ADAROLLOUT: ACCELERATING SYNCHRONOUS RLVR WITH ADAPTIVE PARALLELISM SWITCHING

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning with verifiable rewards (RLVR) relies on large-scale rollout generation, yet its performance is often bottlenecked by synchronous execution that is highly sensitive to long-tail decoding latency. Stochastic decoding produces sequences with widely varying lengths, where a few stragglers can stall batch completion and lead to poor accelerator utilization. This issue is exacerbated under static parallelism: as short sequences complete, the workload drastically shifts from high concurrency to long-tail dominated execution, rendering the initial parallel strategy inefficient. In this paper, we present ADAROLLOUT, a system for *state-preserving rollout re-parallelization* that dynamically reconfigures the execution topology during rollout without restarting in-flight requests. ADAROLLOUT profiles batch-completion throughput offline and derives a lightweight runtime policy that uses unfinished requests as a proxy for the remaining workload while accounting for reconfiguration cost. It further enables efficient topology reconfiguration through architecture-aware weight resharding, cross-topology KV-cache migration, and pre-instantiated runtime resources. By dynamically reallocating resources to long-tail sequences, ADAROLLOUT achieves substantial end-to-end training speedup over optimal static parallelism, while maintaining consistent training dynamics (reward RMSE < 0.1) and downstream performance (accuracy RMSE < 0.02).

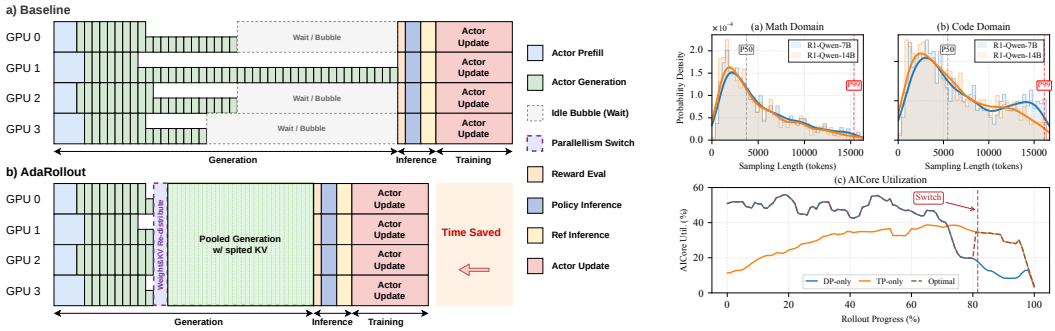
1 INTRODUCTION

Large language models (LLMs) have recently demonstrated remarkable reasoning capabilities across complex tasks such as mathematical problem solving (Hendrycks et al., 2021b; Shao et al., 2024) and code generation (Li et al., 2022). Reinforcement learning with verifiable rewards (RLVR) further enhances these capabilities by replacing learned reward models with deterministic, verifiable signals, enabling more reliable optimization for reasoning tasks (Wen et al., 2025; DeepSeek-AI et al., 2025; Yu et al., 2025; Liu et al., 2025a; Yue et al., 2025). However, realizing the full potential of RLVR in practice requires addressing significant system-level efficiency challenges.

The primary bottleneck of RLVR lies in the *rollout stage*, which exhibits extreme stochasticity and workload heterogeneity. As illustrated in Figure 1a, this bottleneck stems from two fundamental factors. First, autoregressive decoding is inherently stochastic, making output lengths and execution latency difficult to predict, which frequently causes load imbalance across distributed ranks and leads to severe accelerator bubbles. Second, rollout sequence lengths typically follow a pronounced long-tail distribution: while most sampled trajectories terminate quickly, a small fraction continue generating for significantly longer durations. These straggling requests block batch completion and force the majority of accelerators to remain idle, substantially reducing effective resource utilization (Fu et al., 2025; Piché et al., 2025; Zhou et al., 2025; Gao et al., 2025; Qin et al., 2025).

Despite this highly dynamic workload pattern, existing RL systems still primarily rely on static parallel strategies (Hu et al., 2025a; Sheng et al., 2024; Yao et al., 2023; Wang et al., 2025). However, this creates a fundamental mismatch between the evolving rollout workload and a fixed execution topology. For example, DP offers high throughput and suits the early stage with abundant requests, whereas TP provides lower latency and is more efficient in the long-tail dominated late stage (Topcu et al., 2026). This limitation is inherent: no single static parallel strategy can simultaneously optimize both the high-concurrency early stage and the long-tail dominated late stage of rollout. Therefore,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



(a) Accelerator timelines for Baseline (Static DP) vs. ADAROLLOUT (Dynamic).

(b) The inefficiency of static parallelism under long-tail workloads.

Figure 1: Motivation and mechanism of ADAROLLOUT. (a) **Solution:** Unlike static DP (top) which is blocked by stragglers, ADAROLLOUT (bottom) reclaims idle resources by reconfiguring execution topology mid-rollout. (b) **Problem:** Long-tail rollouts (P99/P50 up to 4.27× after removing truncated samples) cause accelerator idleness under static strategies, leaving a visible utilization gap.

effectively utilizing resources requires not only dynamic adaptation, but also the ability to reconfigure execution as workload characteristics evolve.

However, enabling such adaptive switching in practice is non-trivial due to two key challenges. First, determining **when to reconfigure** is difficult, as rollout behavior is highly stochastic and varies across datasets, model scales, and training iterations, requiring online adaptation. Second, enabling **how to reconfigure** efficiently is challenging, as reconfiguration involves stateful transformations that can introduce significant overhead. Therefore, an effective solution must jointly address both decision-making under uncertainty and efficient state transformation.

