USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

RollPacker: Taming Long-Tail Rollouts for RL Post-Training with Tail Batching

Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, and Lunxi Cao, *Hong Kong University of Science and Technology;* Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, Jiamang Wang, Lin Qu, and Bo Zheng, *Alibaba Group;* Wei Wang, *Hong Kong University of Science and Technology*

https://www.usenix.org/conference/nsdi26/presentation/gao-wei

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of Science and Technology

RollPacker: Taming Long-Tail Rollouts for RL Post-Training with Tail Batching

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[‡], Wei Wang[†]

[†]HKUST [‡]Alibaba Group

Abstract

Reinforcement Learning (RL) is a pivotal post-training technique for enhancing the reasoning capabilities of Large Language Models (LLMs). However, synchronous RL post-training frequently suffers from significant GPU underutilization—often referred to as pipeline "bubbles"—caused by imbalanced response lengths within rollout steps. Many RL systems attempt to alleviate this problem by relaxing synchronization, but this can compromise training accuracy. In this paper, we introduce *tail batching*, a novel rollout scheduling strategy for synchronous RL. Tail batching systematically consolidates prompts leading to long-tail responses into a few designated "*long rounds*", ensuring that the majority of rollout steps ("*short rounds*") contain only balanced, short responses. By strategically reordering execution, this approach dramatically reduces GPU idle time and accelerates RL training without sacrificing on-policy accuracy. We present RollPacker, a system that fully harnesses the benefits of tail batching through holistic optimizations across all three RL stages: elastic parallelism adaptation for rollout, dynamic resource allocation and scheduling for reward, and stream-based training. Cluster deployment on up to 128 H800 GPUs demonstrates that RollPacker achieves an end-to-end training speedup of 2.03× to 2.56× over veRL [34], and up to 2.24× speedup compared to RLHFuse [55] across the Qwen2.5 family of LLMs. The code is available at https://github.com/Farrrrland/RollPacker.

1 Introduction

Advanced Large Language Models (LLMs) [5, 6, 28, 31] critically rely on Reinforcement Learning (RL) post-training to enhance reasoning capabilities in complex tasks, such as mathematics [19], code generation [27], and tool use [7, 29]. The standard RL post-training workflow for LLM reasoning models comprises repeated cycles across three stages [6, 33]: *rollout*, *reward*, and *training*. In the rollout stage, the actor LLM generates responses for a batch of input prompts. These responses are subsequently evaluated in the reward stage using various strategies, such as sandbox execution for coding tasks, rule-based logic for mathematical problems, and LLM-as-a-Judge [37] for nuanced tasks including human alignment. In the final training stage, the actor LLM's weights are updated based on the computed reward signal, optionally with a reference LLM to ensure training stability.

To maximize model performance, LLM practitioners often employ *synchronous on-policy* RL post-training to guarantee that responses in the rollout stage are always generated by the most recently updated actor LLM. This is achieved by enforcing a synchronization barrier between the rollout stage and the training stage [18, 24, 35, 42], as illustrated in Figure 1a. However, this synchronization requirement frequently results in severe pipeline "bubbles," especially during rollout stages, which account for around 70% of the total training time in our experiments (see Table 1). Notably, rollout batches typically exhibit a *long-tail distribution* in response length, with the longest response 25×–32× longer than the medium (see Figure 2a). This imbalance leads to prolonged idle periods on GPUs generating short responses, as these devices need to idle wait until the entire batch is completed.

A common approach to mitigating idle bubbles is to overlap the long rollout stage with reward computation (e.g., ROLL [40] and MiMo [44]) and reference model inference (e.g., RLHFuse [55]). However, in LLM reasoning post-training, the combined computation for reward evaluation and reference model inference typically accounts for less than 15% of total training time (see Table 1)—insufficient to fill idle bubbles during long rollout periods.

Many recent RL systems have explored relaxing synchronization constraints for more aggressive stage overlap. One common solution is the "one-off" pipeline, adopted by DeepCoder [23], StreamRL [54], and AsyncFlow [14], wherein rollouts generated in a previous step are used for subsequent training. Some frameworks, such as AReaL [9], even adopt fully asynchronous RL training that continuously performs rollouts and training in parallel. Although these approaches

[*]Wei Gao and Yuheng Zhao contributed equally to this work.

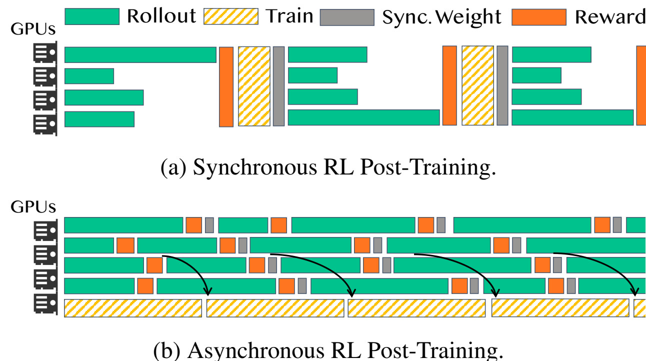
effectively reduce idle bubbles, they often compromise model accuracy because long rollouts are produced with stale model weights relative to short responses. As a result, many RL researchers and practitioners remain hesitant to adopt asynchronous training for LLM post-training.

