, 15, 35, 39]. This synchronization constraint mandates that the training pool must remain idle while waiting for the rollout pool to generate fresh experiences, and conversely, the rollout pool must remain idle

Accelerator	Comp. (TFLOPS)	HBM Cap. (GB)	HBM B/w (TB/s)	Cost (\$/h) [61]
H20	148	96	4.0	1.85
H800	989.5	80	3.35	5.28

Table 1: Performance specifications and cost-effectiveness of the GPUs used in our disaggregated clusters [58, 61].

while waiting for the training pool to update parameters, as illustrated in Figure 1-top.

Existing systems such as AReaL [10, 49], StreamRL [58], and AsyncFlow [16] attempt to eliminate these bubbles by adopting *asynchronous, off-policy* algorithms. However, decoupling rollout from training introduces *sample staleness*, which frequently compromises model convergence and final accuracy, rendering these solutions unsuitable for tasks demanding strict on-policy performance. Consequently, production deployments often revert to synchronous execution at the cost of significant resource idleness. In fact, our evaluation in §7.4 reveals that the idle time induced by dependency bubbles forces disaggregated setups to incur *even higher provisioning costs* (\$0.94k/h) than the hardware-mismatched co-located baselines (\$0.71k/h), despite using cheaper GPUs for rollout. Thus, without a scheduling mechanism to reclaim this lost capacity, the theoretical TCO benefits of disaggregation are effectively nullified by system-level inefficiencies.

## 3 Scheduling Opportunities and Challenges

In this section, we identify the opportunities for mitigating dependency bubbles in disaggregated RL post-training through cluster-level scheduling. We then examine the algorithmic and system-level challenges in realizing these opportunities.

### 3.1 The Co-Scheduling Opportunity

From the perspective of a single job, the dependency bubbles described in §2 are unavoidable without violating the synchronization requirements of on-policy algorithms. However, in *shared, multi-tenant* clusters running diverse RL workloads, these individual inefficiencies represent available capacity that can be reclaimed through global orchestration.

Our key insight is that the idle resources constituting one job’s dependency bubbles can be utilized to execute the active phase of another. By orchestrating jobs into *co-execution groups*, the scheduler can tightly “weave” together their workflows, ensuring that the compute-intensive training phase of one job executes in parallel with the memory-bound rollout phase of another (Figure 1-bottom). This interleaved execution pattern effectively hides dependency bubbles, allowing the system to simultaneously saturate both the cost-effective rollout pool and the high-performance training pool, thereby maximizing cluster-wide efficiency without compromising the synchronization requirements of on-policy learning.



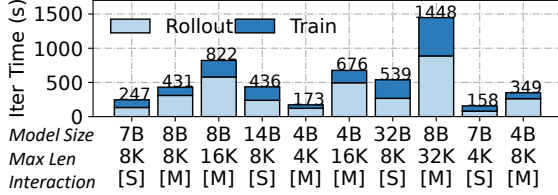


Figure 2: Top 10 popular RL post-training workloads in our production cluster: jobs’ phase durations are highly diverse. [S], [M] refers to single/multi-turn interaction during rollout.

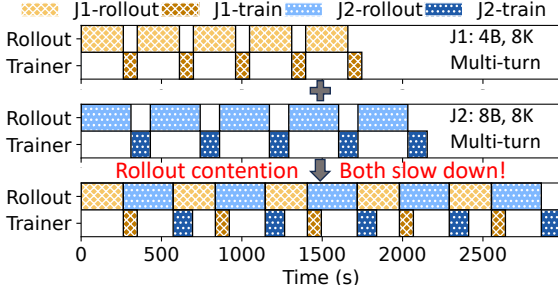


Figure 3: A bad case of naive time-multiplexing: two rollout-heavy jobs compete for a rollout node and both slow down.

### 3.2 Challenges

However, fully unlocking the benefits of co-scheduling at production scale is non-trivial due to three primary challenges.

**C1: Workload Heterogeneity and Scheduling Intractability.** First, realizing efficient co-scheduling is complicated by the extreme diversity of production RL workloads. As depicted in Figure 2, jobs in our cluster span a wide spectrum of model sizes (3B–32B), response lengths (4k–32k tokens), and interaction modes (single- vs. multi-turn). This heterogeneity manifests as highly variable phase durations, ranging from 50s to over 900s, and significant *phase skew*; for instance, multi-turn agentic workloads often exhibit rollout phases that are  $3\times$  to  $4\times$  longer than their corresponding training phases.

Consequently, naive time-multiplexing often proves detrimental due to resource contention. For example, arbitrarily pairing two rollout-heavy jobs creates a bottleneck on the inference nodes, forcing both jobs to stall. As illustrated in Figure 3, such contention results in severe performance degradation, slowing down concurrent jobs by  $1.40\times$  and  $1.64\times$ , respectively. To avoid such interference, the scheduler must identify optimal packings that satisfy strict performance SLOs. However, mapping these heterogeneous, phase-skewed workloads to available resources formulates a Job Shop Scheduling problem [13, 23], which is known to be NP-hard even under the simplifying assumption of deterministic phase durations.

**C2: Stochastic Runtime and Skewness Bubbles.** Second, efficient scheduling orchestration is complicated by the inherent *stochasticity* of RL workloads. Unlike pre-training, RL rollout creates a dynamic workload where execution time depends on the variable length of generated responses, which

Model Size	3B	7B	14B	32B
Rollout	113.4	275.7	445.4	490.3(TP=2)
Train	156.2	240.0(TP=2)	456.1(TP=2)	520.4(TP=4)

Table 2: Memory footprint (GB) required for caching rollout or training actors on an 8-GPU node across model sizes.

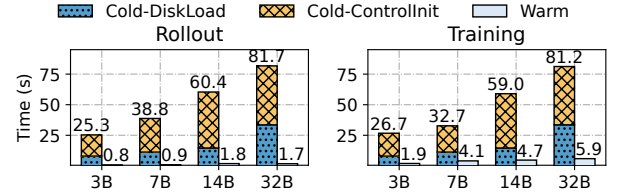


Figure 4: Cold and warm start latency for rollout (left) and training (right) across model sizes on an 8-GPU node.

follows a *long-tail distribution* (Figure 11). This introduces two distinct system challenges. First, the workload is *non-stationary*: the distribution of response lengths drifts across iterations as the model updates, with a few "straggler" requests frequently reaching the maximum token limit [12, 58, 59]. Because training computation scales linearly with token count, this rollout variance propagates to the subsequent training phase, rendering static orchestration planning obsolete. Second, long-tail responses result in *skewness bubbles* during the rollout phase [58, 59]. Within a rollout batch, early-finishing GPUs must idle while waiting for stragglers, effectively serializing the batch completion. This forces the scheduler to solve a dynamic variant of the Job Shop Scheduling problem, where task durations are unpredictable, time-varying, and prone to significant internal resource fragmentation.

**C3: Context Switching Overhead and Memory Residency.** Third, the granularity of efficient time-multiplexing is fundamentally constrained by the cost of inter-job context switching. Unlike stateless LLM inference [11], RL post-training is inherently *stateful*, managing a massive working set that includes large model weights, optimizer states, and complex execution contexts such as dataset pipelines and environment states. Reconstructing these states from scratch upon every switch—known as a *cold start*—incurs prohibitive overheads due to both data- and control-plane re-initialization. As shown in Figure 4, cold-starting a rollout or training phase on an 8-GPU node (H20 for rollout and H800 for training) takes up to 80 seconds, degrading end-to-end throughput by up to 45%.

Furthermore, unlike serverless systems that can mitigate cold starts via high-speed RDMA state transfers [11, 53, 55], disaggregated RL setups are bottlenecked by limited cross-cluster Ethernet bandwidth. Consequently, the only viable mechanism for rapid switching is a *warm-start* strategy, where job states remain cached in local host DRAM. While this approach significantly reduces switching latency by up to  $48\times$  (Figure 4), it imposes severe memory pressure. Since a single phase’s state consumes hundreds of gigabytes (Table 2), even high-memory nodes (1–2 TB) are strictly limited to a

residency of two to five concurrent jobs. This creates a tight *residency constraint*, forcing the scheduler to optimize for utilization within a bounded memory budget.

## 4 The ROLLMUX Scheduling Design

We present ROLLMUX, a cluster scheduling framework that reclaims dependency bubbles for disaggregated RL post-training through cross-cluster orchestration. ROLLMUX adopts a holistic algorithm-system co-design: we decouple the logical scheduling policy (§4) from the underlying execution plane (§5). This section details the core scheduling algorithms.

### 4.1 Co-Execution Group

ROLLMUX’s scheduling objective is to minimize total resource provisioning cost—and thereby minimize dependency bubbles—while strictly adhering to job performance SLOs and node memory constraints. To achieve this, we introduce the *co-execution group* abstraction. A co-execution group is a set of jobs that *share* a specific pair of rollout and training resource pools via *time-multiplexing*. Within a group, all rollout phases execute on the group’s assigned rollout workers, and all training phases execute on the group’s training workers. By partitioning jobs into *disjoint groups*, ROLLMUX transforms the intractable global co-scheduling problem into a collection of independent, parallel sub-problems within groups.