To address these challenges, we propose ADAROLLOUT, a system that enables *state-preserving rollout re-parallelization* for efficient RLVR training. ADAROLLOUT reframes rollout as dynamic reconfiguration over parallelization topologies, adapting execution to evolving workload conditions while preserving in-flight decoding states. To determine when to reconfigure, ADAROLLOUT profiles batch-completion throughput offline and derives a lightweight runtime policy that uses unfinished requests as a proxy for the remaining workload while accounting for reconfiguration cost. To reduce the overhead of reconfiguration, ADAROLLOUT introduces three key mechanisms within a unified state-preserving workflow: 1) *architecture-aware weight resharding*, which redistributes dense and MoE parameters according to their architecture-specific parallelization semantics; 2) *cross-topology KV-cache migration*, which directly remaps existing KV states across different layouts in principle without recomputing generated context; 3) *pre-instantiated runtime resources*, which pre-builds topology-specific execution graphs and communication resources and reuses them during runtime switching. By jointly optimizing *when* and *how* to reconfigure, ADAROLLOUT enables efficient adaptation to rollout workload dynamics while preserving decoding progress and reducing reconfiguration overhead.

We evaluate ADAROLLOUT across model scales using both rollout micro-benchmarks and end-to-end RL training. Results show that ADAROLLOUT consistently improves training efficiency and resource utilization, achieving substantial end-to-end training acceleration over static parallelism, and significantly higher device utilization. Further analysis shows that ADAROLLOUT maintains consistent training dynamics (reward RMSE < 0.1) and final performance (accuracy RMSE < 0.02), validating ADAROLLOUT as a practical solution to the rollout bottleneck in large-scale RL systems.

2 PRELIMINARY EXPERIMENTS

To motivate adaptive parallelism switching, we conduct preliminary experiments around two observations: (1) RL rollout exhibits pronounced long-tail generation lengths across domains and model scales; and (2) static parallel strategies cannot simultaneously optimize the high-concurrency and long-tail phases of a rollout step.

2.1 THE PREVALENCE OF LONG-TAIL EFFECTS

To systematically quantify the long-tail effect, which has been observed in prior work but not thoroughly characterized in RL rollout workloads (Fu et al., 2025; Piché et al., 2025; Zhou et al., 2025; Gao et al., 2025; Qin et al., 2025), we conduct experiments across different model scales (DeepSeek-R1-Distill-Qwen-7B, DeepSeek-R1-Distill-Qwen-14B) (DeepSeek-AI et al., 2025) and task types (DAPO-Math-17k (Yu et al., 2025), APPS (Hendrycks et al., 2021a)). In RLVR, each training iteration requires the actor to sample responses for a batch of prompts before reward computation and policy update. Because autoregressive generation is stochastic, different samples may terminate after substantially different numbers of tokens. Moreover, on-policy algorithms such as GRPO (Shao et al., 2024) impose a synchronous update barrier, requiring each step to wait until all sampled responses finish decoding. As a result, a small number of unusually long sequences can delay the entire batch and amplify workload imbalance.

Figure 1b (a,b) demonstrates the prevalence and severity of the long-tail effect. Math rollouts have P99/P50 ratios of 3.88× for the 7B model and 4.27× for the 14B model, while code rollouts are similarly heavy-tailed, with ratios of 2.74× and 3.08×, respectively. In absolute terms, P99 lengths remain close to the truncation boundary: 15.3K tokens for math and 16.0–16.1K tokens for code, while medians are only 3.6–4.0K and 5.2–5.9K tokens. This indicates that rollout is a dynamic workload: the active-request count decreases over time and eventually becomes dominated by stragglers.

2.2 LIMITATIONS OF STATIC PARALLEL STRATEGIES

Despite this highly dynamic workload pattern, existing RL systems primarily rely on static parallel strategies. Data Parallelism (DP) replicates the model across accelerators and maximizes throughput during high concurrency, but cannot reduce per-token latency for the few remaining long requests. Tensor Parallelism (TP) partitions model computation and KV cache across accelerators, reducing per-request latency but incurring communication overhead that penalizes throughput when concurrency is high. Therefore, no single fixed layout is optimal throughout an entire rollout.

Figure 1b(c) shows the accelerator utilization under DP-only, TP-only, and the optimal static-oracle envelope over rollout progress. DP-only achieves around 50% utilization in the early phase when concurrency is abundant, but utilization drops below 20% after roughly 80% rollout progress as completed sequences leave only a small set of long-tail requests active. TP-only is less efficient early due to communication overhead, yet becomes more attractive in the late phase when only a few stragglers remain. The optimal curve switches from the throughput-oriented DP regime to the latency-oriented TP regime once the remaining workload is dominated by stragglers, improving late-phase utilization from roughly 18% to 35% near the switching point. This gap represents the potential benefit of dynamic switching and motivates our key insight: *the rollout parallel strategy should adapt online to the evolving workload state rather than remain fixed for the whole step*.

3 ADAROLLOUT

In this section, we present ADAROLLOUT, a system for *state-preserving rollout re-parallelization*. ADAROLLOUT dynamically reconfigures the parallelization topology during rollout while preserving the logical states of unfinished requests, enabling execution to adapt to evolving workload conditions without restarting generation. As illustrated in Figure 2, ADAROLLOUT addresses two coupled challenges: *when to reconfigure*, through an offline-profiled, lightweight runtime policy that uses the number of unfinished requests as a proxy for the remaining workload (§3.1), and *how to reconfigure*, through a state-preserving reconfiguration workflow and architecture-aware state transformation that efficiently remap model weights and decoding states across topologies (§3.2, §3.3).