In this paper, we propose *tail batching*, a novel prompt scheduling strategy designed for on-policy RL training that effectively mitigates GPU bubbles induced by long-tail rollouts. Empirically, we observe that within a rollout batch, only a small subset of prompts produce exceedingly long responses that stall the entire batch. Our key idea is to reorder training samples by consolidating these *tail prompts* into a few *designated* rollout steps, referred to as *long rounds*, while ensuring that the majority of rollout steps (*short rounds*) are composed of balanced, short responses, thereby reducing idle bubbles in GPU utilization. Importantly, because tail batching alters only the order of training samples, it preserves training accuracy, as indicated by recent algorithmic research [30, 49, 51].

To implement this approach, we present `RollPacker`, a system engineered to unlock the full potential of tail batching for on-policy RL training. `RollPacker` initiates rollout in a *short round* to sample P_0 prompts, each producing R_0 responses. To collect balanced, short responses, `RollPacker` employs *speculative execution* for both prompts and responses in a short round: it launches more than P_0 prompts but retains only the first P_0 that finish; each prompt produces more than R_0 responses, finishing after the first R_0 complete. Prompts that generate long responses and are excluded from a short round are deferred into a long-prompt queue. Once the queue accumulates P_0 such prompts, `RollPacker` batch-executes them in a dedicated *long round*, without speculative execution.

`RollPacker` further introduces three system-level optimizations, each addressing a bottleneck in a distinct stage of the RL training pipeline. First, we design a *parallelism planner* that adaptively configures parallelization strategies during rollout. Compared to long rounds, short rounds impose higher GPU memory pressure because speculative execution launches more concurrent requests. As training proceeds, response length distributions change significantly [6]. A fixed parallelization scheme cannot accommodate this variability. To address this, `RollPacker` profiles memory footprint across different batch sizes and sequence lengths, then selects the best tensor parallelism (TP) configuration for each training step based on these profiles and the online response length distribution. This dynamic TP configuration quickly adapts to workload changes over RL training, reducing rollout latency by up to 21.9% in our evaluation (see §6.4).

Second, as rollout cost reduces, reward computation can become a bottleneck, particularly for code execution and LLM-based judging tasks. To address this, `RollPacker` introduces a *reward scheduler* that performs asynchronous, per-sample reward computation. It pipelines reward evaluation in parallel with ongoing rollouts to hide the reward overhead, while dynamically adjusting the compute budget for each reward task



(a) Synchronous RL Post-Training.

(b) Asynchronous RL Post-Training.

Figure 1: Execution workflows of synchronous and asynchronous RL post-training.

based on workload characteristics, such as adjusting sandbox timeouts for code execution or dynamically sharing GPUs for judge models. This approach substantially reduces reward and further speeds up the end-to-end latency for an average of 23.9% in our evaluation (see §6.5).

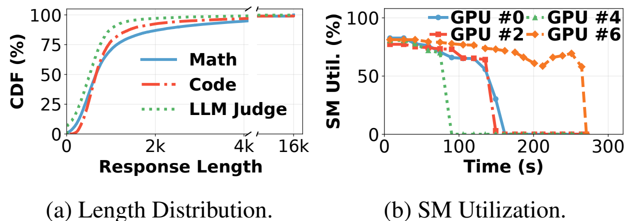
Third, `RollPacker` implements a *stream trainer* that overlaps rollout and training to further reduce GPU idle time. As rollouts progress, especially in long rounds, some GPUs may become idle if their assigned requests complete early. To harvest these idle GPUs, the stream trainer opportunistically initiates training as soon as a partial set of completed prompts is available, while scaling down the GPUs dedicated to rollout. It uses a heuristic algorithm to decide when and which GPUs are reassigned, ensuring minimal disruption to rollout. Completed prompts are asynchronously streamed to the training stage, allowing gradient computation and optional reference logit evaluation to proceed in parallel with ongoing rollouts. To ensure the gradients computed are consistent with those in synchronous on-policy training, `RollPacker` adjusts the loss scales and defers gradient updates until the streaming concludes. This design minimizes idle bubbles across stages while maintaining on-policy training semantics.

We implemented `RollPacker` in 6.6k lines of Python code on top of ROLL [40]. We train models from the Qwen2.5 family (7B–32B) [46] with real-world datasets [17, 45] on a cluster of 128 H800 GPUs using `RollPacker`. Evaluation results show that `RollPacker` substantially outperforms state-of-the-art RL systems, achieving 2.03×-2.56× end-to-end training speedup over veRL [34] and up to 2.24× speedup compared to RLHFuse [55].

2 Background and Motivation

2.1 RL for LLM Post-Training

Reinforcement Learning (RL) has become a pivotal technique for post-training LLMs. Recent advances show that RL algorithms such as GRPO [33] are highly effective in enhancing reasoning capabilities across various domains. An RL



(a) Length Distribution.

(b) SM Utilization.

Figure 2: Characterizing rollout stage: (a) responses across three tasks exhibit a long-tail distribution; (b) long-tail rollouts create prolonged GPU bubbles in the rollout stage.

Table 1: Time breakdown of RL post-training. We train 14B models with a maximum length of 16k using veRL [34] and GRPO [33] with real-world datasets [17, 45] in three tasks.