This group-based scheduling directly addresses two primary challenges identified in §3.2. First, by decomposing the cluster-wide search space into smaller, isolated groups, ROLLMUX ensures that scheduling decisions remain *computationally tractable* at production scale, even with thousands of concurrent jobs (C1). Second, the group abstraction serves as a strict *locality domain*. By pinning jobs to specific sets of nodes, ROLLMUX enforces the residency constraint: it ensures that the massive working sets (weights and optimizer states) of all group members remain resident in host DRAM. This guarantees that context switches can be served via high-speed local memory transfers (warm starts) rather than slow disk or cross-cluster fetches (C3).

The co-execution group abstraction naturally leads to a *two-tier scheduling hierarchy*: (1) *inter-group scheduling* (§4.2), which assigns arriving jobs to groups to minimize provisioning costs while satisfying memory and SLO constraints, and (2) *intra-group scheduling* (§4.3), which orchestrates the runtime execution order of job phases within a group to minimize dependency bubbles.

### 4.2 Inter-Group Scheduler

**Problem Formulation.** We model the cluster as a collection of disjoint co-execution groups. We define a co-execution

group  $G$  as a tuple  $(J_G, R_G, T_G, \Phi_G)$ , where  $J_G$  is the set of active RL jobs in the group,  $R_G$  and  $T_G$  denote the sets of rollout (e.g., H20) and training (e.g., H800) GPUs provisioned for the group, and  $\Phi_G = \{P_j\}_{j \in J_G}$  is the set of resource placements, where  $P_j$  specifies the exact subset of rollout and training nodes job  $j$  is pinned to. This pinning  $P_j$  strictly determines where the job’s state is cached to enable its warm start.

The inter-group scheduler solves the following *online placement problem*: upon the arrival of a job  $j$ , it must assign  $j$  to a co-execution group—either an existing one or a newly created one—and allocates specific resource placement  $P_j$ . To make optimal placement, we define the provisioning cost of a group,  $\text{Cost}(G)$ , as the aggregate hourly price (Table 1) of all allocated GPUs in its rollout and training pools ( $R_G$  and  $T_G$ ). The scheduler’s objective is to minimize the *marginal provisioning cost*  $\Delta$  incurred by admitting job  $j$ :

$$\min_G \Delta = \text{Cost}(G') - \text{Cost}(G),$$

where  $G'$  represents the group’s state after accommodating job  $j$ . This formulation naturally encourages “packing” jobs into existing dependency bubbles (where  $\Delta = 0$ ) over provisioning new hardware (where  $\Delta > 0$ ). The placement decision is subject to two critical constraints:

**1) Memory Residency.** To guarantee warm starts (C3), the aggregate working set of all jobs pinned to a specific node must not exceed that node’s host memory capacity.

**2) SLO Attainment.** The placement must satisfy the performance SLOs of both the new job and all existing jobs. The SLO is defined by each job as the *tolerance for co-execution slowdown* (e.g.,  $1.1 \times$ ) relative to solo execution.<sup>1</sup> Formally, for every job  $k$  in the updated group  $G$ , we require:

$$T_k^{\text{co-exec}} \leq \text{SLO}_k \times T_k^{\text{solo}}.$$

Here,  $T_k^{\text{solo}}$  is the estimated iteration time when job  $k$  is running alone (Figure 1-top), which is simply the sum of its rollout and training phase durations;  $T_k^{\text{co-exec}}$  is the expected iteration time under co-execution, which is derived by simulating the intra-group schedule (§4.3).

**Making Placement Decisions.** Navigating the search space to find an optimal placement is non-trivial due to both workload heterogeneity (C1) and runtime stochasticity (C2). ROLLMUX addresses these complexities with three strategies.

**1) Handling Stochasticity via Conservative Planning.** To guarantee SLO compliance despite the volatile, unpredictable job execution time (C2), ROLLMUX decouples admission control from runtime optimization. The inter-group scheduler acts as a “gatekeeper” that makes placement decisions based on *worst-case execution bounds*. Specifically, for an arriving job  $j$ , we estimate its phase durations ( $T_j^{\text{roll}}$  and  $T_j^{\text{train}}$ ) assuming that every generated response reaches the *maximum token limit* defined in the job configuration. By planning against this upper bound, we ensure that the chosen placement satisfies the SLO constraints even under the most adverse stochastic

<sup>1</sup>We assume a tight SLO, e.g., tolerance for up to  $2 \times$  slowdown.

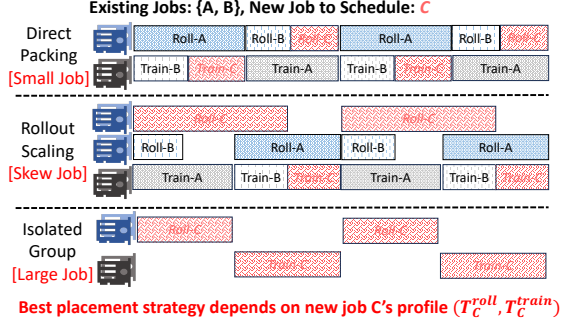


Figure 5: Placement strategies of the inter-group scheduler.

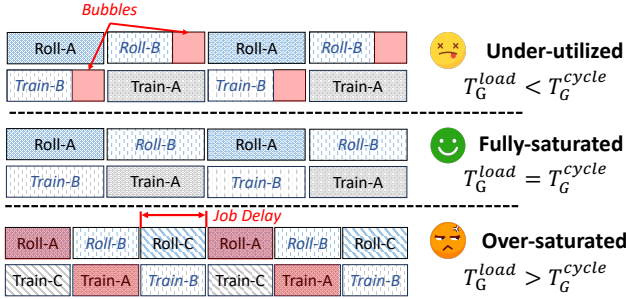


Figure 6: Status of a co-execution group. ROLLMUX only places new jobs into under-utilized groups with unsaturated dependency bubbles and avoids creating over-saturated groups.

conditions. If the actual runtime durations are shorter, which is typical, the intra-group scheduler dynamically reclaims the resulting slack to improve utilization (§4.3).

**2) Optimal Placement Search.** With these conservative estimates, ROLLMUX performs a global search to minimize the marginal provisioning cost  $\Delta$ . For each arriving job, the scheduler iterates through all candidate groups and evaluates three placement strategies:

- **Direct Packing:** Inserting the job into existing dependency bubbles within a group without provisioning new resources (Figure 5-top). This maximizes utilization of already-paid-for capacity.
- **Rollout Scaling:** If a group has available training capacity but is bottlenecked on inference, which is common with rollout-heavy agentic workloads, ROLLMUX scales up the group’s rollout pool by provisioning *just enough* rollout nodes to accommodate the new job (Figure 5-middle).
- **Isolated Provisioning:** As a fallback, ROLLMUX provisions a new, isolated group for the new job (Figure 5-bottom).

The scheduler iterates through these strategies<sup>2</sup>, selecting the *valid placement* that minimizes the marginal cost  $\Delta$  while satisfying both the memory residency and SLO constraints.

**3) Pruning Saturated Groups.** To ensure this search re-

<sup>2</sup>To avoid the significant overhead of reconfiguring distributed parallel groups, ROLLMUX does not scale the training pool but simply adjusts the arriving job’s data parallelism degree to match the training pool size.

mains tractable at production scale (C1), ROLLMUX proactively prunes the search space. Before evaluating specific placements, the scheduler filters out groups that are already *saturated*, where the aggregate job load reaches the group’s bottleneck resource capacity, and adding more work to the group would lead to performance degradation.

Formally, for a group  $G$ , let  $T_G^{\text{cycle}} = \max_{j \in J_G} T_j^{\text{solo}}$  be the natural cycle iteration time dictated by the *longest job* in the group. We define the group’s bottleneck load  $T_G^{\text{load}}$  as the total time required to process all phases on the bottleneck node. Since all training nodes have identical phases, while rollout nodes may differ (see Figure 5), we have

$$T_G^{\text{load}} = \max \left( \sum_{j \in J_G} T_j^{\text{train}}, \max_{n \in \Phi_G} \left( \sum_{j \text{ on node } n} T_j^{\text{roll}} \right) \right).$$

If  $T_G^{\text{cycle}} \geq T_G^{\text{load}}$ , the group is saturated, containing no “slack” to absorb new work (Figure 6). Such groups are pruned immediately as any further addition would force delays.

