ROLL

ROSE: Rollout On Serving GPUs via Cooperative Elasticity for Agentic RL

Wei Gao[†*], Yuheng Zhao[†*], Dilxat Muhtar[‡], Dakai An[†], Xuchun Shang[‡], Tianyuan Wu[†], Lunxi Cao[†], Shaopan Xiong[‡], Weixun Wang[‡], Ju Huang[‡], Teng Ma[‡], Siran Yang[‡], Jiamang Wang[‡], Lin Qu[‡], Bo Zheng[‡], Wei Wang[†]

[†]*HKUST* [‡]*Alibaba Group*

[*]Equal contribution

Abstract

Agentic reinforcement learning (RL) is reshaping LLM post-training, but end-to-end training time is dominated by compute-intensive, multi-turn rollouts whose resource demand varies significantly across training steps. Resource-fixed systems cannot adapt to this variation, while resource-elastic approaches that provision external GPUs on demand suffer from high allocation overhead and limited availability.

We observe that serving clusters leave substantial GPU compute and memory idle, and propose *cooperative elasticity*: sharing already-deployed serving GPUs with rollout workloads to provide on-demand elastic capacity. Realizing this is non-trivial, as it must preserve serving SLOs under bursty traffic while minimizing cross-cluster communication overhead. We present ROSE, a system that realizes cooperative elasticity for agentic RL post-training, comprising three components: (1) an SLO-safe co-serving executor that co-locates heterogeneous serving and rollout models on the same GPUs, dynamically sharing memory and compute while preserving serving SLOs; (2) a cross-cluster weight transfer engine that leverages shard-aware routing and weight sparsity for fast synchronization; and (3) an elastic rollout scheduler that dynamically routes rollouts across dedicated and opportunistic serving GPUs. Experiments across multiple model sizes and cluster scales show that ROSE improves end-to-end throughput by 1.3–3.3× over resource-fixed baselines and reduces rollout time by 1.2–1.5× over resource-elastic baselines, with no serving SLO violations.

1 Introduction

The advent of agentic reinforcement learning (RL) is reshaping large language model (LLM) post-training, shifting models from passive response generation to actively interacting with complex, dynamic environments. Recent advances in tool use [20, 67], computer use [34, 36, 38], and software engineering [26, 60, 76] suggest that this training paradigm enables LLMs to progressively solve complex tasks [31, 47, 58].

The standard agentic RL workflow alternates between two stages: *rollout* and *training*. In the rollout stage, the agent LLM interacts with environments over multiple turns. At each turn, the agent *decodes* tokens that serve as action signals, and then runs *prefill* on the feedback returned by the environment. The resulting sequence of actions and feedback forms a trajectory. In the training stage, the agent LLM updates its weights on the collected trajectories and synchronizes them to the rollout stage for the next iteration.

In industry, LLM teams operate large training clusters for agentic RL and separate serving clusters for production deployment. Serving clusters are provisioned for peak traffic, but fluctuating demand often leaves GPUs underutilized [29, 30, 43, 65, 73, 81, 84]. Meanwhile, agentic RL training imposes substantial resource demands, straining GPU capacity within training clusters.

The training time of agentic RL is dominated by the rollout stage, which accounts for over 70% of total wall-clock time (Figure 1a), with pronounced long-tail latency across rollout batches (Figure 1b). Agentic rollouts further intensify GPU compute and memory demands. First, multi-turn interactions require processing growing conversation histories at each turn, and the compute-intensive prefill phase benefits from prefix caching and resource scaling (Figure 1c). Second, agentic RL typically adopts redundant sampling to yield high-quality samples for gradient stability [80, 86], which exacerbates per-step resource contention.

These characteristics make rollout throughput highly responsive to resource scaling. However, rollout resource demand varies significantly across training steps (Figure 1d). The *resource-fixed* systems [13, 16, 23, 35, 54, 68] that optimize within a *static GPU budget* cannot adapt to this resource demand variation. An allocation sized for peak demand idles GPUs during light-load steps, while one sized for average demand creates contention during heavy-load steps. This variation calls for resource elasticity. Existing *resource-elastic* systems typically provision *additional GPUs exclusively for rollouts* on demand using spot instances [69] or serverless

GPUs [41, 61]. However, these systems incur substantial allocation overhead during capacity churn (Figure 8b), and the training performance is bounded by GPU availability, which can be scarce under cluster-wide contention [68, 69].

A more natural source of elastic capacity for rollouts is the organization's operational serving cluster. Our measurements show that serving GPUs average only 18.9% compute utilization and 14.3% memory utilization (Figure 3b), leaving substantial idle capacity for rollouts. One intuitive approach is *bidirectional autoscaling*, shrinking the serving cluster under low load and redirecting freed GPUs exclusively to rollouts. However, reclaiming these GPUs for serving upon traffic bursts requires tens of seconds for model reloading and runtime initialization overhead (Figure 3c) that would violate serving Service Level Objectives (SLOs). Another option leverages *GPU multiplexing* to co-locate rollout and serving workloads on the same GPU. Yet existing GPU multiplexing systems fall short: those targeting homogeneous models [43] cannot share KV cache (KVC) across heterogeneous layouts, while those designed for comparable SLO workloads [7, 71, 81] either cannot retain in-GPU rollout prefix cache for short-lived reuse or yield rollout KVC and compute cycles promptly under serving bursts. The detailed analysis is in §3.2.

In this paper, we explore *cooperative elasticity*, where serving and rollout workloads cooperatively share *already-deployed* GPUs within an organization to maximize rollout throughput while preserving serving SLOs. We present ROSE, an agentic RL post-training framework to realize cooperative elasticity efficiently. While promising, repurposing serving GPUs for rollouts still poses two primary challenges: preserving serving SLOs under bursty traffic (**C1**) and heavy cross-cluster communication overhead (**C2**). First, colocating heterogeneous serving and rollout LLMs must preserve serving SLOs under bursty traffic, despite contention for both GPU memory and compute. To address this, we design a *co-serving executor* (§4.1) with three modules: (1) VMM-based cross-model KVC sharing for fast memory rebalancing across heterogeneous KVC layouts, (2) Preemptive memory sharing that exploits the short-lived nature of rollout interactions to retain its prefix cache in GPU and reclaim it aggressively during serving bursts, mitigating memory contention, and (3) Dual-SLO admission control leverages the priority asymmetry between rollouts, which can absorb second-level delays due to overlap among trajectories in the long tail, and serving, which requires millisecond-level SLOs, to prioritize serving through temporal sharing while opportunistically executing rollouts and avoiding severe compute interference.

Second, the serving and RL clusters may reside in different datacenters connected by bandwidth-limited links (10–200 Gbps Ethernet), so cross-cluster weight transfer can take tens of seconds to minutes (Figure 3d), eroding the speedups from elastic rollouts. Existing communication systems [10, 33, 49, 66] assume fixed process groups with uniform sharding. Cooperative elasticity breaks both assumptions: serving GPUs may join or leave across RL steps, while training and serving can adopt different parallelism strategies. To mitigate this communication overhead, we introduce a *cross-cluster weight transfer engine* (§4.2) that combines (1) A relay layer for asynchronous, fault-tolerant propagation, (2) Shard-aware routing that automatically maps sharding rules across heterogeneous parallelism configurations, and (3) Sparsity-aware compression that takes advantage of the lossless sparsity of RL weight deltas (>95%, see Figure 6) to transmit only non-zero elements, reducing transfer overhead to within 20 seconds even under 20 Gbps Ethernet (§6.3).

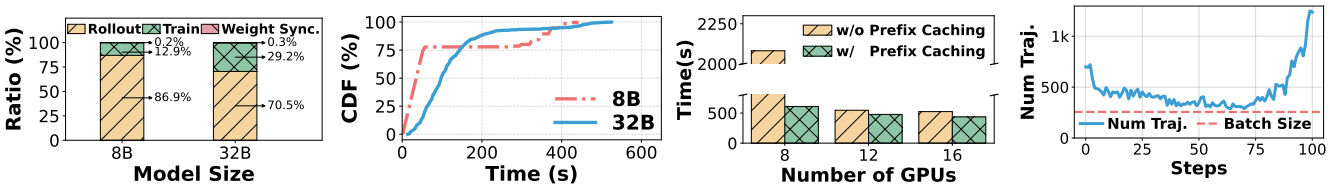
Beyond these challenges, to fully capitalize on cooperative elasticity under time-varying serving load, ROSE includes an *elastic rollout scheduler* that dispatches rollouts across dedicated rollout GPUs and opportunistic serving GPUs (§4.3). It combines (1) turn-wise concurrency-aware routing to offload excess trajectories when dedicated rollout GPUs saturate, and (2) cache-affinity placement to route each turn to the GPU holding its prefix KVC. It also provides fault-tolerant rerouting to migrate trajectories upon GPU stalls or failures.

We implement ROSE atop ROLL [63] and evaluate it with Qwen3-8B and Qwen3-32B [74] on agentic RL tasks [9, 57] using 16–48 training GPUs and 16–64 serving GPUs (H800). ROSE improves average end-to-end training throughput by 1.3–3.3× over resource-fixed baselines (ROLL [63], AReaL [13]), and reduces rollout time by 1.2–1.5× relative to resource-elastic baselines (RLBoost [69], λRL [41, 61]). These gains come with nearly zero allocation overhead and no serving P99 SLO violations, validating ROSE as a practical cooperative elasticity framework for agentic RL.

2 Background and Motivation

2.1 Agentic RL Training

An agentic RL training pipeline alternates between *rollout* and *training*. In modern agentic RL, rollout is often organized using group-based algorithms [11, 52, 80]. Taking GRPO as an example, the rollout stage launches B_0 *environment groups*, where environments within the same group share the same initial state. For each group, the agent LLM (actor) interacts with the environment over multiple turns: at each turn it observes the current *state*, samples an *action* from its policy, and sends it back to the environment; the environment then transitions to a new state and returns feedback. The agent generates G_0 sampled responses per group, producing G_0 trajectories from the same starting point. When each trajectory receives the terminal signal, a reward worker evaluates it and assigns a scalar reward. In the training stage, the agent LLM updates its model weights using the collected trajectories and reward signals, and the updated weights are synchronized back to the rollout workers for the next step.



(a) Performance Breakdown. **(b)** Long-tail Rollouts. **(c)** Impact of Prefill. **(d)** Need for Elasticity.

Figure 1. Characterization of agentic RL: (a) The breakdown of end-to-end training time; (b) The long-tail distribution of rollout execution time; (c) The impact of prefill on rollouts; (d) The demand for resource elasticity.

Figure 2. Illustration of Agentic RL Pipeline.

2.2 Workload Characterization

We perform workload characterization using Qwen3-8B and Qwen3-32B [74] within the agentic RL framework ROLL [63], configured with a maximum response length of 32k tokens, a batch size of 256, and a group size of 8 under the GRPO algorithm [52], on 16 H800 GPUs. We adopt synchronous RL training to profile the end-to-end training time. Specifically, we run the 8B and 32B LLMs on the agentic tasks FrozenLake and ALFWorld, respectively, for five consecutive steps.

Dominant Rollout Overhead. We report the performance breakdown in Figure 1a. The end-to-end training time consists of rollout, training, and weight synchronization overhead. The rollout stage accounts for over 70% of total time and dominates end-to-end training. This motivates the tailored rollout optimizations to improve training efficiency.

Long-tail Rollouts. Figure 1b presents the execution time distribution of trajectories during rollout for the 8B and 32B LLMs. We observe a pronounced long-tail pattern: most trajectories finish quickly, while a small fraction take much longer. In particular, the 75th percentile (P75) is at most 30% of the end-to-end rollout time. This observation is consistent with prior studies [16, 35, 64]. Such long-tail phenomena can cause GPU underutilization, as many GPUs remain idle while waiting for straggler trajectories to complete.