Task	Rollout	Reward	Training
Math	72%	5%	23%
Code	66%	13%	21%
LLM-as-a-Judge	71%	7%	22%

post-training workflow typically orchestrates multiple models with distinct roles. The *actor LLM* generates responses to input prompts and serves as the primary model being optimized. The *reward LLM* evaluates each response and outputs a scalar reward signal, which can be derived from heterogeneous sources such as sandbox execution for code, rule-based logic for mathematics, or through LLM-as-a-Judge [37] for alignment tasks. To further stabilize optimization, a *reference LLM* is often introduced as a regularizer. Overall, the workflow comprises three stages: (1) *rollout*, where the actor LLM is given P_0 prompts and produces R_0 responses for each prompt; (2) *reward*, where the generated responses are evaluated by corresponding reward workers; and (3) *training*, where the actor LLM updates its parameters based on the computed rewards, optionally constrained by the reference model to mitigate gradient instability.

To maximize model performance, RL post-training is usually performed in a *synchronous* setting, known as *on-policy training*. In this setting, rollout must complete before training begins, and the actor's weights are updated only after training concludes (see Figure 1a). This synchronization requirement ensures that all responses are generated using the most recent model parameters, thereby stabilizing training and improving reliability across tasks [13]. However, it often results in severe pipeline bubbles and low utilization, especially in the rollout stage, as shown in our characterization study.

2.2 Characterization of RL Post-Training

We characterize RL post-training workloads using the Qwen2.5-14B model [46], configured with a maximum response length of 16k tokens, a batch size of 128, and a group size of 8 under the GRPO algorithm [33]. We run math, code, and LLM-as-a-Judge tasks with real-world datasets [17, 45] using veRL [34] on 32 H800 GPUs. For the LLM-as-a-Judge experiments, we employ a 7B-parameter judge model.

The Rollout Bottleneck. Table 1 reports the stage-wise latency distribution for RL training under these settings. The rollout stage dominates runtime, accounting for approximately 70% of each training step across three tasks. The reward stage contributes a smaller fraction (5%–13%), while the training stage accounts for 21%–23% of the total step time. These results underscore that rollout is the primary bottleneck in RL post-training.

Long-Tail Response and GPU Bubbles. A key source of inefficiency in RL post-training lies in the *highly skewed distribution* of response lengths. As shown in Figure 2a, responses across math, code, and LLM-as-a-Judge tasks exhibit a pronounced *long tail* distribution: while most responses are short to moderate in length, with the 75th percentile (P75) ranging between 755 and 1.1k tokens, the longest responses can extend up to 16k tokens.

The presence of long-tail responses results in poor GPU utilization under synchronous rollout. Because all GPUs must wait for the longest responses to complete, devices assigned shorter requests become idle, creating prolonged "bubbles" of wasted cycles. Figure 2b illustrates this problem by reporting SM utilization of even-indexed GPUs on a server with tensor parallelism configured to two. Utilization peaks near 80% at the start of rollout but never reaches full saturation, as LLM decoding is inherently memory-bound. Once short responses finish, utilization of corresponding GPUs quickly drops to zero, with idle periods lasting until the entire batch completes. Given that rollout is already the dominant contributor to training latency, such inefficiency significantly stalls the entire pipeline, a problem widely reported in the literature [9, 14, 16, 55].

2.3 Existing Solutions and Limitations

Prior efforts to mitigate GPU bubbles in RL post-training generally fall into two categories: *stage overlap under synchronization constraints* and *relaxed synchronization*.

Stage Overlap under Synchronization Constraints. This approach seeks to improve resource utilization by pipelining the long-tail rollout with the execution of other stages before the synchronization barrier. For example, RLHFuse [55] overlaps rollout with reward computation and reference model inference, while frameworks like ROLL [40] and MiMo [44] overlap the reward computation of each completed response with the ongoing rollout stage to enable *asynchronous reward computation*. While these designs reduce idle bubbles to some extent, they do not fundamentally address the long-tail responses that dominate rollout time with only modest performance improvements (see §6.1). Moreover, as response lengths in RL post-training continue to grow [1], the relative contribution of reward and reference inference diminishes (typically less than 15% of step runtime as reported in Ta-

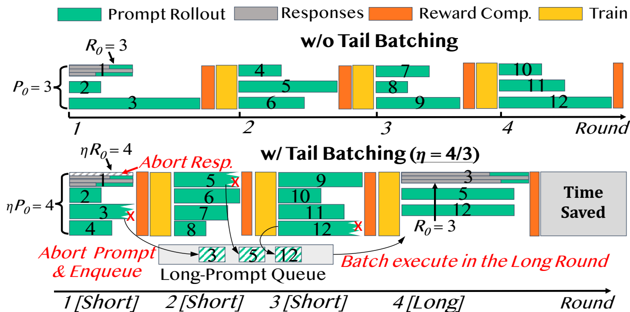


Figure 3: Illustration of Tail Batching.

ble 1), leaving insufficient work to mask the prolonged idle bubbles, even under ideal overlap.

Relaxed Synchronization. A second line of work adopts a more aggressive strategy by relaxing the strict synchronization barriers between rollout and training, as illustrated in Figure 1b. For example, Kimi [39] introduces *partial rollout* by truncating the long-tail responses and preserving generated tokens to continue rollouts in subsequent steps. StreamRL [54], AsyncFlow [14], and RhymeRL[1] [16] allow a one-step staleness, enabling the training stage to proceed with slightly outdated rollouts in a *one-off pipeline*. Pushing further, AReaL [9] introduces *fully asynchronous* RL training, in which rollout and training are completely decoupled, and updates may rely on samples generated many steps earlier. These methods are effective in reducing GPU bubbles, but they introduce a fundamental trade-off: by relaxing synchronization, they compromise the on-policy nature of training, often leading to degraded accuracy and reduced stability.