**Algorithm Summary.** We integrate these strategies into a unified online scheduling logic detailed in Algorithm 1. The algorithm takes a new job  $j$  and the set of existing groups as input. To find the optimal placement, the algorithm first iterates through all existing groups (line 3), discarding those that are already saturated (line 4). For each remaining candidate group, it evaluates potential placement strategies for the job (direct packing or rollout scaling); placements that would violate memory constraints (line 8) or SLO constraints (line 10) are discarded. The algorithm evaluates the marginal cost for each feasible placement (lines 6–12) and updates its records if the placement leads to a lower cost (lines 13–14). Finally, the algorithm compares its records against the baseline cost of provisioning a fresh, isolated group (lines 15–17) and returns the group and job placement that yield the lowest costs.

The algorithm allows for highly efficient decision-making. Since the number of placement strategies per group is small, the search complexity is *linear* with respect to the number of active groups. As empirically demonstrated in §7.5, this heuristic allows the scheduler to make optimal decisions in sub-seconds even in clusters with thousands of jobs.

### 4.3 Intra-Group Scheduler

Once the inter-group scheduler assigns a job to a group, the intra-group scheduler is responsible for orchestrating the runtime execution sequence. Its primary objective is to maximize resource utilization—and thereby minimize dependency bubbles—within the assigned resource pools.

**The Round-Robin Policy.** ROLLMUX employs a *cyclic round-robin schedule*. Within a co-execution group, the scheduler defines a *meta-iteration* in which every active job executes exactly one rollout phase and one training phase (Figure 1). These phases are orchestrated sequentially on their assigned resource pools. For example, in a group with jobs  $\{A, B\}$ , the rollout pool executes  $\text{Roll}_A \rightarrow \text{Roll}_B$ , while the



### Algorithm 1 Inter-Group Scheduling Algorithm

**Require:** Job to schedule  $j$ , all existing groups  $\{G_i\}_{i=1}^n$ .  
**Ensure:** Best group  $G^*$ , best placement  $P_j^*$ .

```

1: procedure SCHEDULE( $j, \{G_i\}_{i=1}^n$ )
2:    $\Delta^* \leftarrow \infty, G^* \leftarrow \text{None}, P_j^* \leftarrow \text{None}$  ▷ Initialize best values.
3:   for each group  $G$  in  $\{G_i\}_{i=1}^n$  do ▷ Try all existing groups.
4:     if  $T_G^{\text{load}} \geq T_G^{\text{cycle}}$  then ▷ Skip saturated groups.
5:       continue
6:      $\mathcal{P} \leftarrow \text{GENERATEPLACEMENTS}(G)$ 
7:     for each resource placement  $P_j$  in  $\mathcal{P}$  do
8:       if  $j.\text{mem\_req} \geq \min_{\text{node} \in P_j} (\text{node}.\text{mem\_avail})$  then
9:         continue
10:      if exists  $k \in \{j\} \cup J_G$ , s.t.,  $T_k^{\text{co-exec}} > \text{SLO}_k \times T_k^{\text{solo}}$  then
11:        continue
12:       $\Delta \leftarrow \text{Cost}(G \cup \{(j, P_j)\}) - \text{Cost}(G)$ 
13:      if  $\Delta < \Delta^*$  then
14:        Update  $\Delta^* \leftarrow \Delta, G^* \leftarrow G, P_j^* \leftarrow P_j$ 
15:       $\Delta \leftarrow \text{Cost}(\{j\}, \{j\})$  ▷ Try to place  $j$  in a new group.
16:      if  $\Delta < \Delta^*$  then
17:        Update  $\Delta^* \leftarrow \Delta, G^* \leftarrow \{j\}, P_j^* \leftarrow \{j\}$ 
18:  return  $G^*, P_j^*$ 

```

training pool executes  $\text{Train}_A \rightarrow \text{Train}_B$ .

While simple, this policy is *optimal* under the preconditions enforced by ROLLMUX. Recall that the inter-group scheduler (§4.2) proactively prunes any group where the aggregate load exceeds the natural cycle time ( $T_G^{\text{load}} > T_G^{\text{cycle}}$ ). For the remaining unsaturated groups, their optimality is provable.

**Theorem 1** (Utilization Optimality) *For any unsaturated group  $G$ , a meta-iteration schedule that executes each job’s phases exactly once in a round-robin order maximizes the aggregate utilization of both rollout and training pools.*

*Proof Sketch.* The optimality rests on the definition of an *unsaturated group*: the bottleneck node’s total workload  $T_G^{\text{load}}$  is no more than the longest job’s standalone cycle time  $T_G^{\text{cycle}}$ . Intuitively, this implies that we can pack all other jobs’ corresponding phases (e.g., their rollout phases) into the longest job’s dependency bubbles (e.g., its idle rollout nodes during training). This ensures a round-robin cycle that executes each job’s phases exactly once to complete in time  $T_G^{\text{cycle}}$ . We then empirically show any deviation from this simple schedule is *suboptimal*. (1) *Executing less is impossible*: Omitting any job from the cycle leads to more bubbles and starvation, which is trivially non-optimal. (2) *Executing more is inefficient*: Repeating any job phase prolongs the cycle time as the added phase can only start after finishing the slowest job. However, this added duration is disproportionately larger than the useful work added, leading to a net decrease in resource utilization.

Therefore, the round-robin schedule, which executes all required work in the shortest possible cycle time, is utilization-optimal. We provide a formal proof in Appendix 9.

**Long-Tail Migration.** While the round-robin schedule is optimal for deterministic workloads, production RL phases are highly stochastic (C2). Specifically, rollout durations follow a *heavy-tailed distribution* where the completion time of an input batch is dictated by a small number of “straggler” re-

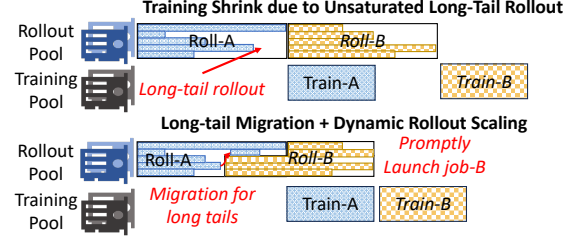


Figure 7: Long-tail migration effectively handles dynamism.

sponses that reach maximum token limits [12, 17, 58, 59]. This phenomenon creates significant *intra-phase fragmentation*: as the majority of responses finish early, most GPUs in the rollout pool idle wait for the few stragglers to complete, creating “skewness bubbles” (Figure 7-top).

To reclaim this fragmented capacity, ROLLMUX employs *long-tail migration* to dynamically adapt the schedule at runtime. The intra-group scheduler continuously monitors the progress of active rollout phases. When a phase enters a *tail-bound state*, triggered when a threshold of responses (e.g., 80%) have completed, the system interrupts the execution, consolidates the remaining long-tail responses onto a small subset of workers, and immediately starts the *next* job’s rollout phase on the newly freed rollout GPUs (Figure 7-bottom). This mechanism effectively pipelines the tail of one job with the head of the next, ensuring high utilization and faster completion.

## 4.4 Generality and Composability

ROLLMUX is agnostic to the specific RL algorithm employed. Its scheduling mechanism generalizes to diverse on-policy RL algorithms, including PPO [37], GRPO [39], and DAPO [54]. While primarily designed for on-policy algorithms, ROLLMUX remains applicable to off-policy jobs (e.g., one-step off-policy [58]) that exhibit structural dependency bubbles due to insufficient overlap between training and rollout. Moreover, ROLLMUX’s cluster-level orchestration is orthogonal to intra-job optimizations. Techniques such as parameter relocation [27], request-level tail batching [12], and speculative decoding [4, 17, 22, 34] operate within the scope of a single job or phase, making them fully composable with ROLLMUX.

## 5 The ROLLMUX Execution Plane

The scheduling policies described in §4 provide a theoretical blueprint for reclaiming dependency bubbles. However, realizing these gains in production clusters requires an execution plane capable of enforcing fine-grained decisions. This section details the system mechanisms that bridge this gap. We focus on two implementation challenges: (1) enabling rapid context switching via a phase-centric control model (§5.1), and (2) mitigating cross-cluster bandwidth bottlenecks via

topology-aware model synchronization (§5.2).

## 5.1 Phase-Centric Control

Conventional deep learning schedulers view the “job” as the atomic unit of resource allocation. This coarse granularity is insufficient to interleave distinct phases of different jobs on the same hardware. To address this, ROLLMUX introduces a *phase-centric execution model* that elevates individual RL phases to first-class schedulable entities.