3.1 ONLINE TOPOLOGY SWITCHING POLICY

Offline Throughput Characterization. ADAROLLOUT first characterizes the throughput of candidate topologies offline to identify when reconfiguration can improve rollout efficiency. For each topology $s \in \mathcal{S}$, concurrency level n, and output length L, we measure the batch-completion through-

put $\Theta(s, n, L)$, defined as the total number of generated tokens divided by the wall-clock time required to complete the corresponding batch. This profiling produces a throughput surface over topology, concurrency, and output length, from which the preferred topology at workload point (n, L) is determined by

$$s^*(n, L) = \arg\max_{s \in \mathcal{S}} \Theta(s, n, L)$$

For the two representative configurations used by AdaRollout, a data-parallel-dominant topology s_{DP} and a tensor-parallel-dominant topology s_{TP}, the crossover length $L^*(n)$ at each concurrency level satisfies

$$\Theta(s_{\mathrm{TP}}, n, L^*(n)) = \Theta(s_{\mathrm{DP}}, n, L^*(n)).$$

This provides an oracle transition boundary when the remaining output length is known.

Runtime Approximation under Unknown Remaining Length. The remaining output length is not observable during rollout, making direct application of the (n, L) boundary impractical without an additional workload predictor. AdaRollout instead uses the number of unfinished requests n_t as a lightweight runtime proxy: as requests complete, n_t decreases and generally indicates that the rollout is entering its tail. We therefore project the offline workload-dependent boundary onto the observable concurrency dimension and derive an empirical threshold n^* from the profiled workload:

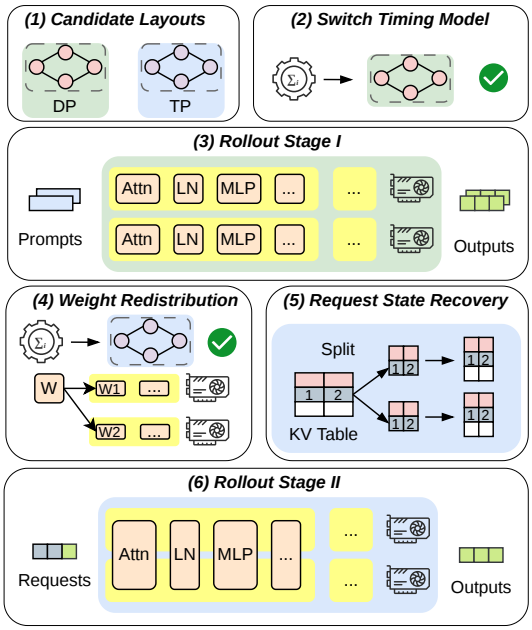


Figure 2: Overview of the AdaRollout workflow: layout selection, online switch decision, weight redistribution, request-state recovery, and resumed rollout under the new topology.

$$n^* = \min\{n : s_{\mathrm{TP}} \text{ is preferred in the profiled tail workload}\}.$$

At runtime, AdaRollout triggers the transition when $n_t \le n^*$, provided that the target topology differs from the current topology. This avoids online prediction of the remaining output length and reduces the switching decision to a negligible-overhead comparison.

Switching with Migration Cost. A topology with higher steady-state throughput is not necessarily beneficial if the one-time reconfiguration cost is too large. Let $\Delta_{\mathrm{oh}}(s \to s')$ denote the measured switching overhead, including weight redistribution, KV-cache migration, and synchronization. For a workload with remaining output length L and approximately constant throughput, the minimum workload required to amortize this cost is

$$L_{\mathrm{amort}} = \Delta_{\mathrm{oh}}(s \to s') \frac{\Theta(s', n, L)\Theta(s, n, L)}{\Theta(s', n, L) - \Theta(s, n, L)}, \qquad \Theta(s', n, L) > \Theta(s, n, L).$$

AdaRollout incorporates this amortization cost when constructing the offline transition boundary rather than evaluating L_{amort} online. Consequently, the runtime policy remains the simple concurrency-threshold rule, while the selected threshold accounts for both the throughput advantage of the target topology and the cost of reconfiguration.

3.2 State-Preserving Reconfiguration

Once a topology transition is triggered, AdaRollout executes a coordinated state-preserving reconfiguration protocol. The key design principle is to separate the *logical decoding state* of a request from its topology-dependent physical representation. This separation allows model parameters, communication groups, and KV-cache storage to be reorganized independently while retaining the logical state required for decoding. As illustrated in Algorithm 1, the transition consists of three phases: *state snapshot*, *topology transformation*, and *state restoration*.

Algorithm 1 ADAROLLOUT State-Preserving Topology Reconfiguration

Require: Current topology s, target topology s', engine E, active requests $\mathcal{M}$
Ensure: E resumes under s' with preserved logical request states
Phase 1: Snapshot
Pause scheduling and drain in-flight decoding operations
for each request $i \in \mathcal{M}$ **do**
  Snapshot logical block mapping $\mathcal{B}_i$
  Snapshot decoding metadata $\mathcal{D}_i$
end for
Phase 2: Topology Transformation
Transform model parameters from s to s'
Activate communication groups associated with s'
Transform physical request states to the target layout
Phase 3: Restoration
for each request $i \in \mathcal{M}$ **do**
  Restore $\mathcal{B}_i$ and $\mathcal{D}_i$ under s'
end for
Synchronize all target ranks
Resume scheduling and decoding under s'

Phase 1: State Snapshot. Before changing the execution topology, ADAROLLOUT pauses scheduling of the active requests and records their logical decoding states. For each request i, we denote this state by $q_i = (\mathcal{B}_i, \mathcal{D}_i)$, where $\mathcal{B}_i$ represents the logical mapping of the request to its paged KV-cache blocks and $\mathcal{D}_i$ denotes the remaining decoding metadata maintained by the inference engine. The $\mathcal{D}_i$ encapsulates engine-specific state, including token-history information, decoding position, sampling configuration and state, and other metadata required to resume decoding.