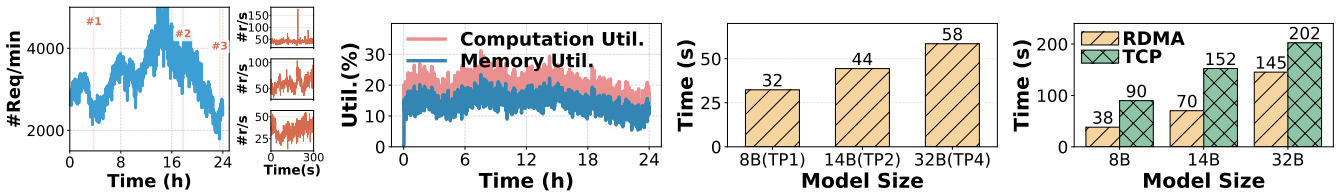
The Impact of Prefill. Multi-turn agentic RL entails frequent prefills [17, 64]. In our workloads, prefill tokens account for 77% (Qwen3-8B) and 86% (Qwen3-32B) of total tokens, making prefill a dominant cost. As a result, prefix caching, which reuses KV cache (KVC) for shared prefixes across requests, is widely used in multi-turn rollouts [12, 17, 35]. To quantify its impact, we vary the number of GPUs for Qwen3-8B and measure average rollout time with and without prefix caching in Figure 1c. Under resource contention, prefix caching improves rollout throughput by up to 3.4× (at 8 GPUs). As more GPUs are allocated, the rollout time reduces from 607 s to 435 s. This contrasts with single-turn RL, where scaling GPUs often yields limited benefit because long-tail samples are dominated by the bandwidth-bound decoding phase. In multi-turn agentic RL, scaling is more effective because of the compute-heavy prefill stage. Next, we analyze how rollout demand varies across steps to understand whether a static GPU allocation can be optimal.

Need for Resource Elasticity. In agentic RL training, agent–environment interaction often yields sparse rewards, and reward values within a group may exhibit low (or even zero) variance, weakening the learning signal. Hence, many algorithmic studies [80, 83, 86] propose redundant sampling: dynamically increasing B_0 to launch additional environment workers, and continuing rollouts until collecting B_0 groups with non-zero reward variance. We observe that many agentic RL jobs enable this feature in our internal cluster. Figure 1d shows that when training a Qwen3-8B model with DAPO [80], the number of generated trajectories vary significantly across steps, with the maximum reaching 5.7× the user specified batch size. With a fixed GPU budget, redundant sampling increases memory pressure, and the high variability in rollout volume makes a static, resource-fixed configuration inefficient. Beyond redundant sampling, even standard GRPO with a fixed batch size benefits from resource elasticity: long-tail rollouts and compute-heavy prefills create variable per-step resource demand, and our end-to-end evaluation (§6.1) confirms that resource elasticity improves average throughput by 1.31–1.46× under GRPO. This motivates a *resource-elastic* system that adjusts rollout GPUs over time to reduce contention and shorten step time.

2.3 Limitations of Existing Solutions

Resource-fixed RL systems struggle to balance utilization and training time. Many existing RL training systems [4, 12, 13, 16, 19, 23, 35, 51, 53, 63, 68, 70, 88, 89] provision a fixed number of GPUs for rollouts throughout training. This static allocation cannot align the actual rollout workload demand: rollout latency exhibits heavy tails, so overprovisioning leaves GPUs underutilized due to long-tail trajectories, while underprovisioning increases contention for compute and memory, inflating rollout time. Moreover, as Figure 1d shows, rollout demand varies across steps, making any single fixed allocation suboptimal over time.

Elastic RL systems suffer from high allocation overhead. Existing resource-elastic designs rely on preemptive GPUs (e.g., spot instances [69], serverless GPUs [41, 61]) to



(a) Serving Traffic. **(b)** GPU Utilization. **(c)** Allocation Overhead. **(d)** Comm Betw. Train&Serve.

Figure 3. Characterization of serving clusters and workloads: (a) Fluctuating serving traffic; (b) Serving GPU underutilization; (c) High allocation overhead; (d) Substantial communication overhead.

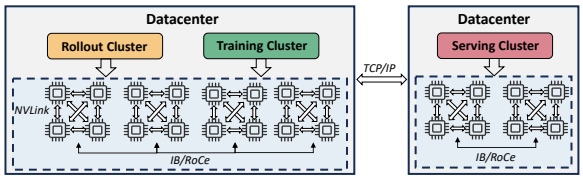


Figure 4. Scheme of Datacenter Infrastructure.

scale rollout capacity on demand. Because these GPUs lie outside the steady-state deployment, each provisioning event requires model loading and runtime initialization, which can take tens of seconds (Figure 3c). Spot preemption and serverless lease expiration further trigger repeated teardown-and-reinitialize cycles, turning allocation overhead into a persistent throughput tax that can consume over 20% of total training time (Figure 8). Cluster-wide resource contention exacerbates this problem: as spot capacity becomes scarce [69] or serverless pools fail to secure GPUs under concurrent RL jobs, the system falls back to fewer GPUs with more frequent reallocation attempts, further inflating allocation overhead and undermining the promised elasticity.

3 Opportunities and Challenges

We first describe the infrastructure setting, then quantify the harvestable serving capacity, explain why strawman approaches fail to exploit it, and highlight the key challenges.

3.1 Infrastructure for RL and Serving

Many AI organizations operate separate RL and serving infrastructures (Figure 4). We categorize GPUs into three clusters: a *training cluster*, a *rollout cluster*, and a *serving cluster*. Training and rollout clusters are co-located and communicate via high-speed interconnects (e.g., NVLink, InfiniBand). Serving clusters may reside in separate datacenters for operational isolation [25, 27, 71], connected by bandwidth-limited links (10–200 Gbps Ethernet) [17, 68]. RL infrastructure and LLM serving infrastructure are operated by different teams within the same organization, each managing large-scale GPUs. Since rollout and serving are both inference workloads, both teams often collaborate on optimizing a shared LLM inference engine, a pattern also observed in open-source communities [50]. This organizational proximity makes it natural to consider repurposing serving resources for rollouts when rollout demand spikes.

3.2 Harvestable Serving Capacity

We next quantify how much serving capacity is empirically available for rollouts and explain why strawman approaches fail to exploit it.

Fluctuating Serving Traffic Leaves GPU Underutilized. Production LLM serving workloads exhibit fluctuating request rates [46, 62, 65, 73, 81]. Figure 3a plots a 24-hour Microsoft trace [59] at minute granularity alongside three zoomed-in 5-minute windows at per-second granularity. At the minute level, the peak rate reaches 1.7× the 24-hour average. At the second level, burstiness is far more pronounced: per-second peaks reach 4.22×, 1.58×, and 1.73× their respective window averages, consistent with second-level spikes reported by BurstGPT [65]. To absorb such spikes, providers often statically overprovision for peak demand [87], resulting in substantial GPU underutilization. We quantify this by replaying a 24-hour production trace from [46] (preserving original prompt lengths, response lengths, and arrival process) on Qwen3-8B with 8 H800 GPUs. Figure 3b shows the GPU utilization sampled at 1-second intervals and smoothed with a one-minute moving average. On average, GPUs reach only 18.9% SM utilization and 14.3% HBM utilization, confirming that peak-provisioned GPUs remain significantly underutilized for much of the day. Beyond per-GPU underutilization, serving clusters often comprise thousands of GPUs [71], so even modest slack per GPU aggregates into a large pool of idle cycles and memory. A natural question is whether this spare capacity can be harvested for rollouts.

Strawman Approaches Cannot Safely Harvest the Capacity. A natural first attempt is *bidirectional autoscaling* [79, 82], which shrinks the serving cluster during low load and redirects freed GPUs to rollouts. However, bidirectional autoscaling is fundamentally limited: reclaiming GPUs from rollouts back to serving requires evicting in-flight rollouts and reloading models, taking tens of seconds (Figure 3c) and far exceeding typical SLO budgets. Because serving traffic is bursty at second-level granularity, frequent mode switching triggers repeated initialization overhead that erodes throughput gains. An alternative is *GPU multiplexing* [7, 43, 71, 81], which co-locates both workloads on the same GPU to avoid repeated initialization. However, co-locating heterogeneous models with asymmetric SLO requirements introduces challenges in memory sharing and compute interference that

existing systems do not address (detailed in §3.3). Our evaluation (§6.1) confirms that neither approach can safely harvest serving slack: both autoscaling [14] and multiplexing [81] violate serving SLOs and degrade rollout throughput.

These failures motivate a different approach. Rather than GPU-level switching or treating rollouts as generic co-located inference, we explore *cooperative elasticity*: keeping both serving and rollout models resident on each GPU and dynamically sharing resources, exploiting properties of agentic RL rollouts that make such sharing safe and efficient. While this can exploit abundant serving capacity with minimal allocation overhead, it raises two key challenges.

3.3 Challenges in Cooperative Elasticity

C1: Preserving Serving SLOs under Bursty Traffic. Cooperative elasticity keeps heterogeneous serving and rollout models deployed on the same GPUs, but preserving serving SLOs under bursty traffic requires accounting for both memory and compute contention in this deployment.

- **Inefficient memory sharing.** Serving and rollout models are often heterogeneous with incompatible KVC layouts. Static reservation of per-model KVC pools [28, 43, 50] requires runtime reinitialization, which can take tens of seconds (Figure 3c). Heterogeneous colocation engines [71, 81] typically omit in-GPU prefix caching or do not account for KVC contention between prefix cache entries and active request KVC across co-located models. Since rollouts benefit significantly from prefix caching, this contention competes with serving KVC for limited HBM and inflates latency under traffic spikes, triggering SLO violations.
- **Severe compute interference.** LLM serving systems enforce latency SLOs on *time-to-first-token* (TTFT) and *time-per-output-token* (TPOT). Several systems disaggregate prefill and decoding onto separate GPU pools (PD disaggregation) to stabilize TTFT/TPOT under bursty traffic [42, 87]. Unlike standard multi-tenant serving, where co-located workloads share comparable SLO requirements, rollout inference has fundamentally asymmetric characteristics: rollouts run a *different* LLM with much looser latency targets, generate long multi-turn sequences that sustain GPU occupancy for seconds, and perform both prefill and decode on the same GPU (PD co-location) even when serving deployment uses PD disaggregation. Standard request scheduling cannot bound the compute interference: a single rollout prefill chunk can delay serving decodes, and sustained rollout batches can starve serving prefills (analyzed in §6.2).

C2: Heavy Cross-cluster Communication Overhead. In many deployments, RL training and online serving are provisioned as separate GPU clusters, and weight updates must traverse cross-cluster links via a flexible transfer engine. To quantify this overhead, we measure the end-to-end time to transfer LLM parameters of varying sizes from a GPU node in the RL cluster to a GPU node in the serving cluster using `Mooncake Store` [45][1], over TCP (200 Gbps Ethernet) and RDMA (400 Gbps InfiniBand), shown in Figure 3d. Even with InfiniBand (which is uncommon across datacenters), it can take up to 145 s and grow quickly with model size, becoming a bottleneck for frequent weight synchronization.

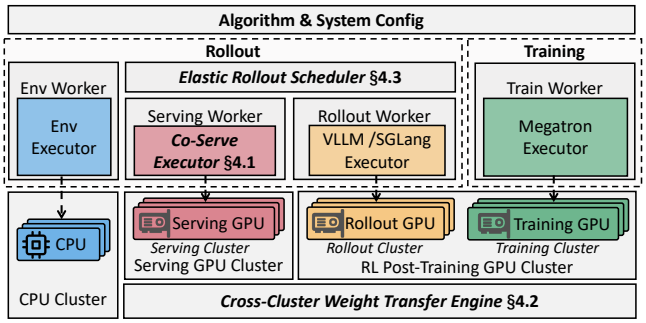


Figure 5. System Architecture of ROSE.

4 System Design

System Overview. To address the above challenges, we introduce ROSE, the architecture of which is illustrated in Figure 5. It includes three novel components: (1) an SLO-safe *co-serving executor* that co-locates *heterogeneous* serving and rollout LLMs on the same GPUs, dynamically sharing memory and compute while preserving serving SLOs (§4.1); (2) a cross-cluster *weight transfer engine* that asynchronously synchronizes model weights from the training cluster to the serving cluster with sharding and sparsity awareness (§4.2); and (3) an elastic *rollout scheduler* that dispatches rollouts across rollout and serving GPUs to realize the cooperative elasticity and achieve end-to-end training throughput improvement under varying serving and rollout load (§4.3).