In summary, existing solutions either provide only marginal improvements by overlapping non-bottleneck stages with rollout, or sacrifice on-policy guarantees by relaxing synchronization. Neither of these approaches can eliminate the inefficiency introduced by long-tail rollouts while preserving the accuracy and stability of synchronous RL training.

3 Tail Batching

In this section, we introduce *tail batching*, a novel prompt scheduling strategy that fundamentally alleviates imbalanced response lengths to reduce GPU bubbles while preserving on-policy RL training semantics without accuracy loss.

Tail Batching. GPU bubbles arise primarily from a small subset of prompts that generate disproportionately long responses. A naive approach would be to exclude such long-tail responses from rollout batches. However, this approach introduces two critical issues: (**P1**) rollout stages may fall short of the required number of prompts or responses that constitute an effective batch; and (**P2**) systematically excluding long prompts distorts the training sample distribution, potentially harming model performance. Tail batching addresses these problems with two key techniques.

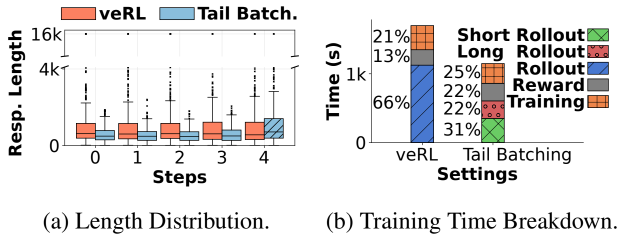
To address **P1**, tail batching leverages *speculative execution* [50] by over-provisioning requests while selectively retaining only the fastest completions. Under the GRPO algorithm [33], each rollout step requires sampling P_0 prompts, each with R_0 responses (Figure 3-top). Instead of launching exactly P_0 prompts, tail batching starts more and admits only the first P_0 to complete. Similarly, each prompt produces more than R_0 responses, but only the first R_0 are retained. This "race-to-completion" speculation naturally filters out long responses, yielding balanced, shorter batches that minimize idle bubbles while preserving the required batch size.

To address **P2**, tail batching guarantees that no prompt is permanently excluded. As shown in Figure 3-bottom, prompts aborted during speculative execution are added to a *long-prompt queue*. Once the queue reaches size P_0, these prompts are batch-scheduled in a dedicated *long round*, where speculative execution is *disabled* to allow full-length responses to be generated. Because such prompts are rare, long rounds occur infrequently and are interleaved with frequent *short rounds* composed of balanced responses. This design ensures that all prompts are eventually included, while the majority of rollout steps remain efficient.

Training Accuracy. From a statistical perspective, tail batching strictly reorders short and long prompts into separate rounds without altering the underlying sample distribution or relaxing synchronization. Our approach guarantees that every prompt is eventually included in a training update. Prior studies confirm that such changes in training order do not degrade model accuracy [3, 4, 8]. In fact, the training dynamics of tail batching resemble *curriculum learning* [38, 51]: by prioritizing shorter (often easier) rollouts, it acts as a natural curriculum that prior work suggests can stabilize gradient descent [38]. Our design is consistent with recent algorithmic research on RL post-training, including SPEED-RL [51], APRIL [56], MoPPS [30], and SortedRL [52], all of which explicitly explore prompt reordering to enhance efficiency without sacrificing performance. Empirically, as shown in Figure 9, tail batching achieves accuracy curves nearly identical to those of standard synchronous RL training, validating that our scheduling strategy preserves model quality.

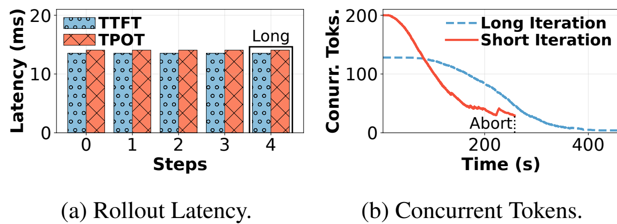
Rollout Efficiency. We empirically validate tail batching's effectiveness in enhancing rollout efficiency by training a Qwen2.5-14B model [46] on the code dataset [45] under the same settings described in §2.2. The speculation factor is set to $\eta = 1.25$, meaning that in each short round, the actor LLM speculatively launches ηP_0 prompts, each generating ηR_0 responses, while accepting only the first $P_0 \times R_0$ completions. Figure 4a compares the response length distribution over five training steps, with and without tail batching. Compared to the baseline approach (veRL [34]), tail batching yields shorter and more balanced responses in the first four steps (short

[1] RhymeRL's HistoPipe scheduling requires one-step off-policyness (see Figure 10 and evaluation in §7.3 of [16]).



(a) Length Distribution. (b) Training Time Breakdown.

Figure 4: Tail batching vs. the baseline veRL when training a Qwen2.5-14B model on the code dataset [45]. (a) Box plot of response length distribution across five rounds, where the hatched box is a long round under tail batching. Whiskers measure 1.5 IQR. (b) Training time breakdown, where the total time is a cumulation of 5 consecutive steps, a full period comprising four short rounds and one long round.



(a) Rollout Latency. (b) Concurrent Tokens.