Phase 2: Topology Transformation. After the logical request states have been isolated, ADAROLLOUT transforms the execution environment from topology s to s'. This transformation includes: (1) Model parameters are redistributed according to the target parallelization topology. (2) The communication topology is switched to the pre-instantiated communication groups associated with s'. The physical representation of the request states may also change during this phase, but the logical state captured in Phase 1 remains unchanged. The detailed parameter and KV-cache transformation algorithms are described in Section 3.3.

Phase 3: State Restoration. Finally, ADAROLLOUT reconstructs the active requests under the target topology. For each request, its logical block mapping and decoding metadata are re-associated with the target memory layout, while its KV-cache contents are transformed into the target topology's physical representation. After all state transformations and required synchronization have completed, the scheduler atomically resumes the suspended requests under s'. Because the logical decoding state is preserved across the transition, the target engine can continue decoding from the same request state rather than restarting generation.

3.3 ARCHITECTURE-AWARE STATE TRANSFORMATION

Architecture-Aware Weight Resharding. Topology reconfiguration requires redistributing model parameters according to the target parallelization layout while preserving their architectural semantics. ADAROLLOUT performs resharding according to the logical structure of each parameter rather than treating every parameter tensor as a homogeneous array. (1) For a fused attention parameter, we first identify the constituent projections and independently map them to the target tensor-parallel layout:

$$W_{\text{attn}} = \text{Concat}(W_Q, W_K, W_V),\ W'_{\text{attn}} = \text{Concat}\Big(\mathcal{P}_Q^{s\to s'}(W_Q), \mathcal{P}_K^{s\to s'}(W_K), \mathcal{P}_V^{s\to s'}(W_V)\Big),$$

where $\mathcal{P}_Q$, $\mathcal{P}_K$, and $\mathcal{P}_V$ apply the head-level partitioning rules of the corresponding projections. This sub-projection-aware transformation is necessary for GQA, where W_Q and $W_{K/V}$ have different numbers of attention heads and therefore cannot in general be correctly redistributed by directly slicing the fused tensor. (2) For MoE models, we apply the same parameter-level transformation

to shared parameters, while expert parameters are additionally remapped according to the target expert-parallel ownership $W^{s'}_{e,r'} = \mathcal{P}^{s\to s'}_e(W_e),\ r' = \pi_{s'}(e)$, where $\pi_{s'}$ specifies the target rank responsible for expert e. The corresponding expert-ownership metadata is updated together with the weights, ensuring that the target routing and parameter placement remain consistent.

Cross-Topology KV-Cache Migration. KV caches are stored as paged blocks and physically distributed according to request ownership and the tensor-parallel layout. During a topology transition, AdaRollout directly maps the generated context from the source to the target KV layout without recomputation. Let $C^{(l),s}_{i,r}$ denote the KV cache of request i at layer l stored on rank r under topology s, and let $\beta^s_{i,r}(k)$ denote the physical block storing the k-th logical block of request i. AdaRollout transfers only the KV-head fragments required by the target rank and places them at the corresponding target blocks:

$$C^{(l),s'}_{i,r'}\left[\beta^{s'}_{i,r'}(k),\ \mathcal{H}^s_r \cap \mathcal{H}^{s'}_{r'}\right] \leftarrow C^{(l),s}_{i,r}\left[\beta^s_{i,r}(k),\ \mathcal{H}^s_r \cap \mathcal{H}^{s'}_{r'}\right],$$

where k ranges over the valid logical blocks of request i, and $\mathcal{H}^s_r$ denotes the KV-heads owned by rank r. Only the required source fragments are transferred within the source DP owner's TP group. For replicated KV-heads under GQA, a deterministic source representative is selected to avoid duplicate transfers. Target blocks are allocated before migration and the received fragments are directly placed at their target locations, avoiding redundant cache replication and recomputation.

Pre-instantiated Runtime Resources. Topology switching incurs additional overhead from constructing topology-specific execution graphs and their associated communication resources. AdaRollout addresses this overhead in two stages. (1) **Initialization-time pre-instantiation** builds the execution graph and communication groups required by each candidate topology in advance, with the resulting graph state stored separately for each layout. (2) **Runtime graph reuse** then directly reuses the pre-instantiated graph when a topology becomes active, avoiding graph construction and communication-group setup on the critical switching path. For MoE execution, we constrain the graph input boundary to ensure reuse across valid routing patterns while keeping graph states isolated across layouts.

4 Experiments

We evaluate AdaRollout in both rollout micro-benchmarks and end-to-end RL training against open-source frameworks across models of different scales. We then conduct system ablations to examine the contribution of each component. Finally, we analyze training consistency and switching overhead to understand the behavior and cost of dynamic execution.