Serving Cluster Setup. ROSE supports diverse serving deployments, including PD disaggregation vs. co-location and autoscaled vs. statically provisioned clusters. In our evaluation, we use PD disaggregation with a statically provisioned serving cluster and enforce P99 tail-latency SLOs on TTFT and TPOT. To avoid the initialization overhead of loading rollout models on demand, we pre-deploy commonly used rollout models alongside the heterogeneous serving LLMs in the serving cluster and expose endpoints for weight transfer and response generation. Rollout workers remain deactivated when idle and are activated during the rollout stage. We detail the concrete model pairing and cluster setup in §6.

System Workflow. The user specifies the RL resource request N_{rl} and an upper bound on the number of serving GPUs that can be borrowed N_{serving}. Upon job submission, the RL cluster reserves N_{rl} GPUs, and the serving cluster selects up to N_{serving} GPUs with the lowest recent KVC usage over a fixed window (e.g., 1 hour). To avoid contention

[1]The weight transfer adopted here is a *batch* baseline in §6.3.

among concurrent RL jobs, ROSE assigns each selected serving GPU to at most one RL job for rollouts. During this setup, ROSE initializes the RL runtime on the reserved RL GPUs and launches rollout workers accordingly. On selected serving GPUs, ROSE activates the rollout runtime into GPU memory.

Each RL step follows a typical RL pipeline. First, ROSE initiates the rollout stage. Each trajectory involves multi-turn environment interactions, and the elastic rollout scheduler dispatches each turn to either dedicated rollout GPUs or borrowed serving GPUs to improve rollout throughput while enforcing serving SLOs. Second, on each serving GPU, the co-serving executor dynamically shares memory and compute between the heterogeneous serving and rollout LLMs, enforcing serving TTFT and TPOT SLOs while harvesting otherwise idle GPU cycles for rollouts. If serving load causes rollout stalls, the scheduler reroutes subsequent trajectories to underutilized GPUs. Third, the generated trajectories are fed into the training stage. Once training produces updated weights, ROSE invokes the cross-cluster weight transfer engine, which asynchronously pushes weight updates to serving GPUs while the next step's intra-cluster synchronization and rollout execution proceed in parallel (§4.2).

4.1 SLO-Safe Co-Serving Executor

The co-serving executor runs rollouts opportunistically on serving GPUs while preserving serving TTFT and TPOT SLOs. This requires managing two sources of contention: *GPU memory* (dominated by KVC) and *GPU compute* (which directly affects token-level latency). As discussed in §3.3, existing multiplexing systems cannot simultaneously meet these requirements. We address these challenges through three techniques, enforcing two principles: *serving-first memory* (serving always has priority on KVC capacity) and *serving-first compute* (serving always has priority on GPU compute).

VMM-based Cross-Model KV Cache Memory Sharing. Heterogeneous serving and rollout models have incompatible KVC layouts that preclude dynamic memory rebalancing in mainstream engines (§3.3). We leverage CUDA Virtual Memory Management (VMM) to enable fast, flexible KVC rebalancing across heterogeneous models. We decouple *virtual* KV address spaces from *physical* GPU pages: each model reserves a contiguous virtual KV address space that preserves its attention-kernel indexing, while all models share a global physical page allocator that maps and unmaps pages on demand. When serving load increases, we unmap physical pages from rollout's virtual address space and remap them into serving's virtual address space at page granularity (typically 2MB). This enables cross-model memory rebalancing without modifying KVC layouts. We keep a lightweight runtime context warm on serving GPUs for fast rollout model (re-)activation. Activating Qwen3-32B completes within 5 s via PCIe/NVLink weight loading, avoiding the tens-of-seconds overhead of add-capacity elasticity.

Preemptive Memory Sharing Policy. Prefix caching is critical for rollout performance (Figure 1c), but existing multiplexing systems [71, 81] either offload completed request KVC to CPU memory or do not manage prefix cache contention across co-located models. However, rollout environment interactions are short-lived: consecutive interactions typically occur within 10 s [17], concentrating prefix reuse in a brief window. CPU-GPU transfer latency makes offloading ineffective for such short intervals. We instead keep rollout prefix cache resident in GPU to exploit short-term reuse, but this creates memory contention with serving that prior systems do not address. We note that prior KVC engines [6, 45] also use host memory offloading, which is ineffective for rollouts because weights are updated frequently, invalidating cached entries while increasing high CPU memory pressure.

We address this tension through *preemptive memory sharing*: we keep rollout prefix cache in GPU under typical load, but aggressively reclaim it during serving bursts, relying on the short interaction window to recapture most reuse before entries expire. At the beginning of each RL step, the elastic rollout scheduler derives a per-GPU rollout KVC budget from the serving model's recent memory usage and reserves a fixed KVC headroom H (e.g., 20% of total GPU memory) for serving. This iteration-level budget stabilizes rollout memory usage under typical serving load, but it cannot react to sudden serving bursts.

The memory sharing policy proceeds in three steps. (1) *Burst trigger:* The co-serving executor continuously monitors serving KVC usage. When serving starts to consume the reserved headroom (i.e., serving KVC usage crosses a high-watermark within H), the executor enters a *pressure* state. (2) *Emergency cut:* In the pressure state, the executor immediately shrinks the rollout KVC budget by a fixed factor (2×) and returns the freed physical pages to the shared allocator. It reclaims rollout KVC pages at request granularity, aborts the affected rollout requests, and notifies the rollout scheduler (§4.3) to reroute the affected trajectories to other underutilized GPUs. This one-time aggressive cut avoids repeated fine-grained reallocations and reduces allocation/reclamation churn during load bursts. (3) *Freeze:* To prevent oscillation between reclaiming and regrowing rollout KVC under bursty traffic, the executor does not increase the rollout budget until the next RL step, when the rollout scheduler recomputes budgets using updated serving statistics. This conservative choice preserves serving SLOs, while the rollout scheduler absorbs the transient capacity loss by shifting subsequent rollout steps to underutilized GPUs, including dedicated rollout GPUs, as rollouts progress. We attach a short lease (e.g., 10 s) to each rollout KVC page and reclaim pages when the lease expires, bounding HBM consumption while capturing most prefix reuse within the typical environment interaction window.

Dual-SLO Admission Controller. Co-serving causes compute interference between rollout and serving under spatial

co-location (analyzed in §6.2). Unlike serving requests that require millisecond-level TTFT (e.g., 100 ms) and TPOT (e.g., 50 ms) SLOs, individual rollout stages exhibit pronounced long-tail latency, with P75 around 55–145s and tails exceeding 400s (Figure 1b). This long-tail overlap, together with trajectory rerouting by the elastic rollout scheduler (§4.3), allows ROSE to absorb second-level rollout delays without materially hurting the performance. We exploit this asymmetry through *temporal sharing* with serving-first admission control: only one workload executes on the GPU at a time while both reside in GPU memory, serving tokens are always prioritized, and rollout tokens are admitted only when sufficient SLO slack exists. When serving bursts arrive, we immediately yield compute—stalled rollouts simply migrate to underutilized GPUs without permanent progress loss.

We pre-profile runtime costs for both serving and rollout execution and use them to make online admission decisions. For prefill, we profile the latency of monolithic and chunked prefill as a function of prompt length, denoted by $\hat{T}_{\text{prf}}(L, m)$ where $m \in \{\text{mono}, \text{chunk}\}$. For decode, we profile the per-step latency as a function of batch size, denoted by $\hat{T}_{\text{dec}}(b)$. These profiles allow the executor to estimate, at each scheduling tick, the remaining slack under the serving TTFT and TPOT SLOs, and to admit rollout tokens only when sufficient slack is available. Although the serving stack uses PD disaggregation, we perform PD colocation for rollouts to maximize serving resource utilization. We further enable chunked prefill for rollouts (e.g., chunk size 512 tokens), which bounds the runtime of each rollout step and avoids head-of-line blocking for serving. Next, we describe the computation of SLO slack on prefiller and decoder instances.

On **prefiller** instances, we first compute the TTFT slack of pending serving requests at each scheduling tick. For a serving request r, let t_{now} be the current time and t_r^{arr} be its arrival time. Let B_{TTFT} be the configured TTFT SLO budget. Given prompt length L_r and the serving prefill mode m, the remaining TTFT slack is

$$S_r^{\text{prf}} = \left(t_r^{\text{arr}} + B_{\text{TTFT}}\right) - t_{\text{now}} - \hat{T}_{\text{prf}}(L_r, m). \tag{1}$$

We conservatively use the minimum slack among queued prefills as the TTFT slack for rollouts, $S_{\min}^{\text{prf}} = \min_{r \in Q_{\text{prf}}} S_r^{\text{prf}}$.

On **decoder** instances, for a serving request r, let t_r^{last} be the time it produced its most recent token. Let B_{TPOT} be the configured TPOT budget, and let b be the current serving decode batch size. The remaining TPOT slack is

$$S_r^{\text{dec}} = \left(t_r^{\text{last}} + B_{\text{TPOT}}\right) - t_{\text{now}} - \hat{T}_{\text{dec}}(b). \tag{2}$$

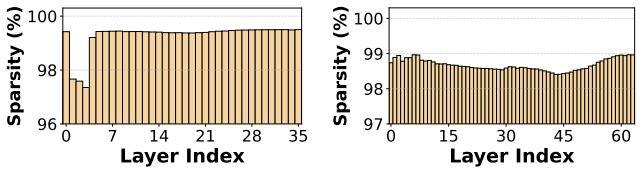
We again use the minimum slack among active decodes, $S_{\min}^{\text{dec}} = \min_{r \in Q_{\text{dec}}} S_r^{\text{dec}}$. We admit rollout token generation only when two conditions hold. First, the serving workload has sufficient TTFT and TPOT slack to accommodate the additional compute for rollout tokens. Second, allocating the corresponding KVC pages for rollout does not reduce the serving model's available KVC capacity below a reserved headroom. If rollout prefill or decoding makes no progress for a fixed timeout (e.g., 2 seconds), the co-serving executor reports a stall to the global scheduler and drops this trajectory, allowing the scheduler to reroute it elsewhere.

4.2 Cross-Cluster Weight Transfer Engine

Cross-cluster communication overhead can bottleneck cooperative elasticity (§3). Existing communication systems [10, 33, 49, 66] optimize conventional model training under low-bandwidth links by adjusting training configurations or exploiting gradient sparsity and quantization. However, they assume collective communication over fixed process groups with uniform sharding, typically under data parallelism. Cooperative elasticity introduces two requirements absent in these settings: (1) dynamic GPU membership, as serving GPUs join or leave across RL steps, precludes fixed collectives; and (2) heterogeneous parallelism strategies between training and serving require automatic shard mapping. Additionally, we observe that RL post-training *weight deltas* are >95% sparse (Figure 6), a property uncommon in conventional LLM training that enables lossless compressed delta transfer for *cross*-cluster weight synchronization. We address these through three techniques.

Asynchronous Weight Transfer. Resource elasticity makes the set of participating serving GPUs *dynamic*: serving GPUs may join or leave between RL steps due to changing serving load. This precludes fixed collective groups and requires fault-tolerant, point-to-point transfer that gracefully handles membership changes. Following RollArt [17], we build the transfer engine on `Mooncake Store` as a relay layer that decouples training and serving: training workers push weights in fixed-size buckets (e.g., 64 MB) to relay workers asynchronously, and serving workers pull in larger batches (e.g., 1 GB) on demand without coordinating with training or other serving workers. This avoids establishing fixed communication groups and makes transfer robust to membership changes. We overlap cross-cluster transfer with NCCL-based intra-cluster synchronization so that rollout workers can resume without waiting for cross-cluster transfer to finish.