**Declarative Phase Management.** We model each RL job as a dependency graph of phases. After a one-time initialization (Init) of the job states (e.g., models, datasets), the job enters a cyclic dependency loop: Rollout → Train → Sync. ROLLMUX exposes this internal structure to the scheduler via a declarative Python API. Users simply annotate their phase functions with a `@rollmux.phase` decorator, which injects a *transparent runtime shim* to manage the execution lifecycle. When a phase is invoked, this shim first blocks execution until it acquires a run permit from the intra-group scheduler. Upon approval, it performs a *warm start* by loading the phase’s resident working set from host DRAM into GPU memory. Once the user function completes, the shim immediately offloads the updated state back to host memory and releases the GPU resources, making the hardware instantly available for the next phase in the group’s queue. Crucially, ROLLMUX optimizes this switching process by decoupling data plane state from control plane context. Naively terminating a process after a phase completes would force the system to tear down and rebuild expensive control plane (e.g., NCCL communicators, environment handles) upon every switch. Instead, ROLLMUX employs a *lightweight suspension* strategy: after offloading, the shim places the process into a sleep loop while retaining its control plane context without consuming GPU resources. On the next wake-up, resuming the phase only requires reloading its cached state onto the GPU, avoiding expensive cold starts from disk and control-plane re-initialization.

**Runtime Hooks.** Finally, the system exposes a runtime hook interface `@rollmux.runtime_hook`. This interface serves two critical roles. First, it drives the round-robin schedule: the intra-group scheduler maintains a FIFO queue for each worker node. When a job’s phase completes, the hook signals the scheduler to enqueue the job’s next phase onto the alternate resource pool’s queue (e.g., moving from rollout to training) and starts the next waiting phase on the now-idle resources. Second, it enables the long-tail migration (§4.3). By exposing internal token generation progress, the hook allows the scheduler to detect tail-bound states and externally trigger migration, dynamically reconfiguring resources in real-time.

## 5.2 Topology-Aware Model Synchronization

To mitigate the cross-cluster bandwidth bottleneck, ROLLMUX employs a *topology-aware* communication strategy

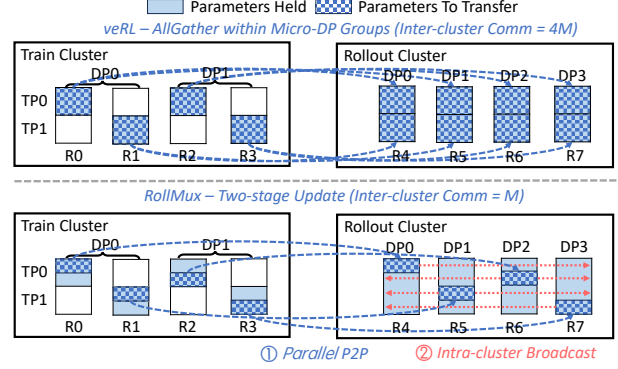


Figure 8: A synchronization example from the training cluster (TP=2, DP=2) to the rollout cluster (DP=4), where ROLLMUX sends exactly one copy across the cross-cluster network.  $M$  denotes the number of total parameters.

to efficiently synchronize model parameters from the training cluster to the rollout cluster. State-of-the-art RL frameworks (e.g., veRL [41]) rely on flat collective operations like AllGather to propagate model updates. While efficient in monolithic clusters, this approach is pathological in disaggregated setups. It treats the slow cross-cluster Ethernet link and the fast intra-cluster InfiniBand/NVLink fabric as a single uniform network. Consequently, it forces every rollout worker to independently fetch a full copy of the model parameters over the slow cross-cluster link (Figure 8-top), causing a severe bottleneck while leaving local high-speed fabrics idle.

ROLLMUX eliminates this inefficiency by replacing the flat collective with a *hierarchical two-stage transfer*. ① In the first stage (**inter-cluster scatter**), ROLLMUX partitions the updated model into  $N$  disjoint shards, where  $N$  is the number of training GPUs. Each training GPU transmits a unique shard to a corresponding rollout GPU via parallel point-to-point (P2P) streams. This ensures that *exactly one full copy* of the model traverses the slow cross-cluster link. ② In the second stage (**intra-cluster broadcast**), the receiving GPUs immediately disseminate their shards to all other rollout workers using the high-bandwidth InfiniBand/NVLink fabric. This two-stage pipeline effectively mitigates the slow cross-cluster bottleneck, minimizing overall synchronization time and fully utilizing network hierarchies.

## 6 System Implementation

We implemented ROLLMUX as a fully functional cluster scheduling framework atop ROLL [43]. The system comprises approximately 5.2k lines of code (LoC), written primarily in Python for controllers and C++ for communication modules.

**Workflow.** The system operation follows the closed loop illustrated in Figure 9. Upon job submission, ROLLMUX first launches a lightweight profiler (①) to generate worst-case du-



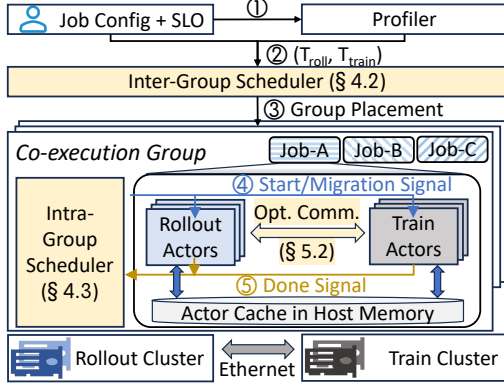


Figure 9: ROLLMUX system architecture.

ration estimates for the job’s rollout and training phases (②). These estimates are fed into the inter-group scheduler (③), which identifies the optimal co-execution group and resource placement to minimize marginal cost (§4.2). Once placed, the job comes under the control of the intra-group scheduler (§4.3). This runtime controller orchestrates the round-robin meta-iteration (④), enforcing the phase-centric state management and triggering long-tail migrations based on real-time feedback from the runtime hooks (§5.1). Once a phase completes, ROLLMUX offloads its state to the actor cache in host DRAM, releases GPU resources, and subsequently launches the next job’s waiting phase from the scheduler queue (⑤).

**Isolation and Fault Tolerance.** To ensure production-grade reliability, ROLLMUX enforces strict fault isolation. Each job owns a dedicated Ray [28] instance with its isolated runtime environment. Jobs communicate exclusively with the scheduler via the Redis [36] channel and never directly with one another. Consequently, a crash in one job is fully contained within its pod, preventing error propagation and ensuring the stability of other jobs within the same co-execution group.

## 7 Evaluation

We evaluate ROLLMUX to answer the following key research questions. **RQ1 (Co-Execution Efficacy):** How effectively does ROLLMUX’s co-execution mechanism manage job groups with diverse phase profiles (§7.2)? **RQ2 (Performance Breakdown):** What are the individual contributions of ROLLMUX’s key optimizations (i.e., long-tail migration, topology-aware model sync) to overall system performance (§7.3)? **RQ3 (Performance at Scale):** How does ROLLMUX perform under a real-world production workload trace, in terms of cluster-level utilization and provisioning cost (§7.4)? **RQ4 (Scheduling Quality):** How efficient and how close to optimal is ROLLMUX’s scheduler (§7.5)?

Job	Turns	Model	Len <sup>3</sup>	Bsz	N <sub>T</sub>	N <sub>R</sub>
Type-A	Single-Turn	Qwen-2.5-7B	8K	256	8	8
Type-B	Single-Turn	Qwen-2.5-14B	8K	256	8	8
Type-C	Single-Turn	Qwen-2.5-32B	8K	256	16	16
Type-D	Multi-Turn	Qwen-3-8B	8K*	256	8	8
Type-E	Multi-Turn	Qwen-3-14B	16K*	64	8	8

Table 3: Job configurations in experiments,  $N_T, N_R$  are the corresponding numbers of training/rollout GPUs.

## 7.1 Experimental Setup

**Cluster Setup.** Our experimental testbed consists of two geo-distributed, heterogeneous clusters, where the training cluster (Cluster-T) is equipped with compute-optimized NVIDIA H800 GPUs, while the rollout cluster (Cluster-R) is composed of cost-effective H20 GPUs. The internal fabric of each cluster is a high-speed 400 Gbps InfiniBand network. However, the two clusters are connected via a bandwidth-constrained 20 Gbps Ethernet link. Table 1 details the hardware specifications and hourly costs, where an H800 GPU is  $2.85\times$  more expensive than an H20 GPU.

**Workloads.** We construct our workloads based on real-world traces collected from a production cluster. For **micro-benchmarks** (§7.2–§7.3), we define a suite of five representative job types (Table 3) using Qwen [50] models (7B–32B) with varying batch sizes, sequence lengths, and rollout/training GPUs, covering both single-turn RLVR (on DeepMath-103K [39] dataset) and multi-turn agentic reasoning (on Math-Orz57K [21] dataset). For **at-scale evaluation** (§7.4), we replay a two-week trace comprising 200 heterogeneous jobs. This trace features high variance in model sizes (3B–32B) and diverse datasets spanning mathematics [39], software engineering [31], games [7], and other in-house datasets.