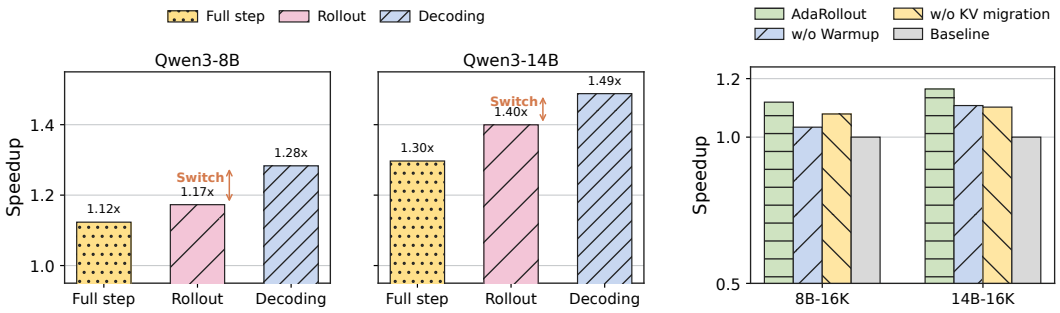
4.1 Experimental Setup

We conduct experiments on a cluster of Ascend 910B NPUs (16 NPUs per node, 64GB HBM each) based on the vLLM and verl frameworks. We evaluate models across different scales, including Qwen3-8B and Qwen3-14B (Yang et al., 2025). All experiments use the DAPO-Math-17k dataset (Yu et al., 2025) and the GRPO algorithm (Shao et al., 2024), with a group size of 8 and a global batch size of 256. Other training configurations are provided in Appendix A. We compare AdaRollout against the optimal static parallel execution strategy, where the parallel topology remains fixed throughout each rollout step.

4.2 Main Results

4.2.1 Rollout Micro-benchmarks

We first isolate the impact of state-preserving topology reconfiguration through controlled rollout micro-benchmarks. We measure the wall-clock time to complete a fixed rollout workload across Qwen3-8B and Qwen3-14B, with generation lengths of 8K and 16K tokens and batch sizes of 64, 128, and 256.



(a) RLVR full-step, rollout, and decoding speedup.

(b) Component ablation speedup.

Figure 4: RLVR and ablation speedup over the static baseline on the same dataset as the micro-benchmark with global batch size of 256. AdaRollout accelerates decoding, rollout, and full-step execution, while the ablation confirms that topology warm-up and KV-cache migration are both needed to retain the speedup.

We compare AdaRollout with the optimal static parallelism (DP-only), with Figure 3 summarizing the resulting speedups. AdaRollout consistently improves rollout completion time across the evaluated configurations. For Qwen3-14B with 8K generation and batch size 256, AdaRollout achieves up to 1.33× speedup over optimal static parallel. For 16K generation, the speedup remains between 1.17× and 1.20× across the evaluated batch sizes. These results show that dynamically adapting the topology can effectively reduce rollout time across different model scales and workload configurations, despite the overhead of state migration.

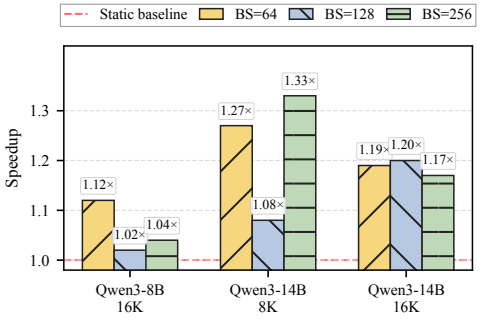


Figure 3: Rollout microbenchmark speedup over optimal static parallelism.

4.2.2 End-to-End RL Training

We next evaluate whether the rollout-level improvement translates to end-to-end RLVR training. We use the same dataset and global batch size as the micro-benchmarks to ensure a consistent workload. Figure 4a reports speedup for the full RLVR step, the rollout phase, and the decoding portion. The decoding measurement isolates steady-state decoding performance after excluding cold-start overhead, whereas the rollout measurement includes the runtime reconfiguration overhead. On Qwen3-8B, AdaRollout reduces decoding time from 810.26s to 631.45s, rollout time from 810.26s to 690.83s, and full-step time from 1088.58s to 969.15s, corresponding to 1.28×, 1.17×, and 1.12× speedups, respectively. On Qwen3-14B, the corresponding times decrease from 1641.78s to 1103.25s for decoding, from 1641.78s to 1173.00s for rollout, and from 2047.76s to 1578.98s for the full step, yielding 1.49×, 1.40×, and 1.30× speedups. The larger gain on Qwen3-14B indicates that the benefit of topology adaptation becomes more pronounced when decoding constitutes a larger fraction of the execution time, allowing the reconfiguration overhead to be amortized over a longer workload.

4.3 Ablation Study

We further isolate the contribution of the key reconfiguration mechanisms using the same RLVR math workload with global batch size 256. Figure 4b reports speedup over static DP for Qwen3-8B and Qwen3-14B under 16K generation. The full system achieves 1.12× and 1.16× speedup on Qwen3-8B and Qwen3-14B, respectively. Removing topology warm-up reduces the speedup to 1.03× and 1.11×, demonstrating that pre-instantiated runtime resources substantially reduce the cost of entering the target topology. Disabling KV-cache migration further reduces the speedup to 1.08× and 1.10×, indicating that preserving and directly transferring the existing decoding state is important for realizing the benefit of in-flight reconfiguration. Overall, the ablation results show that topology