Figure 5: Quantitative analysis of speculative execution overhead with tail-batching. We train a Qwen3-8B model on a mathematical dataset [17], while the max response length is set to 32k. We run over 20 consecutive steps and report the average results. (a) We show the average TTFT and TPOT in short and long rounds. (b) We show the number of concurrent tokens in a short round and a long round, representing the memory pressure of the rollout phase.

rounds), reducing the maximum response length by up to 8.9×. Prompts producing long responses are deferred to the fifth step (long round), where outputs are generally longer than the baseline but capped at the same maximum of 16k tokens. This reorganization substantially reduces rollout costs and shortens end-to-end training time by 1.48× (Table 2).

Figure 4b further breaks down the training time across stages. With rollout overhead mitigated by tail batching, the relative contributions of the reward and training stages become more pronounced. Moreover, short and long rounds exhibit drastically different resource usage profiles, implying that a uniform rollout strategy is suboptimal. These findings motivate the system-level optimizations tailored to each stage of the RL pipeline, which we develop next.

Overhead of Speculative Execution. We empirically quantify the overhead of speculative execution in short rounds. Figure 5a reports both Time-To-First-Token (TTFT) and Time-Per-Output-Token (TPOT). We observe that TTFT and TPOT remain nearly identical between short and long rounds. Although tail batching computes extra tokens that are eventually discarded, the GPU decoding process is primarily memory-bandwidth bound rather than compute bound. Consequently, the marginal latency cost of increasing the batch size for speculative execution is negligible, ensuring that rollout throughput remains unaffected.

While speculative execution incurs negligible latency overhead, Figure 5b shows that short rounds initiate with substantially higher concurrent tokens compared to long rounds, creating a distinct peak in KV cache memory pressure. This sharp disparity in memory footprints between short and long rounds renders a static parallelism strategy inefficient. It necessitates the adaptive approach we introduce in §4.2, where we dynamically adjust parallelism strategies to handle these fluctuations without resource waste.

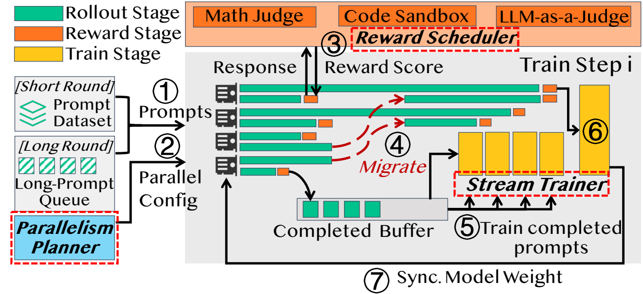


Figure 6: System overview and workflow of `RollPacker`.

4 `RollPacker` System Design

In this section, we present `RollPacker`, an efficient on-policy RL system engineered to realize the benefits of tail batching through a holistic design. We begin with a system overview, then provide detailed descriptions of each component.

4.1 System Overview

`RollPacker` incorporates three key components: the parallelism planner, reward scheduler, and stream trainer, each addressing a distinct bottleneck in rollout, reward, and training stages.

Parallelism Planner. Short rounds, which employ speculative execution, create higher GPU memory pressure than long rounds. A fixed tensor parallelism (TP) configuration cannot adapt to the changing resource profiles across short and long rounds, resulting in frequent KV cache preemption overhead and degraded efficiency. `RollPacker` introduces a parallelism planner that dynamically profiles workloads and selects optimal TP configurations each step to cut rollout overhead.

Reward Scheduler. With rollout costs reduced, reward computation becomes more pronounced (Figure 4b). To prevent it from becoming the new bottleneck, `RollPacker` employs a reward scheduler that pipelines reward computation in parallel with rollouts while dynamically budgeting compute for each sample evaluation, effectively reducing its overhead.

Stream Trainer. In long rounds where speculative execution is disabled, imbalanced responses lead to pronounced

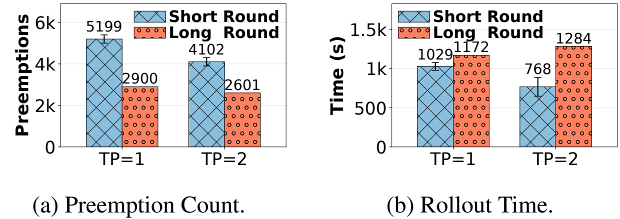
GPU bubbles (Figure 4a). The stream trainer advances prior stage-overlapping approaches [40, 55] by introducing a more fine-grained overlap between rollout and training: completed prompts are streamed into training immediately, while idle GPUs are reassigned from rollout to gradient computation. To maintain on-policy semantics, the stream trainer carefully scales gradients and defers weight updates until the full rollout completes, preserving accuracy while reducing idle time.

Workflow. `RollPacker` operates in two phases: an *offline profiling phase* and *online execution phase*.

1) Offline Profiling: `RollPacker` benchmarks the actor LLM's prefilling and decoding throughput under different TP sizes, batch sizes, and sequence lengths. It also profiles the GPU memory footprint and runtime cost of the judge LLM across varying sequence lengths. Offline profiling is lightweight: excluding LLM runtime initialization overhead, it takes at most 4 seconds per configuration, and the full profiling suite can be completed within 5 minutes on a single GPU server with eight H800 GPUs. This one-time offline cost is negligible relative to total training time (under 2%).