Shard-aware Weight Transfer. Training and serving clusters adopt heterogeneous parallelism strategies (e.g., training with TP8×PP2 and serving with TP4), requiring automatic shard mapping across configurations. Naive approaches require manual resharding or full model aggregation before transfer. ROSE automatically infers each parameter's sharding rule by identifying the sharded dimension from the module type and parameter shape, computing per-rank slice ranges, and encoding this metadata in the Mooncake object key. On the training side, each device pushes its local shard asynchronously rather than first all-gathering the full model. To avoid redundant sends, each data-parallel rank transmits a mutually exclusive set of shards, parallelizing transfers and improving link utilization. On the serving side, each



(a) FrozenLake-8B. **(b)** ALFWorld-32B.

Figure 6. Layer-wise sparsity ratio at 10th step.

rank derives which buckets to pull based on the encoded metadata, fetching only its needed shards rather than a full model replica. This substantially reduces transfer volume and makes transfer transparent to the RL algorithm. We support tensor parallelism (TP) and pipeline parallelism (PP).

Sparsity-aware Weight Transfer. Transfer time scales with data volume, so naively synchronizing full weights becomes prohibitive as model size grows. Lossless gradient sparsity is rare in LLM training [40, 48, 75], but we observe that RL post-training produces a natural source of lossless sparsity: the *weight differentials* $\Delta W_t = W_t - W_{t-1}$ between consecutive steps are highly sparse, because RL algorithms employ gradient-stabilization techniques (e.g., reference models, KL penalties, and conservative update rules) [52, 80] that constrain policy drift. We define *sparsity ratio* as the fraction of zero elements in ΔW_t. Figure 6 reports the layer-wise sparsity for Qwen3-8B and Qwen3-32B at the 10^{th} RL step. Across layers, more than 95% of elements in ΔW_t are zero. §6.3 confirms that this high sparsity persists across RL steps. The engine can therefore ship sparse deltas rather than full replicas.

To exploit this sparsity efficiently, we compress ΔW_t in COO (coordinate) format for transfer and keep the previous-step weights W_{t-1} resident on local devices. At each update, we reconstruct the current weights by applying ΔW_t to W_{t-1}. This additive update introduces at most ∼1 s of compute overhead, which is small relative to the cross-cluster transfer time (up to minutes). To further reduce overhead, we shard the COO tensors according to the chosen parallelism strategy so that each device applies only its local shard, avoiding sparse-to-dense materialization and redundant additions.

4.3 Elastic Rollout Scheduler

The rollout scheduler harnesses cooperative elasticity to orchestrate trajectory generation across rollout and serving GPUs. However, rollout workers on serving GPUs expose a response generation endpoint, and their data plane differs from that of rollout workers running on dedicated rollout GPUs. We therefore introduce a unified rollout proxy to allow the scheduler to manage heterogeneous rollout workers sharing a common interface for response generation and per-trajectory metric collection. Next, we describe how the scheduler decides how many trajectories to offload to serving GPUs and how it places each trajectory.

Turn-wise Concurrency-aware Routing. The GPU compute and memory jointly bound the number of trajectories that can execute efficiently on the dedicated rollout GPUs. We perform offline profile and cap rollout concurrency at a workload-dependent threshold (e.g., 16 per GPU for Qwen3 models with 32K context length) to avoid KVC pressure and excessive scheduling overhead. When the instantaneous rollout demand exceeds this limit, the scheduler offloads the excess trajectories to available serving GPUs. Since multi-turn agentic RL alternates environment interaction with LLM generation, ROSE schedules rollouts at *turn* granularity rather than pinning each trajectory to a single GPU. This fine-grained control allows subsequent turns of a trajectory to spill over to serving GPUs under contention, and to migrate back to rollout GPUs once capacity becomes available.

Cache-Affinity Placement. To maximize the benefit of prefix caching, the rollout scheduler employs a cache-affinity placement policy across both dedicated rollout GPUs and opportunistic serving GPUs. For each trajectory, the scheduler records the rollout worker that served its previous turn, which typically retains the trajectory's prefix KVC. For each new turn, the scheduler first routes the request to the cache-affine worker if it has available capacity on a rollout GPU or, for a serving GPU, if admitting the request would not violate serving SLOs. If the cache-affine worker is unavailable, the scheduler falls back to a load-aware policy by dispatching the request to the least-loaded rollout GPU when possible; otherwise, it dispatches to the least-loaded eligible serving GPU. If neither pool has capacity, the request is queued until resources become available.

Fault Tolerance and Recovery. The rollout scheduler uses a heartbeat mechanism to monitor the liveness of each serving worker. When it receives an execution stall signal from the co-serving executor (§4.1) or detects a failure via health checks, the scheduler promptly reroutes the affected trajectories to other available GPUs, enabling fast recovery.

Extensions to Other Deployment Settings. ROSE can generalize to two other LLM serving deployments. *PD colocation.* When prefill and decode are co-located on the same serving instance, the executor derives per-tick compute slack for rollout admission as the minimum one between TTFT and TPOT slack. *Autoscaling.* This can leave a fraction of GPUs idle under low traffic. ROSE can run rollouts on these idle GPUs and leverage the fast model swap mechanism in §4.1 to utilize GPU compute and memory without contending with serving traffic. Due to the cost of large-scale evaluation across multiple production configurations, we focus on the mainstream PD-disaggregated deployment used in industry.

5 Implementation

We implement ∼5k lines of Python atop agentic training framework ROLL [63] and serving framework vLLM [28].

Agentic RL training. The rollout scheduler and the push side of the weight transfer engine are implemented inside ROLL. ROSE uses Megatron-LM [56] for training, vLLM for

rollout/serving with request migration, and ROLL's native environment runtime to manage environments.

Serving engine. The pull side of the transfer engine and the co-serving executor are built atop vLLM 0.10.0 [28]. We deploy rollout models in the serving cluster using `vllm serve`, exposing endpoints for weight transfer and response generation. When idle, rollout workers remain deactivated (not GPU-resident) and consume at most 2 GB GPU memory; they are activated during the rollout stage. We implement a Ray-based [39] load-balancing policy to route serving requests.

Relay worker. We use Mooncake v0.3.8 in the relay worker, allowing ROLL to publish updated weights and vLLM to pull them. We extend Mooncake with shard awareness and sparsity awareness to reduce communication overhead.

Environment runtime. Environments run in CPU-only containers on a separate Kubernetes cluster, communicating with rollout workers via K8S API calls. This isolates environment execution from GPU workloads. ROSE's design is orthogonal to environment placement and extends to GPU-accelerated environments.

6 Performance Evaluation

Models and Training Configurations. We use Qwen3-8B/32k for FrozenLake [9] and Qwen3-32B/32k for ALFWorld [57]. Both are widely adopted benchmarks in the agentic RL literature [12, 17, 63]. We train them with GRPO [52] and DAPO [80]. For all tasks, we set the group size to 16 and use batch sizes of 256 and 1024 for Qwen3-8B and Qwen3-32B, respectively. For DAPO, the rollout stage continues until it collects the target number of trajectory groups with non-zero reward variance. We adjust the maximum number of concurrent trajectories at each step based on the previous step to speed up the collection of valid trajectories. We train the 8B and 32B models on 16 and 48 dedicated GPUs, with (Rollout, Training) allocations of (8, 8) and (16, 32). Rollout TP is 1 (8B) and 4 (32B), while training parallelism (TP, PP, CP) is (4, 1, 1) and (8, 1, 2). We cap borrowed serving GPUs at 16 (8B) and 64 (32B), and set the per-device rollout batch size to 16 to avoid contention (see Appendix A).

Cluster Setup. We run agentic RL training on an H800 cluster with up to 48 GPUs and online serving on a separate H800 cluster with 64 GPUs. Within each cluster, nodes are connected via 400 Gbps InfiniBand. Due to operational constraints, our evaluation uses a 200 Gbps Ethernet link for cross-cluster communication. We further evaluate communication efficiency under different link bandwidths in §6.3. We use `NCCL` and `Mooncake Store` [45] for intra- and cross-cluster weight transfer. For the serving cluster, we pre-deploy Qwen3-8B and Qwen3-32B as rollout models, which are used by 40% and 19% of agentic RL training jobs in our cluster, respectively, and together cover most workloads. We co-locate them with similarly sized heterogeneous serving models (Qwen3-8B with Qwen2.5-7B, Qwen3-32B with Qwen2.5-32B) because they use compatible parallelism configurations. We set the prefill-to-decoding instance ratio to 1:3. We replay the 24-hour online serving trace from Microsoft [59], and run a load-balancing policy to route online serving requests.

Baselines and Training Recipe. We evaluate ROSE against state-of-the-art RL post-training systems. ROSE is agnostic to the choice of RL algorithms. Due to the substantial overhead of agentic RL, we adopt one-step off-policy training [37] for most baselines and additionally evaluate fully asynchronous training with AReaL [13]. We first compare ROSE with three resource-fixed baselines in an end-to-end training evaluation: (1) **ROLL-GRPO**, which trains with GRPO [52] on ROLL [63] [2]; (2) **ROLL-DAPO**, which trains with DAPO [80] on ROLL; and (3) **AReaL**, a fully asynchronous RL system. On ROSE and all these baselines, we train until model convergence. Specifically, for GRPO and AReaL, we train Qwen3-8B and Qwen3-32B for 100 and 40 steps, and for DAPO we use 50 and 25 steps, respectively. We also compare rollout performance with two elastic baselines that leverage additional compute: (4) **λRL**, a serverless RL baseline inspired by existing commercial products [41, 61], where the number of serverless GPUs matches the maximum available spot GPUs and the function timeout is set to 15 minutes [1]; and (5) **RLBoost+**, a strengthened RLBoost [69] that uses the same training and rollout GPU allocation as ROLL, but additionally utilizes spot instances in one-step off-policy mode. To capture the impact of spot GPU availability, we reproduce a 2-hour window characterized by high resource volatility from the traces (see Appendix B) and run elastic baselines within this window.

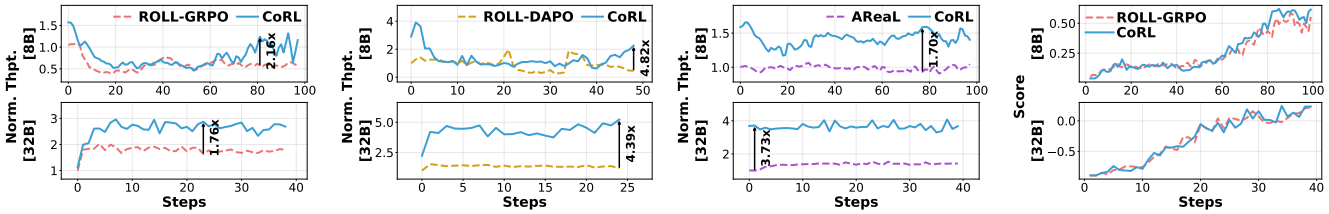
Metrics. We measure training throughput as the total number of input and output tokens processed per global step divided by the step time [23, 88]. We use the average critic score as the accuracy metric. For serving, we set P99 SLOs for (TPOT, TTFT) to (150 ms, 500 ms) for Qwen2.5-7B and (450 ms, 1000 ms) for Qwen2.5-32B [82, 87].

6.1 End-to-End Evaluation

Model Convergence. We report the critic scores over the training in Figure 7d. It shows that ROSE preserves the model convergence due to its algorithm-agnostic design, achieving final scores nearly identical to the baselines.

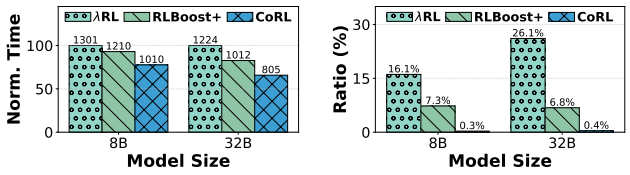
End-to-End Throughput. Figure 7a-c compares the end-to-end throughput against the baselines with different model and tasks. Compared with ROLL, ROSE achieves average throughput improvements of 1.31× and 1.46× with GRPO (see Figure 7a), the maximum throughput increase is 2.16× and 1.76×. The advantage of ROSE becomes more pronounced under the DAPO algorithm, where average throughput increases by 1.42× and 3.31× with a maximum of 4.82× and 4.39× (see Figure 7b). Compared with AReaL, ROSE achieves

[2] We choose ROLL over veRL [55] for better support of agentic tasks.