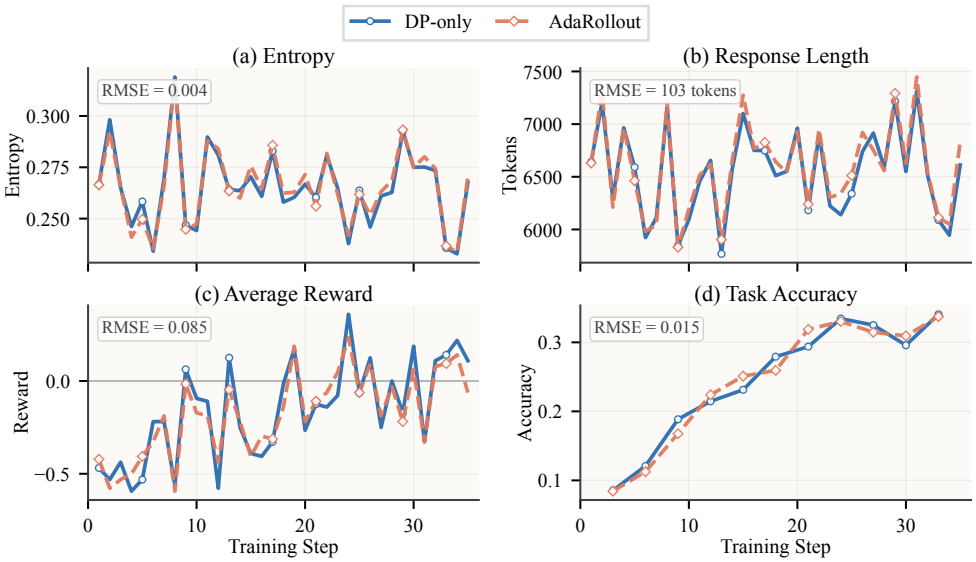


Figure 5: Training dynamics comparison on the end-to-end RLVR run. ADAROLLOUT closely tracks the static baseline across entropy, response length, average reward, and task accuracy. Each panel reports the RMSE between matched training steps.

switching alone is insufficient to obtain the observed gains. The effectiveness of ADAROLLOUT depends on both reducing the cost of reconfiguration and preserving the in-flight rollout state, so that the latency saved by adapting to the workload can amortize the transition overhead.

4.4 IN-DEPTH ANALYSIS

4.4.1 TRAINING QUALITY

A key question is whether dynamic topology reconfiguration alters the RLVR training trajectory or final model quality. Figure 5 compares ADAROLLOUT with the static baseline over matched RLVR training steps. The two configurations exhibit closely aligned training dynamics across entropy, response length, average reward, and task accuracy, with RMSEs of 0.004, 103 tokens, 0.085, and 0.015, respectively. A two-sided paired t-test over matched task-accuracy evaluation points does not detect a statistically significant difference ($p > 0.05$). These results indicate that dynamic reconfiguration preserves the decoding state sufficiently to maintain consistent training dynamics and downstream task performance. In particular, ADAROLLOUT migrates the KV cache and decoding metadata across topology changes rather than regenerating completed context, allowing rollout to continue from the same logical request states.

Table 1: Switching overhead breakdown (DP→TP→DP cycle).

Phase	**Time (s)**
DP→TP transition:	
Control-plane	0.2
Weight/layout	1.5
KV migration	3.2
Engine reinit	1.0
Graph restore	2.8
Sync/stats	0.6
Subtotal	*8.3*
Release	0.01
TP→DP reset	8.8
Total	**17.1**

4.4.2 SWITCHING OVERHEAD

We further profile the cost of a complete DP→TP→DP switching cycle to identify the dominant components of reconfiguration. Table 1 breaks down the measured latency. For the DP→TP transition, KV-cache migration is the largest component (3.2s), followed by graph restoration (2.8s) and weight/layout transformation (1.5s), while control-plane operations account for only 0.2s after runtime resources are pre-instantiated. The complete cycle takes 17.1s, including 8.3s for DP→TP and 8.8s for the subsequent TP→DP transition. This overhead is amortized over the subsequent

rollout execution rather than incurred per generated token. For a representative 8k-token rollout that takes approximately 350s, the complete switching cycle accounts for 4.9% of the rollout time, while ADAROLLOUT achieves a 21.1% speedup in the corresponding workload. These results show that although state migration remains a non-negligible component of reconfiguration cost, the overhead can be effectively amortized when switching is applied to sufficiently long-tail workloads.

5 RELATED WORK

RL Systems for LLM Post-Training. Modern RLHF and RLVR systems scale post-training by combining optimization algorithms with distributed training and inference infrastructure. RLVR improves reasoning through verifiable feedback and has been explored with increasingly effective optimization recipes (Wen et al., 2025; DeepSeek-AI et al., 2025; Yu et al., 2025; Liu et al., 2025a; Yue et al., 2025) and broader verifiable tasks (Liu et al., 2025b; Wu et al., 2025; Chen et al., 2025). Systems such as DeepSpeed-Chat (Yao et al., 2023), OpenRLHF (Hu et al., 2025a), HybridFlow (Sheng et al., 2024), and ROLL (Wang et al., 2025) improve end-to-end efficiency through distributed orchestration, resource placement, sharding, and pipeline scheduling. These systems build on established parallel and inference substrates, including tensor parallelism (Shoeybi et al., 2019), parameter and optimizer sharding (Rajbhandari et al., 2020; Zhao et al., 2023), continuous batching (Yu et al., 2022), and efficient KV-cache management (Kwon et al., 2023; Aminabadi et al., 2022; Zheng et al., 2024). Recent systems also explore resource reallocation and phase-level reconfiguration across RL or serving stages (Mei et al., 2025; Su et al., 2025; Zhong et al., 2025). However, these systems generally keep the parallel execution strategy fixed during an ongoing rollout. In contrast, ADAROLLOUT enables state-preserving rollout re-parallelization, dynamically reconfiguring the parallel topology within a synchronous rollout as the active workload evolves. A lightweight offline-profiled policy determines when to reconfigure, while state-preserving transformation enables the topology change without restarting in-flight requests.