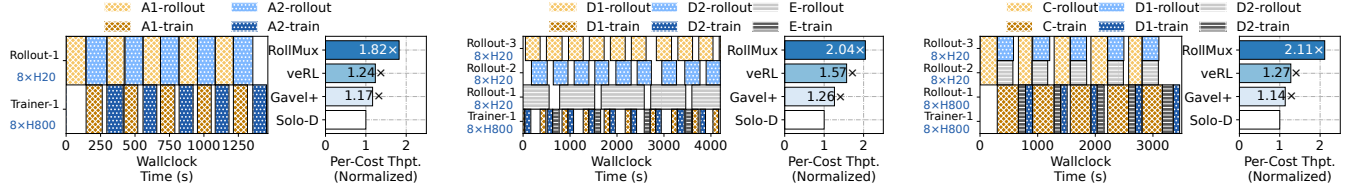
**Baselines.** We compare ROLLMUX against three baselines.

- *Solo Disaggregation (Solo-D):* The standard disaggregation practice where jobs are executed on dedicated rollout and training pools without time-multiplexing.
- *Co-location (veRL [41]):* The traditional monolithic approach, where all phases execute on the high-performance training cluster (Cluster-T) using the popular veRL framework. It avoids network bottlenecks but suffers from hardware resource mismatch.
- *Gavel+ [29]:* An enhanced version of the heterogeneity-aware Gavel scheduler [29], modified to support RL post-training. Gavel+ optimizes resource allocation at the job level (calculating optimal GPU fractions) but lacks fine-grained control to interleave phase-level executions.

## 7.2 Micro-Benchmarks

To demonstrate how ROLLMUX effectively reclaims dependency bubbles (§4.3), we conduct three micro-benchmarks in

<sup>3</sup>For multi-turn workloads, Len refers to a per-turn output length.



(a) Temporal Mux (single-turn, Type-A x 2). (b) Train Mux (multi-turn, Type-D x 2 + E). (c) Spatial Mux (mixed, Type-C + D x 2).

Figure 10: Micro-benchmarking results. For each benchmark, the left panel is a gantt chart showing the co-execution timeline; the right panel quantifies the benefit, in which ROLLMUX achieves  $1.82 - 2.11\times$  higher cost-efficiency.

different multiplexing scenarios and measure cost efficiency (throughput per dollar) against the baselines.

**Temporal Multiplexing.** First, we evaluate co-executing two jobs with similar structures (Type-A) via temporal multiplexing, representing an ideal case where jobs are fully complementary. As shown in Figure 10a, ROLLMUX perfectly interleaves their execution, keeping both the rollout and training clusters fully utilized. Consequently, ROLLMUX improves cost-efficiency by 82%, 55.6%, and 46.8% over Solo-D, Gavel+, and veRL, respectively. The baselines fall short because Solo-D and Gavel+ leave one resource pool idle at all times, while the monolithic veRL underutilizes its expensive H800 compute power during memory-bound rollout phases.

**Handling Rollout-Heavy Jobs.** Next, we target rollout-heavy workloads by co-scheduling two Type-D jobs ( $T_D^{\text{roll}} \approx 2.5T_D^{\text{train}}$ ) and one Type-E job ( $T_E^{\text{roll}} \approx 6T_E^{\text{train}}$ ). In this scenario, ROLLMUX scales the rollout pool to 24 H20 GPUs, dedicating an inference node to each job’s rollout phase while time-multiplexing a single H800 training node for all training phases in a round-robin sequence (Figure 10b). This schedule achieves 104%, 61.9%, and 29.9% higher cost-efficiency than Solo-D, Gavel+, and veRL. Solo-D and Gavel+ perform poorly as the prolonged rollout phases force the expensive training nodes to sit idle for extended periods.

**Spatial Multiplexing.** Finally, we evaluate ROLLMUX’s ability to handle heterogeneity by co-scheduling one large Type-C job (requiring  $16 \times \text{H20} + 16 \times \text{H800}$ ) and two smaller Type-D jobs (each requiring  $8 \times \text{H20} + 16 \times \text{H800}$ ). As depicted in Figure 10c, ROLLMUX identifies the idle resources created by the large job’s rollout phase and strategically “packs” the two smaller jobs into these bubbles. This dynamic spatial packing maximizes aggregate utilization, delivering 111%, 85.1%, and 66.1% higher cost-efficiency compared to Solo-D, Gavel+, and veRL. This result highlights a critical advantage: unlike the baselines, ROLLMUX can dynamically consolidate diverse jobs to available capacity.

**Interference Overhead.** To quantify the cost of co-execution, we measure the throughput degradation caused by inter-job contention (Table 4). Because the inter-group scheduler proactively prunes placements that would violate residency or performance constraints, ROLLMUX incurs a minimal 5–9%

Micro-benchmark	Solo Disaggregation	Ideal	ROLLMUX
(a) Temporal Mux	1.00	1.07	0.98
(b) Train Mux	1.00	1.07	0.95
(c) Spatial Mux	1.00	1.11	0.91

Table 4: Normalized training throughput, showing ROLLMUX incurs less than a 10% overhead compared to isolated execution (baseline 1.0). For reference, ‘Ideal’ represents the performance ceiling from co-locating all phases on H800.

overhead compared to solo execution. Even compared to an idealized co-location upper bound where each job runs all phases exclusively on expensive H800 GPUs with zero network cost, the throughput gap remains a modest 9.0–20.0%, confirming that the heavy lifting of context switching and synchronization is effectively masked by our optimizations, and bad placements potentially causing contention are precluded.

### 7.3 Ablation Study

We next break down ROLLMUX’s performance to quantify the individual contributions of its key runtime optimizations.

**Long-Tail Migration.** First, we evaluate the effectiveness of request migration in neutralizing the stochasticity of RL rollout (§4.3). As illustrated in Figure 11-left, the generation length of rollout requests exhibits a pronounced heavy-tailed distribution across all model sizes and output lengths, where a small fraction of “straggler” requests persist long after the majority have finished. Figure 11-right demonstrates that enabling request migration effectively reclaims the capacity of “skewness bubbles.” By preemptively migrating the tail-bound requests to a small subset of GPUs, ROLLMUX allows the next job’s rollout phase to begin immediately on the freed majority of resources, improving the end-to-end throughput by  $1.06\times$  to  $1.28\times$ . Notably, the gains are most pronounced for workloads with longer output sequences (e.g., 14B-8k) where the straggler effect is amplified. The benefit is slightly more modest when pairing jobs with highly dissimilar characteristics (e.g., 7B-8k and 14B-8k), as the natural variance in their phase durations already mitigates some contention.

**Topology-Aware Model Sync.** Next, we evaluate the efficiency of ROLLMUX’s topology-aware model synchronization scheme in geo-disaggregated setups (§5.2). As shown in Figure 12, for a single-node update ( $8 \text{ H800s} \rightarrow 8 \text{ H20s}$ ),

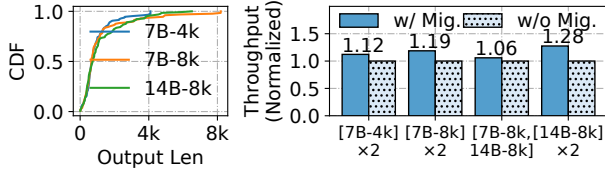


Figure 11: **Left**: the long-tail distribution of LLM generation length in the rollout phase. **Right**: effectiveness of request migration in mitigating long-tail rollouts.

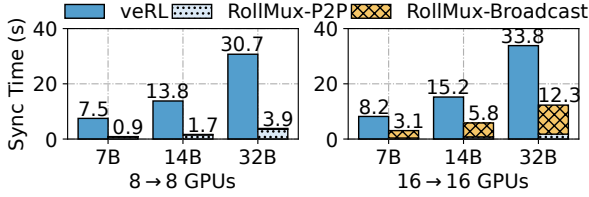


Figure 12: Model synchronization time. **Left**: single-node, from 8 H800 to 8 H20 GPUs. **Right**: multi-node, from 16 H800 to 16 H20 GPUs. ROLLMUX’s topology-aware model sync is up to 8.33 $\times$  faster compared to veRL [41].

ROLLMUX achieves 7.87 $\times$ –8.33 $\times$  speedup over veRL. This significant improvement stems from its hierarchical strategy: ROLLMUX transmits exactly one copy of the model parameters across the slow inter-cluster link and leverages the high-bandwidth local NVLink fabric for the final intra-cluster broadcast. In contrast, the baseline is bottlenecked by redundantly fetching independent copies for every rollout GPU. This advantage scales robustly: even in a multi-node setting (16 H800s  $\rightarrow$  16 H20s), ROLLMUX maintains 2.62 $\times$ –2.75 $\times$  speedup (Figure 12), confirming that our topology-aware protocol effectively masks the bandwidth limitations inherent to disaggregated clusters.

## 7.4 ROLLMUX at Scale

To evaluate ROLLMUX’s performance at production-scale, we replay a two-week trace from one of our cluster tenants. The trace comprises 200 highly heterogeneous RL post-training jobs using Qwen-family models, including both single- and multi-turn interaction patterns on diverse datasets. Among them, the model sizes range from 3B to 32B, the maximum response lengths range from 4k to 32k tokens (mean: 12.1k), and the mean job duration is 27.9 hours. We assign each job an SLO sampled uniformly from (1, 2) relative to its solo run-time. We compare ROLLMUX against the industry-standard solo disaggregation (Solo-D), with 1:1 rollout and training GPUs, and the monolithic co-located baseline (veRL).