The profiled results are then used for guiding online decisions. There are several empirical observations that motivate our system design. First, the prefilling overhead at the maximum sequence length with batch size 1 is within 100 milliseconds. This suggests that, during rollout, we can migrate a small number of selected requests to improve resource utilization without incurring substantial recomputation overhead (§4.4). Second, the time between output tokens for batch sizes ranging from 32 to 64 does not vary significantly: the max/min ratio is 1.23. This indicates that decoding latency remains stable over a moderate range of batch sizes because decoding is largely memory-bandwidth bound. However, GPU memory footprint roughly doubles over this range, which can lead to significant GPU memory pressure. This observation aligns with Figure 5 and motivates the design of the parallelism planner (§4.2).

2) Online Execution: `RollPacker` orchestrates rollout, reward, and training in a synchronous RL job (Figure 6). ① During rollout, tail batching decides whether to apply speculative execution based on the size of the long-prompt queue. ② The parallelism planner then selects an optimal TP configuration by combining historical job loads with profiled performance data. ③ In parallel, the reward scheduler overlaps evaluation with rollout and dynamically adjusts budgets for each reward task. ④ Concurrently, the stream trainer monitors rollout progress to determine when to reassign GPUs from rollout to training. ⑤ As prompts complete, they are streamed into training for immediate gradient computation. ⑥ Once the full rollout completes, the stream trainer stops streaming, accumulates all computed gradients, and triggers a synchronized gradient computation and update across all available GPUs. ⑦ Finally, the updated actor weights are synchronized with the rollout stage before the next RL step begins.



(a) Preemption Count. (b) Rollout Time.

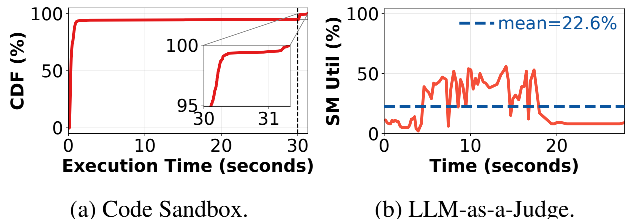
Figure 7: Rollout performance when training Qwen3-8B/32k on eight H800 GPUs. (a) Preemption count in each step. (b) Rollout time of each step. The metric is collected in one consecutive period of 4 short rounds and one long round.

4.2 Parallelism Planner

Short Rounds Create High Memory Pressure. As described in §3, tail batching increases the number of concurrent responses in short rounds, placing greater pressure on GPU memory. Existing LLM serving engines [20, 32] typically alleviate memory pressure by preempting ongoing requests, i.e., swapping out their KV cache to free GPU memory for others. A high preemption count thus indicates extensive memory contention. Figure 7a shows that when training Qwen3-8B with a 32k response length and a batch size of 128, short rounds incur up to 1.79× more preemptions than long rounds, introducing substantial computational overhead.

Increasing TP Alleviates GPU Memory Pressure. Tensor Parallelism (TP) is a standard technique to alleviate GPU memory pressure. As shown in Figure 7a, configuring a larger TP size partitions model weights across more GPUs, freeing memory for KV cache and cutting preemption counts by 21.1% in short rounds and 10.3% in long rounds. This additional KV cache capacity alleviates memory contention and shortens rollout latency. As shown in Figure 7b, with TP=1, the rollout time in a short round is 87% of that in a long round, negating the gains of tail batching; increasing TP size to 2 reduces short-round duration by 25%. However, a larger TP size also introduces communication overhead, which dominates in long rounds where rollout is bound by long-tail responses, eroding performance.

Adaptive TP Selection. Most RL training frameworks [18, 34] adopt a fixed parallelism configuration, which our analysis shows is inefficient for tail batching (Figure 7). Optimal TP sizes differ between short and long rounds, necessitating the design of dynamic adaptation. `RollPacker` introduces a *parallelism planner* to reconfigure TP sizes on a per-step basis with negligible overhead. In the offline phase, the planner profiles optimal TP sizes without tail batching and uses them as default configurations at the beginning of the training. It then keeps track of the preemption counts and adapts the TP sizes accordingly using a lightweight heuristic: a sudden rise in preemptions (e.g., $> 1.05\times$) triggers an increase in TP (doubling the size), while sustained zero preemptions across four steps trigger a decrease (halving the size). To limit cross-node communication, TP groups are constrained within



(a) Code Sandbox. (b) LLM-as-a-Judge.

Figure 8: Reward computation introduces a non-negligible overhead with tail batching. (a) The distribution of sandbox execution time for code responses. (b) The SM utilization on the GPU allocated to a 7B-parameter judge LLM over time.

a single GPU server.

4.3 Reward Scheduler

With rollout costs reduced, reward computation becomes a non-trivial contributor to end-to-end latency (Figure 4b). To mitigate this, `RollPacker` pipelines reward evaluation with rollout and adaptively budgets compute for each task.

Asynchronous Reward Computation. In this method, reward evaluation is performed asynchronously: completed responses are dispatched to reward workers in parallel with the ongoing rollout stage, similar to ROLL [40] and MiMo [44]. This design overlaps reward evaluation with rollout, partially hiding the overhead. However, only relying on this design is insufficient to address the potential bottleneck, especially for code sandbox execution and judge LLM evaluation.

Code Sandbox Execution. In coding task, practitioners often impose a maximum execution timeout per test case. For example, in our training experiments described in Figure 4, a 30-second timeout is enforced. Yet, as shown in Figure 8a, around 5% of prompts hit this timeout. These prolonged executions, which ultimately yield zero reward, delays the entire reward stage, stretching it to nearly 22% of the total training time (Figure 4b). Since many correct responses complete much faster, a fixed timeout wastes substantial computation on doomed samples.