(a) One-Off w/ GRPO. **(b)** One-Off w/ DAPO. **(c)** Fully-Async w/ AReaL. **(d)** GRPO Critic Scores.

Figure 7. (a)-(c) ROSE's end-to-end throughput improvements compared with baselines, for each baseline we run 8B and 32B model. The data are normalized to the baseline's first step. (d) End-to-end critic scores for 8B and 32B models using GRPO.



(a) Elastic Baselines. **(b)** Allocation Overhead.

Figure 8. End-to-end evaluation. (a) Rollout time and (b) Allocation overhead compared with elastic baselines.

Table 1. ROSE vs. alternative serving engines: rollout time (s) and P99 serving SLO (ms).

Model	Method	Rollout	TTFT	TPOT
8B	ROSE	496.3	338.1	136.1
	ServerlessLLM [14]	–	314.8	117.8
	ServerlessLLM+Rollout	651.7	1166.1	135.6
	Prism [81]	731.7	973.2	115.4
32B	ROSE	960.1	837.5	398.1
	ServerlessLLM [14]	–	716.1	312.8
	ServerlessLLM+Rollout	1161.8	2426.2	565.3
	Prism [81]	1301.2	1625.4	351.7

1.44× and 2.69× higher throughput on average (see Figure 7c). Although AReaL eliminates GPU idle time by continuously generating trajectories without waiting for training to complete, by expanding effective GPU capacity through cooperative elasticity, ROSE provides gains orthogonal to asynchronous execution.

The throughput gains are driven by three main design elements. (1) *Cooperative elasticity via co-serving executor*: by harvesting idle serving GPUs, ROSE effectively expands the rollout GPU pool. With 16 additional serving GPUs, rollout time decreases by 1.69× (see Appendix E), confirming that the co-serving executor successfully converts serving slack into rollout capacity. (2) *Elastic rollout scheduler*: DAPO's redundant sampling can launch up to 5.7× the base batch size (Figure 1d). The resource-fixed baseline cannot absorb this burst and suffers severe contention, whereas ROSE's scheduler dynamically offloads excess trajectories to serving GPUs. The scheduler's turn-wise routing and KVC affinity together improve rollout efficiency by up to 1.48× (Table 3). (3) *Weight transfer engine*: cross-cluster synchronization completes within 21 s for Qwen3-32B (§6.3), preventing weight updates from becoming a bottleneck between steps. Moreover, ROSE eliminates the allocation overhead that plagues elastic baselines (<0.5% vs. 6.8–26.1% for RLBoost+ and λRL, Figure 8b). Across all experiments, ROSE meets the serving P99 latency SLOs (§6.2).

Comparison with Elastic Baselines. Figure 8a compares rollout efficiency against λRL and RLBoost+ throughout the training using GRPO. λRL uses 16 and 32 serverless GPUs for rollouts of the 8B and 32B models, respectively, which is consistent with the maximum number of GPUs available to RLBoost+. We observe that allocating more GPUs to rollouts yields up to 1.31× and 1.23× speedups over ROLL for Qwen3-8B and Qwen3-32B, respectively. Although RLBoost+ experiences fluctuating spot GPU availability, its resource allocation changes less frequently than that of λRL, which reallocates resources at a fixed 15-minute lease interval. As a result, RLBoost+ achieves 1.41× and 1.48× speedups over ROLL for Qwen3-8B and Qwen3-32B, respectively, although its gains are still constrained by the instability of spot GPUs. ROSE further reduces rollout time by 1.20× and 1.26× compared to RLBoost+. This improvement suggests that opportunistically leveraging serving GPUs provides more abundant and stable compute for rollouts while avoiding frequent preemption, thereby shortening rollout time.

Allocation Overhead. We further analyze the allocation overhead of each elastic scheme. We define *preempted GPU time* as the number of preempted GPUs multiplied by the per-preemption recovery time, normalized by total GPU time. As shown in Figure 8b, λRL incurs the highest overhead (up to 26.1%), because serverless GPUs are leased for fixed durations often shorter than a complete rollout, leading to frequent preemptions and restarts. RLBoost+ experiences less frequent preemptions on spot instances but still incurs non-negligible overhead (6.8–7.3%). In contrast, ROSE requires only a one-time model activation of at most five seconds, keeping the allocation overhead ratio below 1%.

Comparison with Alternative Serving Engines. Due to the high cost of full training runs, we compare alternative serving engines at a single checkpoint (step 50 for 8B, step 20 for 32B) that exhibits moderately stable rollout time and fluctuating serving load. Each experiment runs ten RL steps.

Table 2. [Co-Serve Executor]. The impact of memory sharing policy for rollout and SLO efficiency.

Model	Policy	Rollout	TTFT (ms)		TPOT (ms)	
		Time (s)	P95	P99	P95	P99
Qwen3-8B	Static Partition	776.7	299.5	330.95	1462.5	1488.2
	+ Memory Preemption	591.7	348.8	517.9	153.6	162.8
	+ Prefix Caching	469.3	318.5	333.6	173.2	186.3
Qwen3-32B	Static Partition	1310.5	595.6	603	6835.9	7021.1
	+ Memory Preemption	959.7	596.7	651.3	493.5	503.8
	+ Prefix Caching	920.4	607.2	640.6	477.9	483.2

Bidirectional Autoscaling. We replace ROSE's co-serving executor with ServerlessLLM [14]. As shown in Table 1, ServerlessLLM achieves low serving latency without rollouts (TTFT-P99: 314.8 ms for 8B), but severely violates SLOs when rollouts are enabled (1166 ms and 2426 ms). Repeated rollout model eviction and reloading during serving bursts inflates rollout time by 1.31× (8B) and 1.21× (32B), confirming that bidirectional autoscaling cannot safely harvest serving slack under fluctuating traffic.

GPU Multiplexing. We evaluate Prism [81], a heterogeneous LLM colocation engine, configured to co-schedule rollout and serving requests (rollout SLO = 4× serving SLO). Prism does not provide dual-SLO admission support or prefix caching for rollouts. As shown in Table 1, the lack of prefix caching inflates rollout time by 1.47× (8B) and 1.36× (32B), while the absence of dual-SLO admission support leads to TTFT SLO violations (973 ms and 1625 ms vs. 500 ms and 1000 ms targets). This demonstrates that generic multiplexing without RL-aware co-serving is insufficient.

6.2 Analysis of Co-serving Executor

To analyze the efficiency of the co-serving executor, we use the end-to-end setups for Qwen3-8B and Qwen3-32B and run GRPO for five steps. We utilize the same model checkpoint and serving traffic in Table 1.

Preemptive Dynamic Memory Sharing Policy. This policy incorporates two memory-sharing optimizations, namely *memory preemption* and *prefix caching*. We run rollouts on serving GPUs via spatial GPU sharing and measure end-to-end rollout time, along with serving P95 and P99 TTFT and TPOT, to quantify rollout efficiency and SLO compliance. We first evaluate memory preemption. As a *static partition* baseline, we split GPU memory evenly between serving and rollout LLMs. Compared to *static partitioning*, memory preemption reduces P99 TPOT latency by 9.14× for Qwen3-8B and 13.94× for Qwen3-32B, indicating that it absorbs bursty serving-side memory demand and reduces tail latency.

Second, we enable *prefix caching* atop memory preemption. This reduces rollout time by 1.26× for Qwen3-8B and 1.04× for Qwen3-32B, with only a slight increase in serving tail latency. These results indicate that prefix caching mainly improves rollout throughput, but does not by itself ensure serving SLO compliance. Overall, the memory sharing policy

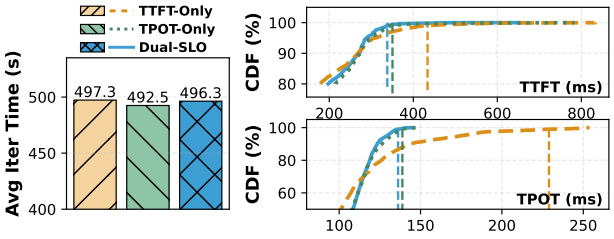
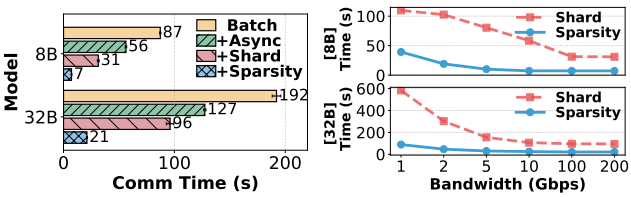


Figure 9. Analysis of Dual-SLO Admission Controller.



(a) Weight transfer overhead. **(b)** Sensitivity to bandwidth.

Figure 10. [Transfer Engine]. (a) Cross-cluster weight transfer time under different optimizations; each optimization is additive over the previous one. (b) Sensitivity of shard-aware and sparsity-aware transfer of different LLMs to cross-cluster bandwidth.

effectively limits tail-latency inflation, but without explicit SLO-aware scheduling, P99 latency still misses our target.

Dual-SLO Admission Controller. This controller compares the current serving TTFT and TPOT against their SLO targets and time-multiplexes between serving traffic and rollout requests to maintain SLO compliance. We evaluate three policies under the same setup in Figure 9: *TTFT-only*, *TPOT-only*, and *Dual-SLO*. *TTFT-only* enforces only the TTFT SLO, while *TPOT-only* enforces only the TPOT SLO. The average step time is similar across all three policies, suggesting that the policy choice has limited impact on rollout time. Compared to the single-objective baselines, *Dual-SLO* consistently achieves lower P99 tail latency on both metrics. It reduces P99 TPOT by 3.4% and 76.3% relative to *TPOT-only* and *TTFT-only*, respectively, and reduces P99 TTFT by 3.7% and 28.4% over the same baselines. Overall, *Dual-SLO* explicitly provides guarantees for both TTFT and TPOT while preserving rollout efficiency. We further explore the sensitivity of the co-serving executor to prefix caching lease time (Appendix C) and online serving load (Appendix D), demonstrating its robustness.

6.3 Effectiveness of Transfer Engine

We evaluate the transfer engine using 16 GPUs for training and 16 GPUs for serving, and measure cross-cluster communication overhead over three RL steps. Unless otherwise specified, the training and serving clusters are connected by a 200 Gbps Ethernet link.

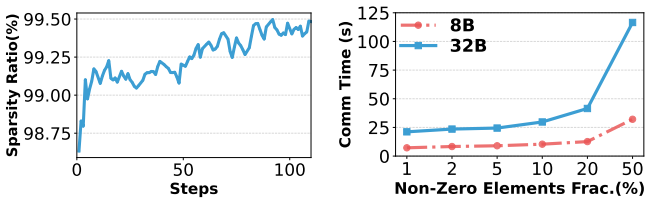
Table 3. [Rollout Scheduler]. The speedup of the elastic rollout scheduler on rollout time.

Policy	Qwen3-8B	Qwen3-32B
Baseline	1.00×	1.00×
+ Turn-Wise Routing	1.11×	1.08×
+ KVC Affinity	1.16×	1.48×

Analysis of Weight Transfer Optimizations. Figure 10a quantifies the cross-cluster communication efficiency of *asynchrony*, *shard-awareness*, and *sparsity-awareness* for Qwen3-8B and Qwen3-32B. As a baseline (*batch*), training workers all-gather model weights and transmit the entire model, and then serving workers pull model weights from relay workers into GPU memory. *Asynchronous* transfer streams parameters at bucket granularity and pipelines publishing with pulling, reducing end-to-end communication time by 1.6× (Qwen3-8B) and 1.5× (Qwen3-32B). *Shard-awareness* enables each training worker to publish only its local shard, and each serving worker to pull only the shards it hosts, further reducing communication time by 1.8× (Qwen3-8B) and 1.3× (Qwen3-32B). *Sparsity-awareness* provides the largest gain, reducing communication time by 4.4× (Qwen3-8B) and 4.6× (Qwen3-32B). Overall, our optimized cross-cluster weight transfer engine reduces end-to-end communication time by 12.4× for Qwen3-8B and 9.1× for Qwen3-32B, bringing cross-cluster transfer down to tens of seconds. Detailed timeline breakdown is provided in Appendix F.