In terms of cluster provisioning cost, ROLLMUX spends only \$510 per hour to accommodate all jobs, a 1.84 $\times$  and 1.38 $\times$  reduction compared to Solo-D and veRL, respectively, while meeting all job SLOs (Figure 13a). This cost efficiency directly stems from our scheduling algorithm (Algorithm 1)

that minimizes the marginal cost upon job arrivals.

In terms of resource efficiency, ROLLMUX reduces dependency bubbles by 24.4% on the rollout cluster and 43.1% on the training cluster compared to solo disaggregation. The improvement is more pronounced for training because the workloads are typically rollout-heavy, leaving more bubbles on the training GPUs (Figure 13b and Figure 13c). By tightly packing jobs, ROLLMUX requires a peak of only 152 H800 GPUs for training, a 2.16 $\times$  reduction from the 328 H800s needed by both veRL and solo disaggregation. For rollout, ROLLMUX’s peak usage is 216 H20 GPUs, a 1.52 $\times$  reduction compared to the 328 H20s required by solo disaggregation. Although the co-located veRL baseline uses no separate rollout GPUs, ROLLMUX is still 1.38 $\times$  more cost-effective overall. It achieves this by offloading memory-bandwidth-intensive rollout phases to cheaper H20 GPUs via disaggregation while filling the resulting dependency bubbles by co-scheduling.

## 7.5 Scheduler Performance

We finally evaluate the optimality and scalability of ROLLMUX’s inter-group scheduler (§4.2) via large-scale trace simulation.

**Experimental Setup.** We use job arrival patterns from a 300-job, 580-hour segment of the Microsoft Philly multi-tenant training cluster trace [24], the average job duration is 14.4 hours, and the longest is 142.9 hours. While the trace dictates arrival times and durations, we synthesize the job characteristics to model modern RL post-training workloads. As detailed in Table 6, we define three job profiles based on the ratio of rollout time ( $T_{\text{roll}}$ ) to training time ( $T_{\text{train}}$ ). (1) **Balanced (BL)**: Balanced  $T_{\text{roll}}$  and  $T_{\text{train}}$ , representative of single-turn workloads like RLHF [30] or RLVR [39]; (2) **Rollout-Heavy (RH)**:  $T_{\text{roll}} \gg T_{\text{train}}$ , modeling multi-turn workloads such as agentic reasoning [31]; and (3) **Train-Heavy (TH)**:  $T_{\text{train}} \gg T_{\text{roll}}$  for evaluation completeness, but is rare in real-world RL training.

For each profile, we generate jobs of three sizes (Small, Medium, Large), resulting in nine distinct job configurations. We test each profile individually and also use a **Mixed** workload, which contains a uniform mix of all nine configurations, to simulate a realistic production environment.

**Baselines.** We compare ROLLMUX against following baselines:

- *Offline Optimal (Opt)*. Assigns arriving jobs to *offline optimal* placements found via a brute-force search over all possible groupings and placements—including job re-ordering and re-grouping that are infeasible in online deployments—and serves as a theoretical upper bound.
- *Random*. Assigns arriving jobs to a random group (or a new one) that can accommodate it. The job is then placed on random rollout and train nodes within this group.
- *Greedy (Most-Idle)*. Assigns arriving jobs to the group with the highest idle-time percentage. Within that group, it places the job on the most idle rollout and train nodes.



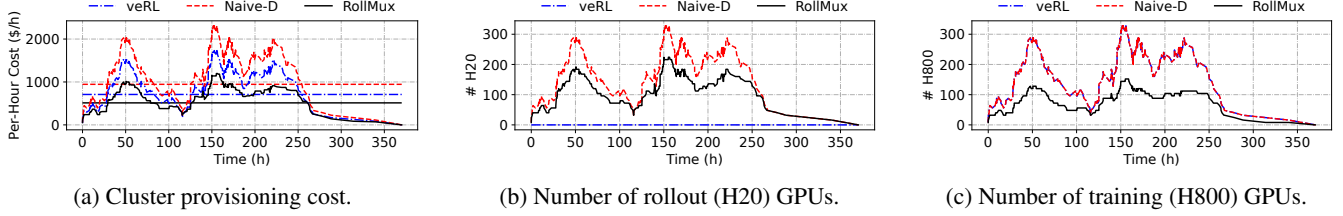


Figure 13: **[Testbed]** Cluster provisioning cost and GPU usage of ROLLMUX and baselines under real-world production workloads, where ROLLMUX reduces total provisioning cost by  $1.38\times$  and  $1.84\times$  compared to veRL and Solo-D, respectively.



Figure 14: **[Simulation]** Sensitivity analysis of ROLLMUX’s inter-group scheduler.

**Sensitivity Analysis.** To determine how sensitive the scheduler’s quality is to its key parameters, we vary a single parameter while all others are set to a default configuration<sup>4</sup>.

**Impact of Workload Characteristics.** Figure 14a shows that ROLLMUX’s near-optimal performance holds across all workload types. It consistently achieves 100% SLO attainment with a cost overhead of just  $1.01\times$ – $1.12\times$  relative to the optimal. In contrast, the baselines perform poorly across different workloads, with Greedy slightly outperforms the Random strategy. The Random strategy’s cost is  $1.72\times$ – $2.00\times$  optimal with only 37–58% SLO attainment, while the Greedy scheduler is slightly better but still inadequate, reaching  $1.38\times$ – $1.89\times$  optimal cost for 42–61% SLO attainment.

**Impact of Job SLOs.** We vary the job SLO requirements, testing both uniform SLOs (all jobs set to 1.2, 1.5, or 2.0) and heterogeneous, job-specific SLOs drawn from  $\text{Unif}(1, 2)$ . As shown in Figure 14b, ROLLMUX’s performance is highly stable against SLO tightness, always achieving 100% attainment with a consistent, near-optimal cost. Conversely, the baselines are highly sensitive to the SLO target and more expensive ( $1.59\times$ – $2.09\times$  optimal). As the SLO target loosens from 1.2 to 2.0, the SLO attainment for Random and Greedy improves from 38%/43% to 71%/73%, respectively. This shows that while looser SLOs make it easier for naive heuristics to succeed by chance, they still fail to provide guarantees.

**Impact of Group Residency.** Since node memory capacity directly limits the group size, we evaluate its impact by varying the maximum allowed group size from 2 to 5. Figure 14c shows that performance is relatively insensitive to the maximum group size for all methods. Across all configurations, ROLLMUX consistently maintains the lowest cost and 100% SLO attainment, while the baselines remain significantly de-

<sup>4</sup>Default configuration: mixed workload types, heterogeneous SLOs drawn from  $\text{Unif}(1, 2)$ , and a max group residency of 5.

Decision	Number of Concurrent Jobs						
Lat. (ms)	5	9	13	100	500	1000	2000
ROLLMUX	5.6	6.5	7.6	41.9	198	318	591
Opt.	113	>1min <sup>†</sup>	>5h <sup>†</sup>	—*	—*	—*	—*

<sup>†</sup> Represents latency exceeding 1 minute and 5 hours, respectively.

\* Not applicable; computation is intractable at this scale.

Table 5: Decision latency (ms) vs. number of concurrent jobs. ROLLMUX scales well; Brute-force Opt’s latency grows exponentially and quickly becomes impractical.

fectured (only 48%–61% SLO attainment) and more expensive ( $1.59\times$ – $2.15\times$  optimal). This suggests that even small group sizes (e.g., 2 or 3) provide sufficient packing flexibility for ROLLMUX to find efficient placements; larger groups do not bring improved performance or cost efficiency.

**Scheduler Scalability and Latency.** We evaluate ROLLMUX’s decision latency and scalability, with results presented in Table 5. ROLLMUX demonstrates near-linear scalability: its decision time is only 591 ms for 2,000 jobs, confirming that Algorithm 1 efficiently handles production-scale workloads. In stark contrast, the brute-force optimal solver exhibits exponential growth in latency, exceeding five hours for just 13 jobs—rendering it infeasible for any practical workload size.

## 8 Related Work

**Deep Learning Schedulers.** Extensive research has focused on scheduling for general-purpose deep learning clusters. These works aim to improve fairness [14, 26, 47], GPU sharing efficiency [46, 48], heterogeneity awareness [29, 40], and job goodput [33, 56]. However, these systems universally target the entire job as the atomic unit of scheduling and assume stable, predictable iteration times. ROLLMUX is the first phase-

centric co-scheduling system for stochastic RL post-training jobs.