`RollPacker` introduces an *adaptive timeout mechanism*. For each test case, it tracks the maximum execution time among correct responses during training, denoted as T_{anchor}. When a new response exceeds this threshold, sandbox execution is terminated early and a zero reward is assigned. In view of the potential CPU contention during code execution and to avoid overly aggressive cutoffs, the timeout is relaxed to

$$T_{\text{timeout}} = \min\left(\max(T_{\min}, \lambda T_{\text{anchor}}), T_{\max}\right),$$

where we empirically set $\lambda = 1.5$, $T_{\min} = 2$s, and $T_{\max} = 30$s to attain good performance. This design fast fails doom cases while preserving the evaluation of promising responses.

LLM-as-a-Judge. In asynchronous reward computation, RL systems often reserve a fixed number of GPUs (e.g., 25% of

Algorithm 1 Stream Trainer

1: **Input:** Requests R, GPUs G
2: **procedure** StreamTrainer(R, G)
3: $G_{\text{rollout}} \leftarrow G$; $G_{\text{train}} \leftarrow \emptyset$
4: $R_{\text{run}} \leftarrow R$; $R_{\text{comp}} \leftarrow []$
5: $\mathit{scaled_down} \leftarrow$ false
6: $\Delta_R \leftarrow 0$
7: **while** $|R_{\text{run}}| \neq 0$ **do**
8: $R_{\text{fin}} \leftarrow$ LLM.generate($R_{\text{run}}, G_{\text{rollout}}$)
9: $\Delta_R \leftarrow \Delta_R + |R_{\text{fin}}|$
10: **for** *req* in R_{fin} **do**
11: R_{run}.remove(*req*)
12: R_{comp}.append(*req*)
13: **if** not *scaled_down* **then**
14: **if** $0.2 \le |R_{\text{comp}}|/|R| \le 0.5$ and $\Delta_R/|R| \ge 0.05$ **then**
15: $\Delta_R \leftarrow 0$
16: $G_{\text{free}} \leftarrow$ PickScaleDownGPUs(G)
17: **if** MeetScaleCriteria(G_{free}) **then**
18: $G_{\text{rollout}} \leftarrow G_{\text{rollout}} \setminus G_{\text{free}}$
19: $G_{\text{train}} \leftarrow G_{\text{free}}$
20: MigrateRequests($G_{\text{free}}, G_{\text{rollout}}$)
21: $\mathit{scaled_down} \leftarrow$ true
22: **if** *scaled_down* **then**
23: ComputeGrad($G_{\text{train}}, R_{\text{comp}}$)

total GPUs) *exclusively for the judge LLM* to avoid interference with other workers. However, this strategy results in poor utilization. As shown in Figure 8b, when a 7B-parameter judge LLM scores responses, its reserved GPU achieves only $\sim 22.6\%$ average SM utilization. The inefficiency stems from the fact that the judge typically processes small batches of responses, leaving much of the reserved capacity idle.

To improve efficiency, `RollPacker` *colocates the judge LLM with the actor LLM* on the same GPU devices for concurrent execution. This design, however, introduces two issues. First, rollout and reward evaluation now share GPU resources, potentially interfering with one another. Nevertheless, we observe that neither the actor LLM nor the judge LLM alone saturates GPU SM utilization. To enable efficient sharing, `RollPacker` enables *Multi-Process Service* (MPS) [25], which partitions GPU resources at the warp level and allows both models to run concurrently with minimal interference.[2]

Second, hosting both models on the same GPU risks exhausting memory, as the actor LLM already requires substantial space for its KV cache. To address this, `RollPacker` introduces a *layer-wise pipeline scheme* that reduces the memory footprint of the judge LLM. Inspired by prior work [2], it offloads most layers of the judge LLM to host memory and streams its parameters over PCIe in sync with activation computation on the GPU. Since rollout rarely saturates PCIe bandwidth, this pipelined offloading imposes little overhead. `RollPacker` dynamically adjusts the number of offloaded layers to accommodate varying input sequence lengths, ensuring the judge LLM fits within memory while maximizing utilization.

[2] We utilize `MPS` [25] to enable spatial GPU sharing, which does not guarantee error isolation. As future work, we plan to explore Green Contexts [26] to improve fault tolerance.

4.4 Stream Trainer

Despite prior optimizations, long rounds still suffer from idle bubbles as responses complete unevenly (Figure 4b). The *stream trainer* mitigates this with a novel stage overlap strategy that pipelines ongoing rollouts with gradient computation, reducing end-to-end latency while preserving the synchronous on-policy RL semantics.

Repurposing Rollout GPUs for Training. As rollout advances, GPU utilization declines. The stream trainer scales down the number of GPUs dedicated to rollout and *repurposes* the freed devices for training. Training on repurposed GPUs proceeds with only a subset of data-parallel replicas; gradient updates are deferred until rollout fully completes, preserving the correctness of on-policy RL. Reference logits, which contribute only marginally to the workload, can be computed by temporarily swapping actor and reference model weights if needed.

Algorithm 1 outlines the general workflow of stream trainer. It continuously monitors rollout progress and triggers GPU downscaling once the fraction of completed requests exceeds a threshold (Line 14). When scaling criteria are met (Line 17), a subset of rollout GPUs is repurposed for training (Lines 18-19). To migrate ongoing requests, `RollPacker` employs a *recomputation-based policy* (Line 20): generated tokens are preserved while KV caches are recomputed to resume rollout on the remaining GPUs with minimal overhead [10, 16]. Once training instances are launched, the stream trainer asynchronously fetches completed responses and computes gradients in parallel with the ongoing rollout (Line 23).