Sensitivity to Cross-Cluster Bandwidth. We study the sensitivity of shard-aware and sparsity-aware transfer to cross-cluster bandwidth by throttling link capacity from 200 Gbps to 1 Gbps using `tc` command, and measuring end-to-end communication overhead for Qwen3-8B (Figure 10b, top) and Qwen3-32B (Figure 10b, bottom). For Qwen3-8B, shard-aware transfer increases from 31 s (200 Gbps) to 109 s (1 Gbps), while sparsity-aware transfer stays within 7–39 s, delivering a 2.8×–4.4× speedup. For Qwen3-32B, shard-aware transfer grows from 96 s to 584 s, whereas sparsity-aware transfer remains within 21–89 s (4.6×–6.6× faster). Overall, sparsity-awareness flattens the scaling curve by reducing transferred bytes. Even at 5 Gbps, transfer completes within 10 s (Qwen3-8B) and 30 s (Qwen3-32B). Because cross-cluster transfer is overlapped with training and intra-cluster communication, it increases training time by at most 5%.

Analysis of Weight Differential Sparsity. Figure 11a shows that the weight-differential sparsity for Qwen3-8B stays around 99% throughout training. This indicates that sparsity is persistent rather than a transient artifact of a particular training phase. We next vary the non-zero fraction in Figure 11b for Qwen3-8B and Qwen3-32B to evaluate the sensitivity of our sparsity-aware transfer engine to the sparsity ratio. As the non-zero fraction increases, communication overhead rises and the benefit of sparse transfer diminishes. Beyond ~20%, sparse-format metadata (e.g., indices) and (de)sparsification overhead begin to offset the reduction in transmitted weights. In our workloads, the measured non-zero fraction remains well below this threshold, enabling consistently efficient weight propagation.



(a) Sparsity Ratio [8B]. **(b)** Sensitivity to Sparsity.

Figure 11. [Analysis of Sparsity]. (a) The sparsity of weight differentials across steps for Qwen3-8B. (b) The sensitivity of transfer engine to sparsity.

6.4 Effectiveness of Rollout Scheduler.

We follow the end-to-end setups and evaluate the elastic rollout scheduler using Qwen3-8B and Qwen3-32B with GRPO algorithm for the first five RL steps. Table 3 analyzes the contribution of two heuristics adopted by the rollout scheduler. As a baseline, we pin each trajectory to a fixed rollout worker for its entire lifetime. This static assignment leads to load imbalance because trajectory runtimes vary widely. As a result, enabling turn-wise routing reduces rollout time by 1.11× (Qwen3-8B) and 1.08× (Qwen3-32B), demonstrating the benefit of fine-grained, flexible routing. Adding KVC affinity yields further improvements, bringing the cumulative speedup to 1.16× for Qwen3-8B and 1.48× for Qwen3-32B. This is because larger models incur heavier prefill costs, making KV reuse more effective at reducing per-turn overhead. Overall, these simple heuristics substantially improve rollout efficiency. The scheduler adds negligible overhead (at most 10 ms), suggesting that it scales well.

7 Related Works

Agentic RL Training Systems. Many systems optimize conventional single-turn RL post-training [5, 16, 19, 22–24, 44, 54, 69, 78, 88, 89]. The rise of agentic LLMs has also motivated agentic RL training frameworks [4, 12, 17, 57, 63, 68]. However, these agentic systems typically assume fixed resources. A few elastic RL systems exploit spot instances [69] or serverless GPUs [41, 61]. ROSE explores underutilized serving GPUs for resource elasticity and is complementary to these elastic approaches.

Serving GPU Sharing. GPU multiplexing has been widely studied. Prior DL systems [15, 18, 32, 72, 85] interleave multiple models on the same GPUs to improve utilization. Recent LLM serving systems extend multiplexing to co-serve multiple LLMs: systems targeting homogeneous models [43] share a single KVC layout across co-located workloads, while

heterogeneity-aware systems [7, 71, 73, 81, 84] multiplex workloads with comparable SLO targets. Lyra [29] loans idle serving servers to training at machine granularity. ROSE targets a different setting: co-locating heterogeneous models with asymmetric SLO requirements on the same GPU.

Sparsity-based Optimization. Recent systems including Check-N-Run [8] and LowDiff [77] leverage the sparsity to reduce storage and checkpoint overhead. Many communication optimization systems [2, 10, 33, 66] exploit gradient sparsity to cut communication cost with lossy compression. Inspired by these works, we observe the sparsity in the weight differential of RL training and leverage it to reduce the communication overhead with lossless compression.

Cycle Stealing. Harvesting idle resources across workloads is a well-studied concept. Cycle stealing [21] demonstrated that beneficiaries gain unbounded benefit from donors' idle cycles with only slight donor penalty. In the GPU era, Ekya [3] balances inference and continuous retraining on edge GPUs, and Lyra [29] loans idle serving GPUs to training jobs. ROSE extends this philosophy to co-serving heterogeneous LLMs, additionally handling incompatible KVC layouts, prefix caching contention, and cross-cluster weight synchronization.

8 Conclusion

In this paper, we present ROSE, a cooperative, elastic post-training system that opportunistically harvests serving GPUs to accelerate agentic RL training. ROSE combines (i) a co-serving executor for SLO-safe compute and memory sharing, (ii) a cross-cluster weight transfer engine for low-overhead weight synchronization, and (iii) an elastic rollout scheduler that efficiently realizes the cooperative elasticity. Extensive experiments show that ROSE improves training efficiency over baselines while preserving serving SLOs.

References

[1] Alibaba Cloud. 2026. Creating a GPU function. https://www.alibabacloud.com/help/en/functioncompute/fc/user-guide/creating-a-gpu-function/. (2026). Accessed: 2026-04.

[2] Dan Alistarh, Demjan Grubic, Jerry Z. Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: communication-efficient SGD via gradient quantization and encoding. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1707–1718.

[3] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. 2022. Ekya: Continuous Learning of Video Analytics Models on Edge Compute Servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 119–135. https://www.usenix.org/conference/nsdi22/presentation/bhardwaj

[4] Shiyi Cao, Dacheng Li, Fangzhou Zhao, Shuo Yuan, Sumanth R. Hegde, Connor Chen, Charlie Ruan, Tyler Griggs, Shu Liu, Eric Tang, Richard Liaw, Philipp Moritz, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. SkyRL-Agent: Efficient RL Training for Multi-turn LLM Agent. *arXiv preprint arXiv:2511.16108* (2025).

[5] Rongxin Cheng, Kai Zhou, Xingda Wei, Siyuan Liu, Mingcong Han, Mingjing Ai, Yeju Zhou, Baoquan Zhong, Wencong Xiao, Rong Chen, and Haibo Chen. 2025. Fast LLM Post-training via Decoupled and Best-of-N Speculation. *arXiv preprint arXiv:2511.16193* (2025).

[6] Yihua Cheng, Yuhan Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. 2025. LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference. *arXiv preprint arXiv:2510.09665* (2025).

[7] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. 2024. MuxServe: Flexible Spatial-Temporal Multiplexing for Multiple LLM Serving. In *ICML*.

[8] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. Check-N-Run: A checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 929–943.

[9] Farama Foundation. 2024. Gymnasium - FrozenLake Environment. https://gymnasium.farama.org/environments/toy_text/frozen_lake/. (2024). Accessed: 2025-09.

[10] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. 2021. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 676–691.

[11] Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. 2025. Group-in-Group Policy Optimization for LLM Agent Training. *arXiv preprint arXiv:2505.10978* (2025).

[12] Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. 2025. Group-in-Group Policy Optimization for LLM Agent Training. *arXiv preprint arXiv:2505.10978* (2025).

[13] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. 2025. AReaL: A Large-Scale Asynchronous Reinforcement Learning System for Language Reasoning. *arXiv preprint arXiv:2505.10978* (2025).

[14] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *OSDI'24*.

[15] Wei Gao, Zhuoyuan Ouyang, Peng Sun, Tianwei Zhang, and Yonggang Wen. 2025. IceFrog: A Layer-Elastic Scheduling System for Deep Learning Training in GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems* 36, 6 (2025), 1071–1086. https://doi.org/10.1109/TPDS.2025.3553137

[16] Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, Jiamang Wang, Lin Qu, Bo Zheng, and Wei Wang. 2025. RollPacker: Mitigating Long-Tail Rollouts for Fast, Synchronous RL Post-Training. *arXiv preprint arXiv:2509.21009* (2025).

[17] Wei Gao, Yuheng Zhao, Tianyuan Wu, Shaopan Xiong, Weixun Wang, Dakai An, Lunxi Cao, Dilxat Muhtar, Zichen Liu, Haizhou Zhao, Ju Huang, Siran Yang, Yongbin Li, Wenbo Su, Jiamang Wang, Lin Qu, Bo Zheng, and Wei Wang. 2025. RollArt: Scaling Agentic RL Training via Disaggregated Infrastructure. *arXiv preprint arXiv:2512.22560* (2025).

[18] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 539–558.

[19] Zhenyu Han, Ansheng You, Haibo Wang, Kui Luo, Guang Yang, Wenqi Shi, Menglong Chen, Sicheng Zhang, Zeshun Lan, Chunshi Deng, Huazhong Ji, Wenjie Liu, Yu Huang, Yixiang Zhang, Chenyi Pan, Jing Wang, Xin Huang, Chunsheng Li, and Jianping Wu. 2025. AsyncFlow: An Asynchronous Streaming RL Framework for Efficient LLM Post-Training. *arXiv preprint arXiv:2507.01663* (2025).

[20] Bingguang Hao, Maolin Wang, Zengzhuang Xu, Yicheng Chen, Cunyin Peng, Jinjie GU, and Chenyi Zhuang. 2025. Exploring Superior Function Calls via Reinforcement Learning. *arXiv preprint arXiv:2508.05118*

(2025).

[21] Mor Harchol-Balter, Cuihong Li, Takayuki Osogami, Alan Scheller-Wolf, and Mark S. Squillante. 2003. Cycle stealing under immediate dispatch task assignment. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '03)*. Association for Computing Machinery, New York, NY, USA, 274–285. https://doi.org/10.1145/777412.777462

[22] Eric Harper, Somshubra Majumdar, Oleksii Kuchaiev, Li Jason, Yang Zhang, Evelina Bakhturina, Vahid Noroozi, Sandeep Subramanian, Koluguri Nithin, Huang Jocelyn, Fei Jia, Jagadeesh Balam, Xuesong Yang, Micha Livne, Yi Dong, Sean Naren, and Boris Ginsburg. 2025. NeMo: a toolkit for Conversational AI and Large Language Models. (2025). https://github.com/NVIDIA/NeMo

[23] Jingkai He, Tianjian Li, Erhu Feng, Dong Du, Qian Liu, Tao Liu, Yubin Xia, and Haibo Chen. 2025. History Rhymes: Accelerating LLM Reinforcement Learning with RhymeRL. *arXiv preprint arXiv:2508.18588* (2025).

[24] Jian Hu, Xibin Wu, 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).

[25] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: scaling large language model training to more than 10,000 GPUs. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*. USENIX Association, USA, Article 41, 16 pages.

[26] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv preprint arXiv:2310.06770* (2024).

[27] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. 2023. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th annual international symposium on computer architecture*. 1–14.

[28] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[29] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. 2023. Lyra: Elastic Scheduling for Deep Learning Clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, USA, 835–850. https://doi.org/10.1145/3552326.3587445

[30] Yufei Li, Zexin Li, Yinglun Zhu, and Cong Liu. 2025. Lemix: Unified Scheduling for Llm Training and Inference on Multi-Gpu Systems. In *2025 IEEE Real-Time Systems Symposium (RTSS)*.