**RL Post-Training Frameworks.** Recent systems have focused on optimizing the performance of a *single, individual* RL post-training job. While early frameworks used static resource partitioning [20, 25, 52], subsequent work improved utilization via co-location [41], multi-controller designs [45], long-tail mitigation [12, 59], and asynchronous algorithms [10, 16, 58]. Different from these single-job optimizations, ROLLMUX addresses the complementary part by taking a global, cluster-level perspective, orchestrating multiple concurrent jobs via co-scheduling.

**Disaggregated Systems.** ROLLMUX builds on the principle of disaggregation, a concept explored in OS kernels [38] and serving systems [19, 32, 57]. The most direct parallel is the prefill-decode disaggregation in LLM inference [19, 32, 57]. Within this domain, recent serving systems have also explored efficient resource management, but all in a single-job scope [5, 9, 18, 60]. In contrast, ROLLMUX introduces the first scheduling framework specifically designed for multi-tenant clusters, a scenario not addressed by prior works.

## 9 Conclusion

This paper presents ROLLMUX, a multi-tenant cluster scheduler tailored for rollout-training disaggregated RL post-training. ROLLMUX introduces a two-tier scheduling mechanism that near-optimally partitions RL jobs into co-execution groups and orchestrates their execution in a tightly-woven pattern. This approach effectively reduces dependency bubbles caused by disaggregation, thereby minimizing cluster provisioning costs. Extensive evaluation using real-world production traces on disaggregated clusters with up to 328 GPUs each demonstrates that ROLLMUX achieves up to  $1.84\times$  cost savings while maintaining 100% performance SLO attainment.

## References

- [1] Introducing openai o3 and o4-mini. <https://openai.com/index/introducing-o3-and-o4-mini/>, 2024.
- [2] AIME 2024 Dataset, 2025.
- [3] Qwq-32b: Embracing the power of reinforcement learning. <https://qwenlm.github.io/blog/qwq-32b/>, 2025.
- [4] Qiaoling Chen, Zijun Liu, Peng Sun, Shenggui Li, Guoteng Wang, Ziming Liu, Yonggang Wen, Siyuan Feng, and Tianwei Zhang. Respec: Towards optimizing speculative decoding in reinforcement learning systems. *arXiv preprint arXiv:2510.26475*, 2025.
- [5] Weihao Cui, Yukang Chen, Han Zhao, Ziyi Xu, Quan Chen, Xusheng Chen, Yangjie Zhou, Shixuan Sun, and Minyi Guo. Optimizing slo-oriented llm serving with pd-multiplexing, 2025.
- [6] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
- [7] Farama Foundation. Gymnasium - frozenlake environment. [https://gymnasium.farama.org/environments/toy\\_text/frozen\\_lake/](https://gymnasium.farama.org/environments/toy_text/frozen_lake/), 2024. Accessed: 2025-09.
- [8] Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjuan Zhong. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*, 2025.
- [9] Jingqi Feng, Yukai Huang, Rui Zhang, Sicheng Liang, Ming Yan, and Jie Wu. Windserve: Efficient phase-disaggregated llm serving with stream-based dynamic scheduling. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 1283–1295, 2025.
- [10] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, et al. Areal: A large-scale asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025.
- [11] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-latency serverless inference for large language models. In *USENIX OSDI*, pages 135–153, 2024.
- [12] Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, et al. Rollpacker: Mitigating long-tail rollouts for fast, synchronous rl post-training. *arXiv preprint arXiv:2509.21009*, 2025.
- [13] Michael R Garey and David S Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.
- [14] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.
- [15] Shixiang Gu, Tim Lillicrap, Richard E. Turner, Zoubin Ghahramani, Bernhard Schölkopf, and Sergey Levine.

- Interpolated policy gradient: Merging on-policy and off-policy gradient estimation for deep reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.
- [16] Zhenyu Han, Ansheng You, Haibo Wang, Kui Luo, Guang Yang, Wenqi Shi, Menglong Chen, Sicheng Zhang, Zeshun Lan, Chunshi Deng, et al. Asyncflow: An asynchronous streaming rl framework for efficient llm post-training. *arXiv preprint arXiv:2507.01663*, 2025.
  - [17] Jingkai He, Tianjian Li, Erhu Feng, Dong Du, Qian Liu, Tao Liu, Yubin Xia, and Haibo Chen. History rhymes: Accelerating llm reinforcement learning with rhymerrl. *arXiv preprint arXiv:2508.18588*, 2025.
  - [18] Ke Hong, Lufang Chen, Zhong Wang, Xiuhong Li, Quli Mao, Jianping Ma, Chao Xiong, Guanyu Wu, Buhe Han, Guohao Dai, et al. semi-pd: Towards efficient llm serving via phase-wise disaggregated computation and unified storage. *arXiv preprint arXiv:2504.19867*, 2025.
  - [19] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.
  - [20] Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Zilin Zhu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, et al. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. *arXiv preprint arXiv:2405.11143*, 2024.
  - [21] Jingcheng Hu, Yinmin Zhang, Qi Han, Daxin Jiang, Xiangyu Zhang, and Heung-Yeung Shum. Open-reasoner-zero: An open source approach to scaling up reinforcement learning on the base model, 2025.
  - [22] Qinghao Hu, Shang Yang, Junxian Guo, Xiaozhe Yao, Yujun Lin, Yuxian Gu, Han Cai, Chuang Gan, Ana Klimovic, and Song Han. Taming the long-tail: Efficient reasoning rl training with adaptive drafter. *arXiv preprint arXiv:2511.16665*, 2025.
  - [23] Anant Singh Jain and Sheik Meeran. Deterministic job-shop scheduling: Past, present and future. *European Journal of Operational Research*, 113(2):390–434, 1999.
  - [24] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.
  - [25] Oleksii Kuchaiev, Jason Li, Huyen Nguyen, Oleksii Hrinchuk, Ryan Leary, Boris Ginsburg, Samuel Krman, Stanislav Beliaev, Vitaly Lavrukhin, Jack Cook, et al. Nemo: a toolkit for building ai applications using neural modules. *arXiv preprint arXiv:1909.09577*, 2019.
  - [26] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient {GPU} cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.
  - [27] Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. Real: Efficient rlhf training of large language models with parameter reallocation. *arXiv preprint arXiv:2406.14088*, 2024.
  - [28] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elilbol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.
  - [29] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498, 2020.
  - [30] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.
  - [31] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym, 2024.
  - [32] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.
  - [33] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.



- [34] Ruoyu Qin, Weiran He, Weixiao Huang, Yangkun Zhang, Yikai Zhao, Bo Pang, Xinran Xu, Yingdi Shan, Yongwei Wu, and Mingxing Zhang. Seer: Online context learning for fast synchronous llm reinforcement learning. *arXiv preprint arXiv:2511.14617*, 2025.
- [35] Abhinav Rastogi, Albert Q Jiang, Andy Lo, Gabrielle Berrada, Guillaume Lample, Jason Rute, Joep Barmantlo, Karmesh Yadav, Kartik Khandelwal, Khyathi Raghavi Chandu, et al. Magistral. *arXiv preprint arXiv:2506.10910*, 2025.
- [36] Salvatore Sanfilippo. Redis - the real-time data platform, 2009. Accessed: 2025-09-08.
- [37] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [38] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiyang Zhang. {LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, 2018.
- [39] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.
- [40] Weihang Shen, Mingcong Han, Jialong Liu, Rong Chen, and Haibo Chen. {XSched}: Preemptive scheduling for diverse {XPU}s. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 671–692, 2025.
- [41] 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. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297, 2025.
- [42] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- [43] ROLL Team. Reinforcement Learning Optimization for Large-Scale Learning: An Efficient and User-Friendly Scaling Library. *arXiv preprint arXiv:2506.06122*, 2025.
- [44] Jinghui Wang, Shaojie Wang, Yinghan Cui, Xuxing Chen, Chao Wang, Xiaojiang Zhang, Minglei Zhang, Jiarong Zhang, Wenhao Zhuang, Yuchen Cao, et al. Seamlessflow: A trainer agent isolation rl framework achieving bubble-free pipelines via tag scheduling. *arXiv preprint arXiv:2508.11553*, 2025.
- [45] Zhixin Wang, Tianyi Zhou, Liming Liu, Ao Li, Jiarui Hu, Dian Yang, Yinhui Lu, Jinlong Hou, Siyuan Feng, Yuan Cheng, et al. Distflow: A fully distributed rl framework for scalable and efficient llm post-training. *arXiv preprint arXiv:2507.13833*, 2025.
- [46] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of fragmentation: Scheduling {GPU-Sharing} workloads with fragmentation gradient descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 995–1008, 2023.
- [47] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.
- [48] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020.
- [49] Ran Yan, Youhe Jiang, Tianyuan Wu, Jiaxuan Gao, Zhiyu Mei, Wei Fu, Haohui Mai, Wei Wang, Yi Wu, and Binhang Yuan. AReL-Hex: Accommodating asynchronous rl training over heterogeneous gpus. *arXiv preprint arXiv:2511.00796*, 2025.
- [50] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025.
- [51] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world