Asynchronous RL and Rollout Efficiency. The stochastic and long-tailed nature of LLM generation has motivated several approaches to reducing rollout inefficiency. Asynchronous RL systems such as AReaL (Fu et al., 2025) and Laminar (Sheng et al., 2025) overlap rollout and training across different policy versions, reducing accelerator idle time by relaxing synchronous execution. Other approaches mitigate rollout stragglers through partial rollout, workload prediction, or packing. PipelineRL (Piché et al., 2025), APRIL (Zhou et al., 2025), RollPacker (Gao et al., 2025), and Seer (Qin et al., 2025) reduce tail effects by truncating, repacking, deferring, or predicting rollout workloads. Speculative decoding methods (Hu et al., 2025b; Shao et al., 2025; Chang et al., 2025) instead reduce generation cost by modifying the decoding procedure. These approaches improve rollout efficiency by changing synchronization, rollout composition, or decoding. In contrast, ADAROLLOUT retains synchronous RLVR execution and the original rollout procedure while adapting the parallel topology to the evolving workload. By combining workload-aware topology reconfiguration with state-preserving transformation of model and decoding states, ADAROLLOUT reduces rollout tail inefficiency while maintaining consistent training dynamics.

6 CONCLUSION

In this work, we presented ADAROLLOUT, a system for state-preserving rollout re-parallelization in RL training. ADAROLLOUT dynamically reconfigures the parallel topology as rollout concurrency evolves, using an offline-profiled workload-aware policy to determine when reconfiguration can amortize its cost. Combined with state-preserving transformation of model and decoding states, this enables topology changes without restarting in-flight decoding. Across diverse RLVR workloads, ADAROLLOUT improves rollout and end-to-end training efficiency while maintaining consistent training dynamics and downstream task performance. These results demonstrate that adapting the parallel topology to the evolving rollout workload provides an effective approach to mitigating long-tail inefficiency while preserving synchronous execution.

AI USE STATEMENT

In this work, we used generative AI tools to provide feedback on research experiments. We have not used generative AI tools to develop theoretical models or conceptual frameworks, or support qualitative and thematic data analysis, and the rest of the required disclosure tasks are not applicable to this work. Additionally, we used generative AI tools to propose a title or keywords for a research paper. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale. *arXiv preprint arXiv:2207.00032*, 2022.

Chi-Chih Chang, Siqi Zhu, Zhichen Zeng, Haibin Lin, Xin Liu, Jiaxuan You, Mohamed S. Abdelfattah, Ziheng Jiang, and Xuehai Qian. SRT: Accelerating reinforcement learning via speculative rollout with tree-structured cache. In *The First Workshop on Efficient Reasoning at NeurIPS*, 2025. URL https://openreview.net/forum?id=UpEKKJStAY.

Yukang Chen, Wei Huang, Baifeng Shi, Qinghao Hu, Hanrong Ye, Ligeng Zhu, Zhijian Liu, Pavlo Molchanov, Jan Kautz, Xiaojuan Qi, Sifei Liu, Hongxu Yin, Yao Lu, and Song Han. Scaling rl to long videos. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, pp. 172842–172870. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/fc2865a2fdd5fd401050d37ee4946850-Paper-Conference.pdf.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

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. Areal: A large-scale asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025.

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. Rollpacker: Mitigating long-tail rollouts for fast, synchronous rl post-training. *arXiv preprint arXiv:2509.21009*, 2025.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021a. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021b.

Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, Wenkai Fang, Xianyu, Yu Cao, Haotian Xu, and Yiming Liu. OpenRLHF: A ray-based easy-to-use, scalable and high-performance RLHF framework. In Ivan Habernal, Peter Schulam, and Jörg Tiedemann (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 656–666, Suzhou, China, November 2025a. Association for Computational Linguistics. ISBN 979-8-89176-334-0. doi: 10.18653/v1/2025.emnlp-demos.48. URL https://aclanthology.org/2025.emnlp-demos.48/.

Qinghao Hu, Shang Yang, Junxian Guo, Xiaozhe Yao, Yujun Lin, Yuxian Gu, Han Cai, Chuang Gan, Ana Klimovic, and Song Han. Taming the long-tail: Efficient reasoning rl training with adaptive drafter. *arXiv preprint arXiv:2511.16665*, 2025b.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Remi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022. ISSN 1095-9203. doi: 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158.

Mingjie Liu, Shizhe Diao, Ximing Lu, Jian Hu, Xin Dong, Yejin Choi, Jan Kautz, and Yi Dong. ProRL: Prolonged reinforcement learning expands reasoning boundaries in large language models. *arXiv preprint arXiv:2505.24864*, 2025a.

Xiaoyuan Liu, Tian Liang, Zhiwei He, Jiahao Xu, Wenxuan Wang, Pinjia He, Zhaopeng Tu, Haitao Mi, and Dong Yu. Trust, but verify: A self-verification approach to reinforcement learning with verifiable rewards. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, pp. 130475–130501. Curran Associates, Inc., 2025b. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/bd3d106d8e9789ce4fd40e1edafe71d5-Paper-Conference.pdf.

Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. Real: Efficient rlhf training of large language models with parameter reallocation. In M. Zaharia,

G. Joshi, and Y. Lin (eds.), *Proceedings of Machine Learning and Systems*, volume 7. MLSys, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/file/3b3889d313ba9476c12c2d77ea66b24f-Paper-Conference.pdf.