[31] Zhiwei Li, Yong Hu, and Wenqing Wang. 2025. Encouraging Good Processes Without the Need for Good Answers: Reinforcement Learning for LLM Agent Planning. *arXiv preprint arXiv:2508.19598* (2025).

[32] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 663–679.

[33] Hwijoon Lim, Juncheol Ye, Sangeetha Abdu Jyothi, and Dongsu Han. 2024. Accelerating model training in multi-cluster environments with consumer-grade gpus. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 707–720.

[34] Yuhang Liu, Pengxiang Li, Congkai Xie, Xavier Hu, Xiaotian Han, Shengyu Zhang, Hongxia Yang, and Fei Wu. 2025. Infigui-r1: Advancing multimodal gui agents from reactive actors to deliberative reasoners. *arXiv preprint arXiv:2504.14239* (2025).

[35] Han Lu, Zichen Liu, Shaopan Xiong, Yancheng He, Wei Gao, Yanan Wu, Weixun Wang, Jiashun Liu, Yang Li, Haizhou Zhao, Ju Huang, Siran Yang, Xiaoyang Li, Yijia Luo, Zihe Liu, Ling Pan, Junchi Yan, Wei Wang, Wenbo Su, Jiamang Wang, Lin Qu, and Bo Zheng. 2025. Part II: ROLL Flash – Accelerating RLVR and Agentic Training with Asynchrony. *arXiv preprint arXiv:2510.11345* (2025).

[36] Zhengxi Lu, Yuxiang Chai, Yaxuan Guo, Xi Yin, Liang Liu, Hao Wang, Han Xiao, Shuai Ren, Guanjing Xiong, and Hongsheng Li. 2025. UI-R1: Enhancing Efficient Action Prediction of GUI Agents by Reinforcement Learning. *arXiv preprint arXiv:2503.21620* (2025).

[37] Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. 2025. DeepCoder: A Fully Open-Source 14B Coder at O3-mini Level. https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-mini-Level-1cf81902c14680b3bee5eb349a512a51. (2025). Notion Blog.

[38] Run Luo, Lu Wang, Wanwei He, and Xiaobo Xia. 2025. GUI-R1 : A Generalist R1-Style Vision-Language Action Model For GUI Agents. *arXiv preprint arXiv:2504.10458* (2025).

[39] 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.

[40] Aashiq Muhamed, Oscar Li, David Woodruff, Mona Diab, and Virginia Smith. 2024. Grass: Compute efficient low-memory llm training with structured sparse gradients. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 14978–15003.

[41] OpenPipe. 2025. Serverless RL. (2025). https://openpipe.ai/blog/serverless-rl

[42] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2025. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24)*. IEEE Press, 118–132. https://doi.org/10.1109/ISCA59077.2024.00019

[43] Yifan Qiao, Shu Anzai, Shan Yu, Haoran Ma, Shuo Yang, Yang Wang, Miryung Kim, Yongji Wu, Yang Zhou, Jiarong Xing, Joseph E. Gonzalez, Ion Stoica, and Harry Xu. 2025. ConServe: Fine-Grained GPU Harvesting for LLM Online and Offline Co-Serving. *arXiv preprint arXiv:2410.01228* (2025).

[44] Ruoyu Qin, Weiran He, Weixiao Huang, Yangkun Zhang, Yikai Zhao, Bo Pang, Xinran Xu, Yingdi Shan, Yongwei Wu, and Mingxing Zhang. 2025. Seer: Online Context Learning for Fast Synchronous LLM Reinforcement Learning. *arXiv preprint arXiv:2511.14617* (2025).

[45] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Heyi Tang, Feng Ren, Teng Ma, Shangming Cai, Yineng Zhang, Mingxing Zhang, et al. 2024. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *ACM Transactions on Storage* (2024).

[46] Haoran Qiu, Anish Biswas, Zihan Zhao, Jayashree Mohan, Alind Khare, Esha Choukse, Íñigo Goiri, Zeyu Zhang, Haiying Shen, Chetan Bansal, Ramachandran Ramjee, and Rodrigo Fonseca. 2025. ModServe: Modality- and Stage-Aware Resource Disaggregation for Scalable Multimodal Model Serving. In *Proceedings of the 2025 ACM Symposium on Cloud Computing (SoCC 2025)*. Association for Computing Machinery, New York, NY, USA.

[47] Mrinal Rawat, Ambuje Gupta, Rushil Goomer, Alessandro Di Bari, Neha Gupta, and Roberto Pieraccini. 2025. Pre-Act: Multi-Step Planning and Reasoning Improves Acting in LLM Agents. *arXiv preprint arXiv:2505.09970* (2025).

[48] Amir Sarfi, Benjamin Thérien, Joel Lidin, and Eugene Belilovsky. 2025. Communication Efficient LLM Pre-training with SparseLoCo. (2025). arXiv:cs.LG/2508.15706 https://arxiv.org/abs/2508.15706

[49] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[50] SGLang Team. 2025. SGLang: Fast Serving Framework for Large Language Models. https://github.com/sgl-project/sglang. (2025). Version 0.4.

[51] Zelei Shao, Vikranth Srivatsa, Sanjana Srivastava, Qingyang Wu, Alpay Ariyak, Xiaoxia Wu, Ameen Patel, Jue Wang, Percy Liang, Tri Dao, Ce Zhang, Yiying Zhang, Ben Athiwaratkun, Chenfeng Xu, and Junxiong Wang. 2025. Beat the long tail: Distribution-Aware Speculative Decoding for RL Training. *arXiv preprint arXiv:2511.13841* (2025).

[52] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300* (2024).

[53] Guangming Sheng, Yuxuan Tong, Borui Wan, Wang Zhang, Chaobo Jia, Xibin Wu, Yuqi Wu, Xiang Li, Chi Zhang, Yanghua Peng, Haibin Lin, Xin Liu, and Chuan Wu. 2025. Laminar: A Scalable Asynchronous RL Post-Training Framework. *arXiv preprint arXiv:2510.12633* (2025).

[54] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. HybridFlow: A Flexible and Efficient RLHF Framework. *arXiv preprint arXiv:2409.19256* (2024).

[55] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. verl: Volcano Engine Reinforcement Learning for LLM. https://github.com/volcengine/verl. (2024).

[56] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[57] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2021. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. *arXiv preprint arXiv:2010.03768* (2021).

[58] Joykirat Singh, Raghav Magazine, Yash Pandya, and Akshay Nambi. 2025. Agentic Reasoning and Tool Integration for LLMs via Reinforcement Learning. *arXiv preprint arXiv:2505.01441* (2025).

[59] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. Dynamollm: Designing llm inference clusters for performance and energy efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1348–1362.

[60] The Terminal-Bench Team. 2025. Terminal-Bench: A Benchmark for AI Agents in Terminal Environments. (2025). https://github.com/laude-institute/terminal-bench

[61] Thinking Machines AI. 2025. Tinker. https://thinkingmachines.ai/tinker/. (2025). Accessed: 2026-02.

[62] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. 2025. KVCache cache in the wild: characterizing and optimizing KVCache cache at a large cloud provider. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '25)*. USENIX Association, USA, Article 28, 18 pages.

[63] Weixun Wang, Shaopan Xiong, Gengru Chen, Wei Gao, Sheng Guo, Yancheng He, Ju Huang, Jiaheng Liu, Zhendong Li, Xiaoyang Li, Zichen Liu, Haizhou Zhao, Dakai An, Lunxi Cao, Qiyang Cao, Wanxi Deng, Feilei Du, Yiliang Gu, Jiahe Li, Xiang Li, Mingjie Liu, Yijia Luo, Zihe Liu, Yadao Wang, Pei Wang, Tianyuan Wu, Yanan Wu, Yuheng Zhao, Shuaibing Zhao, Jin Yang, Siran Yang, Yingshui Tan, Huimin Yi, Yuchi Xu, Yujin Yuan, Xingyao Zhang, Lin Qu, Wenbo Su, Wei Wang, Jiamang Wang, and Bo Zheng. 2025. Reinforcement Learning Optimization for Large-Scale Learning: An Efficient and User-Friendly Scaling Library. *arXiv preprint arXiv:2506.06122* (2025).

[64] Weixun Wang, XiaoXiao Xu, Wanhe An, Fangwen Dai, Wei Gao, Yancheng He, Ju Huang, Qiang Ji, Hanqi Jin, Xiaoyang Li, Yang Li, Zhongwen Li, Shirong Lin, Jiashun Liu, Zenan Liu, Tao Luo, Dilxat Muhtar, Yuanbin Qu, Jiaqiang Shi, Qinghui Sun, Yingshui Tan, Hao Tang, Runze Wang, Yi Wang, Zhaoguo Wang, Yanan Wu, Shaopan Xiong, Binchen Xu, Xander Xu, Yuchi Xu, Qipeng Zhang, Xixia Zhang, Haizhou Zhao, Jie Zhao, Shuaibing Zhao, Baihui Zheng, Jianhui Zheng, Suhang Zheng, Yanni Zhu, Mengze Cai, Kerui Cao, Xitong Chen, Yue Dai, Lifan Du, Tao Feng, Tao He, Jin Hu, Yijie Hu, Ziyu Jiang, Cheng Li, Xiang Li, Jing Liang, Xin Lin, Chonghuan Liu, ZhenDong Liu, Zhiqiang Lv, Haodong Mi, Yanhu Mo, Junjia Ni, Shixin Pei, Jingyu Shen, XiaoShuai Song, Cecilia Wang, Chaofan Wang, Kangyu Wang, Pei Wang, Tao Wang, Wei Wang, Ke Xiao, Mingyu Xu, Tiange Xu, Nan Ya, Siran Yang, Jianan Ye, Yaxing Zang, Duo Zhang, Junbo Zhang, Boren Zheng, Wanxi Deng, Ling Pan, Lin Qu, Wenbo Su, Jiamang Wang, Wei Wang, Hu Wei, Minggang Wu, Cheng Yu, Bing Zhao, Zhicheng Zheng, and Bo Zheng. 2026. Let It Flow: Agentic Crafting on Rock and Roll, Building the ROME Model within an Open Agentic Learning Ecosystem. *arXiv preprint arXiv:2512.24873* (2026).

[65] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. 2025. BurstGPT: A Real-world Workload Dataset to Optimize LLM Serving Systems. *arXiv preprint arXiv:2401.17644* (2025).

[66] Zhuang Wang, Zhaozhuo Xu, Jingyi Xi, Yuke Wang, Anshumali Shrivastava, and TS Eugene Ng. 2025. {ZEN}: Empowering Distributed Training with Sparsity-driven Data Synchronization. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 537–556.

[67] Junde Wu, Jiayuan Zhu, Yuyuan Liu, Min Xu, and Yueming Jin. 2025. Agentic reasoning: A streamlined framework for enhancing llm reasoning with agentic tools. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 28489–28503.

[68] Tianyuan Wu, Lunxi Cao, Yining Wei, Wei Gao, Yuheng Zhao, Dakai An, Shaopan Xiong, Zhiqiang Lv, Ju Huang, Siran Yang, Yinghao Yu, Jiamang Wang, Lin Qu, and Wei Wang. 2025. RollMux: Phase-Level Multiplexing for Disaggregated RL Post-Training. *arXiv preprint arXiv:2512.11306* (2025).

[69] Yongji Wu, Xueshen Liu, Haizhong Zheng, Juncheng Gu, Beidi Chen, Z. Morley Mao, Arvind Krishnamurthy, and Ion Stoica. 2025. RLBoost: Harvesting Preemptible Resources for Cost-Efficient Reinforcement Learning on LLMs. *arXiv preprint arXiv:2510.19225* (2025).