- web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022.
- [52] Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, et al. Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales. *arXiv preprint arXiv:2308.01320*, 2023.
- [53] Minchen Yu, Rui Yang, Chaobo Jia, Zhaoyuan Su, Sheng Yao, Tingfeng Lan, Yuchen Yang, Yue Cheng, Wei Wang, Ao Wang, et al.  $\lambda$ Scale: Enabling fast scaling for serverless large language model inference. *arXiv preprint arXiv:2502.09922*, 2025.
- [54] Qiyang Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gao-hong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.
- [55] Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. {BlitzScale}: Fast and live large model autoscaling with o(1) host caching. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 275–293, 2025.
- [56] Pengfei Zheng, Rui Pan, Tarannum Khan, Shivaram Venkataraman, and Aditya Akella. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 703–723, 2023.
- [57] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.
- [58] Yinmin Zhong, Zili Zhang, Xiaoniu Song, Hanpeng Hu, Chao Jin, Bingyang Wu, Nuo Chen, Yukun Chen, Yu Zhou, Changyi Wan, et al. Streamrl: Scalable, heterogeneous, and elastic rl for llms with disaggregated stream generation. *arXiv preprint arXiv:2504.15930*, 2025.
- [59] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, et al. Optimizing rlhf training for large language models with stage fusion. *arXiv preprint arXiv:2409.13221*, 2024.
- [60] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: Towards optimal large language model serving throughput, 2025.
- [61] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. Megascale-infer: Serving mixture-of-experts at scale with disaggregated expert parallelism, 2025.

## Appendix

**Proof of Meta-iteration Schedule’s Optimality.** To show the utilization-optimality of the meta-iteration schedule for non-overloaded groups, we first formally define the utilization of a group with meta-iteration time  $T_{\text{meta}}$ , where its rollout ( $U_R$ ) and training utilization ( $U_T$ ) are:

$$U_R = \frac{\sum_{j \in J_G} T_j^{\text{roll}}}{T_{\text{meta}}}, \quad \text{and} \quad U_T = \frac{\sum_{j \in J_G} T_j^{\text{train}}}{T_{\text{meta}}}.$$

Since the meta-iteration schedule repeats, it suffices to analyze a single meta-iteration where every job’s rollout and training phase executes once. Recall that  $T_G^{\text{cycle}} = \max_{j \in J_G} T_j^{\text{solo}}$  is the solo iteration time of the slowest job, and let  $j_1$  be the job attaining this maximum so that  $T_G^{\text{cycle}} = T_{j_1}^{\text{roll}} + T_{j_1}^{\text{train}}$ . For simplicity, the following proof considers only one rollout node and one training node, while the optimality generalizes to multi-node settings. By the definition of a non-overloaded group, we have

$$T_G^{\text{cycle}} = T_{j_1}^{\text{roll}} + T_{j_1}^{\text{train}} \geq \max \left( \sum_{j \in J_G} T_j^{\text{roll}}, \sum_{j \in J_G} T_j^{\text{train}} \right).$$

This directly implies that the total rollout work of all other jobs can fit within  $j_1$ ’s training phase, and vice-versa:

$$\sum_{j \in J_G \setminus j_1} T_j^{\text{roll}} \leq T_{j_1}^{\text{train}}, \quad \sum_{j \in J_G \setminus j_1} T_j^{\text{train}} \leq T_{j_1}^{\text{roll}}.$$

Consequently, job  $j_1$ ’s execution time decides the entire schedule’s cycle time, yielding the cycle time as  $T_{j_1}^{\text{solo}}$ .

Conversely, any schedule that repeats a job is sub-optimal for a non-overloaded group because it extends the meta-iteration’s duration but adds proportionally less work, leading to a net decrease in resource utilization. We show this by trying to add a repetition of any job  $k$  to the meta-iteration schedule. Repeating job  $k$  adds work to both pools and necessarily prolongs the cycle time by at least  $T_k^{\text{solo}}$  due to waiting for the longest  $j_1$ , causing the new utilization  $U'$  lower than the original  $U$ . We can bound the change in utilization,  $\Delta U = U' - U$ , as follows. The change for the rollout and the training pool,  $\Delta U_R$  and  $\Delta U_T$ , is bounded by:

$$\begin{aligned} \Delta U_R &= U'_R - U_R \leq \frac{\sum_{j \in J_G} T_j^{\text{roll}} + T_k^{\text{roll}}}{T_{j_1}^{\text{solo}} + T_k^{\text{roll}}} - \frac{\sum_{j \in J_G} T_j^{\text{roll}}}{T_{j_1}^{\text{solo}}}, \\ \Delta U_T &= U'_T - U_T \leq \frac{\sum_{j \in J_G} T_j^{\text{train}} + T_k^{\text{train}}}{T_{j_1}^{\text{solo}} + T_k^{\text{train}}} - \frac{\sum_{j \in J_G} T_j^{\text{train}}}{T_{j_1}^{\text{solo}}}. \end{aligned}$$

Therefore, the total change in utilization is

$$\begin{aligned} \Delta U &= \Delta U_R + \Delta U_T = U'_R - U_R + U'_T - U_T \\ &\leq \frac{T_k^{\text{train}} T_{j_1}^{\text{solo}} - T_k^{\text{solo}} \sum_{j \in J_G} T_j^{\text{train}} + T_{j_1}^{\text{solo}} T_k^{\text{roll}} - T_k^{\text{solo}} \sum_{j \in J_G} T_j^{\text{roll}}}{T_{j_1}^{\text{solo}} (T_{j_1}^{\text{solo}} + T_k^{\text{solo}})} \\ &= \frac{T_k^{\text{solo}} (T_{j_1}^{\text{solo}} - \sum_{j \in J_G} T_j^{\text{solo}})}{T_{j_1}^{\text{solo}} (T_{j_1}^{\text{solo}} + T_k^{\text{solo}})} \leq 0. \end{aligned}$$

This proves that any repetition degrades overall utilization, making the round-robin schedule utilization-optimal for non-overloaded groups.

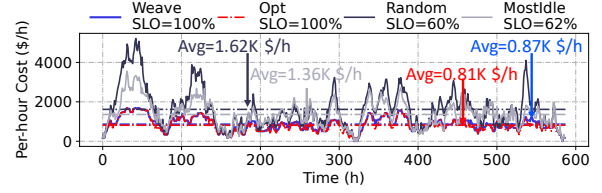


Figure 15: **[Simulation]** Cost-effectiveness and SLO attainment under a realistic mixed workload. (Workload: mixed, SLO~Unif(1, 2), max group size=5).

Table 6: Job configurations used in the simulation. Each job is defined by its rollout time ( $T_{\text{roll}}$ ) and train time ( $T_{\text{train}}$ ), which are drawn from different uniform distributions.

Workload	Size	Rollout Time ( $T_{\text{roll}}$ )	Train Time ( $T_{\text{train}}$ )
<b>Balanced (BL)</b>	Small	Unif(50, 100)	Unif(50, 100)
	Medium	Unif(100, 200)	Unif(100, 200)
	Large	Unif(200, 300)	Unif(200, 300)
<b>Rollout-Heavy (RH)</b>	Small	Unif(100, 200)	Unif(25, 50)
	Medium	Unif(200, 400)	Unif(50, 100)
	Large	Unif(400, 600)	Unif(100, 200)
<b>Train-Heavy (TH)</b>	Small	Unif(25, 50)	Unif(100, 200)
	Medium	Unif(50, 100)	Unif(200, 400)
	Large	Unif(100, 200)	Unif(400, 600)
<b>Mixed</b>	All	All	All

**Workloads for Simulation.** Table 6 details the job profiles used in § 7.5.

**[Simulation] End-to-end Performance.** This part reports the end-to-end performance of ROLLMUX’s scheduler under a realistic setting with Mixed workloads with heterogeneous job SLOs. As illustrated in Figure 15, ROLLMUX, achieves 100% SLO attainment by design, matching the theoretical Offline Optimal scheduler. In contrast, the heuristic-based baselines struggle significantly: the Random and Greedy (Most-Idle) schedulers meet the SLOs for only 60% and 62% of jobs, respectively. This is because ROLLMUX’s cost-based optimization (§ 4.3) explicitly prunes any placements that would violate a job’s SLO by assigning it an infinite cost.

In terms of resource efficiency, the average cost of ROLLMUX is at 0.87K \$/h, only 1.06× higher than the Offline Optimal. In contrast, the baselines are far more expensive. The Random and Greedy strategies incur average costs of 1.62K and 1.36K \$/h, respectively (1.97× and 1.66× of optimal). This high cost is driven by their sub-optimal group partitioning, where their costs spike to over 5K \$/h by scaling out to 1400 GPUs. ROLLMUX, however, manages load with a peak cost of only ~1.8K \$/h, requiring at most 504 GPUs, demonstrating its effectiveness at identifying SLO-aware, cost-efficient placements in a complex, online setting.