Alexandre Piché, Ehsan Kamalloo, Rafael Pardinas, Xiaoyin Chen, and Dzmitry Bahdanau. PipelineRL: Faster on-policy reinforcement learning for long sequence generation. *arXiv preprint arXiv:2509.19128*, 2025.

Ruoyu Qin, Weiran He, Weixiao Huang, Yangkun Zhang, Yikai Zhao, Bo Pang, Xinran Xu, Yingdi Shan, Yongwei Wu, and Mingxing Zhang. Seer: Online context learning for fast synchronous llm reinforcement learning. *arXiv preprint arXiv:2511.14617*, 2025.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, November 2020. doi: 10.1109/sc41405.2020.00024. URL http://dx.doi.org/10.1109/SC41405.2020.00024.

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. Beat the long tail: Distribution-aware speculative decoding for rl training. *arXiv preprint arXiv:2511.13841*, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv:2409.19256*, 2024.

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. Laminar: A scalable asynchronous rl post-training framework. *arXiv preprint arXiv:2510.12633*, 2025.

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

Qidong Su, Wei Zhao, Xin Li, Muralidhar Andoorveedu, Chenhao Jiang, Zhanda Zhu, Kevin Song, Christina Giannoula, and Gennady Pekhimenko. Seesaw: High-throughput llm inference via model re-sharding. In M. Zaharia, G. Joshi, and Y. Lin (eds.), *Proceedings of Machine Learning and Systems*, volume 7. MLSys, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/file/cbc4ab80cd77aa0eb87da062fbcddb46-Paper-Conference.pdf.

Burak Topcu, Musa Oguzhan Cim, Poovaiah Palangappa, Meena Arunachalam, and Mahmut Taylan Kandemir. Parallelization strategies for dense llm deployment: Navigating through application-specific tradeoffs and bottlenecks. *arXiv preprint arXiv:2603.05692*, 2026.

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. Reinforcement learning optimization for large-scale learning: An efficient and user-friendly scaling library. *arXiv preprint arXiv:2506.06122*, 2025.

Xumeng Wen, Zihan Liu, Shun Zheng, Shengyu Ye, Zhirong Wu, Yang Wang, Zhijian Xu, Xiao Liang, Junjie Li, Ziming Miao, Jiang Bian, and Mao Yang. Reinforcement learning with verifiable rewards implicitly incentivizes correct reasoning in base llms. *arXiv preprint arXiv:2506.14245*, 2025.

Jialong Wu, Shaofeng Yin, Ningya Feng, and Mingsheng Long. Rlvr-world: Training world models with reinforcement learning. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, pp. 125312–125350. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/b63a24a1832bd14fa945c71f535c0095-Paper-Conference.pdf.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

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. Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales. *arXiv preprint arXiv:2308.01320*, 2023.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, Carlsbad, CA, July 2022. USENIX Association. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, juncai liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru 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, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, pp. 113222–113244. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/a4277440d50f1f15d2cb4c14f7e0c0d2-Paper-Conference.pdf.

Yu Yue, Yufeng Yuan, Qiying Yu, Xiaochen Zuo, Ruofei Zhu, Wenyuan Xu, Jiaze Chen, Chengyi Wang, Tiantian Fan, Zhengyin Du, Xiangpeng Wei, Gaohong Liu, Juncai Liu, Lingjun Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, and Ru Zhang. VAPO: Efficient and reliable reinforcement learning for advanced reasoning tasks. *arXiv preprint arXiv:2504.05118*, 2025.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proceedings of the VLDB Endowment*, 16(12):3848–3860, August 2023. ISSN 2150-8097. doi: 10.14778/3611540.3611569. URL http://dx.doi.org/10.14778/3611540.3611569.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/724be4472168f31ba1c9ac630f15dec8-Paper-Conference.pdf.

Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, and Xin Jin. Optimizing RLHF training for large language models with stage fusion. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pp. 489–503, Philadelphia, PA, April 2025. USENIX Association. ISBN 978-1-939133-46-5. URL https://www.usenix.org/conference/nsdi25/presentation/zhong.

Yuzhen Zhou, Jiajun Li, Yusheng Su, Gowtham Ramesh, Zilin Zhu, Xiang Long, Chenyang Zhao, Jin Pan, Xiaodong Yu, Ze Wang, Kangrui Du, Jialian Wu, Ximeng Sun, Jiang Liu, Qiaolin Yu, Hao Chen, Zicheng Liu, and Emad Barsoum. April: Active partial rollouts in reinforcement learning to tame long-tail generation. *arXiv preprint arXiv:2509.18521*, 2025.

A Implementation Details

The training configurations and hyperparameters are in Table 2.

Table 2: Training and system configurations used in our experiments.

Training	
Algorithm	GRPO
Learning rate	1×10^{-6}
Rollout batch size	32
PPO mini-batch size	32
Temperature	0.9
Sequence Length	
Maximum prompt length	2,048
Maximum response length	16,384
Maximum sequence length	18,432
Overlong prompt filtering	Enabled
Model and Rollout	
Rollout engine	vLLM
Rollout mode	Synchronous
Rollout samples per prompt	8
Rollout GPU memory utilization	0.8
Rollout tensor parallelism	1
Parallel Switching	
Initial parallelism	TP=1, DP=8
Target parallelism	TP=4, DP=2
Unfinished-request threshold	64
KV-cache migration	Enabled
Optimization and Memory	
Gradient checkpointing	Enabled
Parameter offloading	Enabled
Optimizer offloading	Enabled
KL loss	Enabled
KL loss coefficient	0.001
KL loss type	Low-variance KL
Clip ratio	0.2
High clip ratio	0.28