[70] Bingquan Xia, Bowen Shen, Cici, Dawei Zhu, Di Zhang, Gang Wang, Hailin Zhang, Huaqiu Liu, Jiebao Xiao, Jinhao Dong, Liang Zhao, Peidian Li, Peng Wang, Shihua Yu, Shimao Chen, Weikun Wang, Wenhan Ma, Xiangwei Deng, Yi Huang, Yifan Song, Zihan Jiang, Bowen Ye, Can Cai, Chenhong He, Dong Zhang, Duo Zhang, Guoan Wang, Hao Tian, Haochen Zhao, Heng Qu, Hongshen Xu, Jun Shi, Kainan Bao, Kai Fang, Kang Zhou, Kangyang Zhou, Lei Li, Menghang Zhu, Nuo Chen, Qiantong Wang, Shaohui Liu, Shicheng Li, Shuhao Gu, Shuhuai Ren, Shuo Liu, Sirui Deng, Weiji Zhuang, Weiwei Lv, Wenyu Yang, Xin Zhang, Xing Yong, Xing Zhang, Xingchen Song, Xinzhe Xu, Xu Wang, Yihan Yan, Yu Tu, Yuanyuan Tian, Yudong Wang, Yue Yu, Zhenru Lin, Zhichao Song, and Zihao Yue. 2025. MiMo: Unlocking the Reasoning Potential of Language Model – From Pretraining to Posttraining. *arXiv preprint arXiv:2505.07608* (2025).

[71] Yuxing Xiang, Xue Li, Kun Qian, Yufan Yang, Diwen Zhu, Wenyuan Yu, Ennan Zhai, Xuanzhe Liu, Xin Jin, and Jingren Zhou. 2025. Aegaeon: Effective GPU pooling for concurrent LLM serving on the market. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 1030–1045.

[72] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic scaling on GPU clusters for deep learning. In *USENIX OSDI*.

[73] Jiarong Xing, Yifan Qiao, Simon Mo, Xingqi Cui, Gur-Eyal Sela, Yang Zhou, Joseph Gonzalez, and Ion Stoica. 2025. Towards Efficient and Practical GPU Multitasking in the Era of LLM. *arXiv preprint arXiv:2508.08448* (2025).

[74] 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. 2025. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388* (2025).

[75] David H. Yang, Mohammad Mohammadi Amiri, Tejaswini Pedapati, Subhajit Chaudhury, and Pin-Yu Chen. 2025. Sparse Gradient Compression for Fine-Tuning Large Language Models. (2025). arXiv:cs.LG/2502.00311 https://arxiv.org/abs/2502.00311

[76] John Yang, Kilian Lieret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025. Swe-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798* (2025).

[77] Chenxuan Yao, Feifan Liu, Yuchong Hu, Zhengyu Liu, Xinjue Zheng, and Wenxiang Zhou. 2025. LowDiff: Efficient Frequent Checkpointing via Low-Cost Differential for High-Performance Distributed Training Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25)*. Association for Computing Machinery, 1113–1126.

[78] Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, Zhongzhu Zhou, Michael Wyatt, Molly Smith, Lev Kurilenko, Heyang Qin, Masahiro Tanaka, Shuai Che, Shuaiwen Leon Song, and Yuxiong He. 2023. DeepSpeed-Chat: Easy, Fast and Affordable RLHF Training of ChatGPT-like Models at All Scales. *arXiv preprint arXiv:2308.01320* (2023).

[79] Minchen Yu, Rui Yang, Chaobo Jia, Zhaoyuan Su, Sheng Yao, Tingfeng Lan, Yuchen Yang, Yue Cheng, Wei Wang, Ao Wang, and Ruichuan Chen. 2025. lambdaScale: Enabling Fast Scaling for Serverless Large Language Model Inference. *arXiv preprint arXiv:2502.09922* (2025).

[80] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. 2025. DAPO: An Open-Source LLM Reinforcement Learning System at Scale. *arXiv preprint arXiv:2503.14476* (2025).

[81] Shan Yu, Jiarong Xing, Yifan Qiao, Mingyuan Ma, Yangmin Li, Yang Wang, Shuo Yang, Zhiqiang Xie, Shiyi Cao, Ke Bao, Ion Stoica, Harry Xu, and Ying Sheng. 2025. Prism: Unleashing GPU Sharing for Cost-Efficient Multi-LLM Serving. *arXiv preprint arXiv:2505.04021* (2025).

[82] Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. 2025. {BlitzScale}: Fast and Live Large Model Autoscaling with O (1) Host Caching. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 275–293.

[83] Ruiqi Zhang, Daman Arora, Song Mei, and Andrea Zanette. 2025. SPEED-RL: Faster Training of Reasoning Models via Online Curriculum Learning. *arXiv preprint arXiv:2506.09016* (2025).

[84] Yihao Zhao, Jiadun Chen, Peng Sun, Lei Li, Xuanzhe Liu, and Xin Jin. 2025. SeaLLM: Service-Aware and Latency-Optimized Resource Sharing for Large Language Model Inference. *arXiv preprint arXiv:2504.15720* (2025).

[85] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2022. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 428–440.

[86] Haizhong Zheng, Yang Zhou, Brian R. Bartoldson, Bhavya Kailkhura, Fan Lai, Jiawei Zhao, and Beidi Chen. 2025. Act Only When It Pays: Efficient Reinforcement Learning for LLM Reasoning via Selective Rollouts. *arXiv preprint arXiv:2506.02177* (2025).

[87] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 193–210.

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

[89] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, et al. 2025. Optimizing {RLHF} training for large language models with stage fusion. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 489–503.

Appendix

A Rollout Concurrency Profiling

ROSE profiles and caps rollout concurrency on dedicated rollout GPUs to avoid excessive KV cache (KVC) memory pressure. Figure 12 shows the rollout throughput of Qwen3-8B with a 32K context length under different per-GPU batch sizes. Throughput increases with concurrency up to a batch size of 16, after which it saturates. Increasing concurrency beyond this point increases rollout latency due to memory contention and KVC fragmentation, which in turn degrades effective throughput. Therefore, unless otherwise specified, we cap the maximum number of concurrent rollout requests per dedicated rollout GPU at 16.

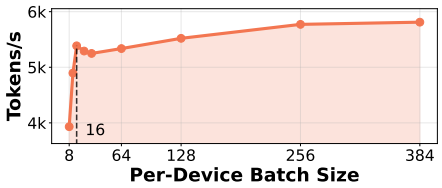


Figure 12. The system throughput with different per-device batch sizes. [Qwen3-8B/32K]

B Spot instance trace

We extract the spot-instance traces for the 8B model from Seg.B in Figure 8(a) and for the 32B model from Seg.B in Figure 9 of the RLBoost paper [69]. Figure 13 shows the variations in the number of preemptible and reserved GPUs over time. In the experiments, we used these traces to evaluate RLBoost+'s performance.

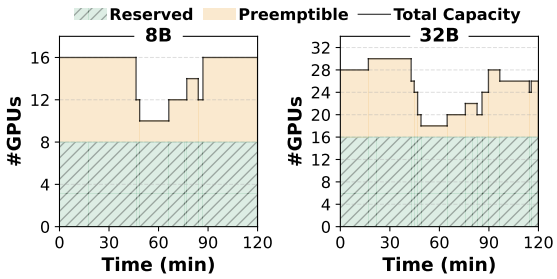


Figure 13. RLBoost trace used for 8B and 32B models.

C Sensitivity to caching lease

We set the lease time for prefix cache of rollout in the memory sharing policy. Table 4 evaluates how the prefix cache lease time affects rollout efficiency and serving SLOs for Qwen3-8B. The rollout time is largely insensitive to the lease time, suggesting that longer cache persistence provides limited additional benefit for rollouts in this setting. In contrast, lease time has a clear impact on serving tail latency: since the Dual-SLO admission controller is designed to enforce SLO thresholds rather than explicitly minimize tail latency, a longer lease can keep more rollout KVC resident and reduce GPU memory headroom for bursty serving traffic, inflating P99 latency. Consequently, a moderate lease time offers the best trade-off, retaining most prefix cache of rollouts benefits while provide scheduling flexibility to prevent rollout prefix-cache residency from degrading serving SLOs.

Table 4. [Co-Serve Executor]. The impact of prefix-cache lease time on rollout and serving SLO.

Lease (s)	Avg Rollout Time (s)	TTFT P99 (ms)	TPOT P99 (ms)
10	496.3	338.11	136.13
20	497.1	334.71	135.28
50	492.2	371.10	140.70
100	502.1	491.90	148.10

D Sensitivity to Serving Traffic.

Serving GPUs can be repurposed for rollouts because serving demand fluctuates over time, leaving transient compute and memory headroom. To understand how ROSE behaves as this headroom shrinks, we scale the serving request arrival density and measure both rollout efficiency (average rollout time) and serving QoS (TTFT/TPOT P99). Table 5 shows that higher serving density increases resource contention, which in turn prolongs rollouts and degrades serving tail latency.

Two additional observations stand out. First, TTFT is more sensitive to increasing load than TPOT, suggesting that contention primarily hurts the prefill, while per-token generation is comparatively less affected. Second, the larger model exhibits more stable rollout time as serving density increases, but its serving tail latencies still rise with load, indicating that compute/memory interference persists and must be managed by the co-serving executor. Overall, the results confirm the expected trade-off under cooperative elasticity: as serving load grows, ROSE gradually reduces effective rollout capacity while keeping serving performance degradation bounded rather than causing sharp SLO violations.

Table 5. [Co-Serve Executor]. The impact of serving traffic density on rollout and serving SLO.

Model	Density	Avg Rollout Time (s)	TTFT P99 (ms)	TPOT P99 (ms)
Qwen3-8B	1	496.3	338.1	136.1
	1.5	511.7	380.2	149.0
	2	569.9	459.1	150.1
Qwen3-32B	1	960.1	837.5	398.1
	1.5	977.7	870.2	421.3
	2	989.9	899.1	441.2

E Serving GPU Availability

The amount of underutilized capacity in the serving cluster also affects rollout performance. To quantify this effect, we measure the average rollout time of Qwen3-8B on FrozenLake using eight dedicated rollout GPUs over the first five RL steps, while varying the number of available serving GPUs. As shown in Figure 14, rollout time decreases as the serving

GPU quota increases. With an additional 16, 8 and 4 serving GPUs, ROSE reduces rollout time by 1.69×, 1.45×, and 1.26×, respectively. These results show that ROSE can effectively harvest underutilized serving resources to reduce the rollout overhead.

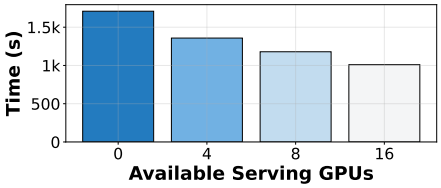


Figure 14. Sensitive Analysis of Serving GPU Availability.

F Timeline Breakdown of Weight Transfer

Figure 15 provides a detailed timeline breakdown of shard-aware and sparsity-aware weight transfer for Qwen3-32B. The top timeline illustrates shard-aware transfer: on the sender side, each training worker streams ~60 buckets (64 MB each); each bucket takes 0.2–0.4 s to push, for a total of 65 seconds. On the receiver side, serving workers pull the corresponding weight buckets from the relay and load them into GPU memory, taking 42 s in total. Overall, shard-awareness prevents redundant weight all-gather and avoids pulling and re-sharding a full replica on each serving worker, while increasing effective parallelism by utilizing multiple cross-cluster links.

The bottom timeline shows sparsity-aware transfer for the same model. Because weight deltas are highly sparse, each bucket is much smaller: per-bucket push and pull latency drops from hundreds of milliseconds to just a few milliseconds. The remaining per-bucket overhead is dominated by (de)sparsification, specifically Dense-to-Sparse (D2S) on the training side and Sparse-to-Dense (S2D) on the serving side, which stays sub-second. As a result, end-to-end transfer time decreases to 21 s, demonstrating the substantial benefit of exploiting weight delta sparsity for cross-cluster synchronization.

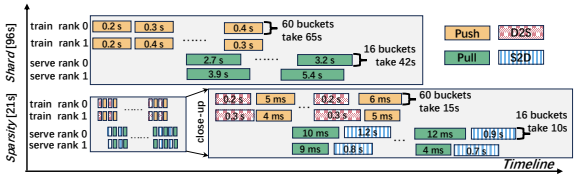


Figure 15. Timeline breakdown of shard-aware and sparsity-aware transfer for Qwen3-32B. D2S denotes the dense-to-sparse conversion, and S2D denotes the sparse-to-dense conversion.