Scaling Criteria. The trainer evaluates two criteria in Algorithm 1 (Line 17) to maximize the benefits of GPU scaling.

1) Which GPUs to scale down? The trainer must carefully select GPUs to reassign from rollout to training, since the two stages often rely on different communication group topologies. To ensure correctness, tightly coupled groups, such as TP groups used in rollout, must remain intact and cannot be split across rollout and training. In practice, the stream trainer attempts to repurpose half of rollout GPUs for training. Before taking actions, it validates whether this is possible without splitting communication groups needed by a data-parallel replica. If not, the scaling attempt is aborted.

2) When to scale? Downscaling rollout GPUs risks slowing response generation. By consolidating more requests onto fewer GPUs, it enlarges per-device batch sizes, aggravating the memory pressure for KV cache and eventually harming rollout throughput (§4.2). To prevent this, the stream trainer calculates peak KV cache usage by combining historical response length distributions with per-token cache footprints when the fraction of completed requests reaches milestones between 20% and 50% (in 5% increments). GPU scaling is triggered only if the projected peak cache demand remains within memory limits after migration. For simplicity, we omit the modest overhead introduced by recomputation-based request migration and the minor decoding throughput reduction after scaling (§6.6).

Overlapped Stream Execution. Once the scaling criteria are met and GPUs are reassigned, the stream trainer begins processing completed responses on the repurposed training GPUs. Rollout and training now operate as a producer-consumer pair: rollout generates responses, while training consumes them through a streaming model that aligns production and consumption rates. The stream trainer asynchronously fetches completed responses and computes gradients in parallel with ongoing rollouts, thereby reducing the overall step time.

Preserving On-Policy Semantics. A critical requirement of the stream trainer is to ensure that the gradient computations are *mathematically equivalent* to the standard on-policy training pipeline. This guarantee is maintained in two phases. First, during stream execution, gradients for completed responses are computed and buffered, but *no updates* are applied to model parameters or optimizer state. We extend the underlying LLM training framework [36] to disable gradient synchronization during back-propagation, ensuring strict adherence to on-policy constraints. Second, after rollout completes, the remaining responses are distributed across all data-parallel replicas for final gradient computation and model updates. Because the streaming nature of `RollPacker` results in an uneven distribution of processed samples across replicas, a standard arithmetic mean (i.e., simple All-Reduce averaging) would introduce a statistical bias toward replicas with fewer samples. To resolve this, we implement a sample-weighted normalization scheme, and each replica i scales its locally accumulated gradient sum by the factor $\frac{n_i}{N}$, where n_i is the number of samples processed by replica i and N is the total global batch size. `RollPacker` ensures that the final global gradient is numerically identical to one computed over a monolithic batch, thereby ensuring the final update is equivalent to that of standard on-policy training.

5 Implementation

We implemented `RollPacker` in ∼6.6k lines of Python code on top of ROLL [40]. The system seamlessly integrates existing LLM infrastructure with lightweight extensions for rollout, reward, and training.

Rollout Stage. `RollPacker` uses vLLM v0.8.4 [20] as the serving backend. Each rollout instance supports request-level routing, allowing requests to be directed to specific instances. We extend vLLM's `abort_request` and `add_request` interfaces to terminate in-progress requests and resubmit them elsewhere, enabling speculative execution and migration.

Reward Stage. Reward evaluation is implemented with `ray.remote`. Code sandbox execution and mathematical evaluation run on CPUs, with per-task timeouts of 30s and 2s, respectively. We employ `torch.cuda.Stream` to manage GPU

streams for activation computation and parameter transfers.

Training Stage. Actor training is built on Megatron-LM v0.12.2 [36], with optimizer states partitioned across GPUs. In the stream trainer, gradients are computed without loading optimizer states. Gradient tensors are offloaded to host and later reloaded into GPU memory when synchronizing across all GPUs for final gradient computation and updates.

6 Evaluation

We evaluate `RollPacker` on Qwen2.5 models using a diverse benchmark of real-world datasets. Our evaluation is designed to answer the following key questions:

- **End-to-End Performance:** How does `RollPacker` compare against state-of-the-art RL post-training systems in terms of training efficiency and model accuracy? (§6.1)
- **System Breakdown:** What are the individual performance contributions of tail batching, the parallelism planner, the reward scheduler, and the stream trainer? (§6.2–§6.6)
- **Generalizability:** Can `RollPacker`'s optimizations benefit relaxed-synchronization paradigms, such as asynchronous one-off training pipelines? (§6.7)
- **Scalability:** How efficiently does RollPacker scale across varying batch sizes and large GPU deployments? (§6.8)

Cluster Setup. We deploy `RollPacker` on a dedicated cluster of 16 nodes, comprising a total of 128 NVIDIA H800 GPUs interconnected via a 400 Gbps InfiniBand network.

Models. We evaluate our system using the Qwen2.5 [46] family of models with 7B, 14B, and 32B parameters, configured with maximum response lengths of 8k, 16k, and 32k tokens, respectively. Our end-to-end multi-task evaluation utilizes a uniform mix of real-world datasets spanning mathematics [17], code generation [45], and multi-subject question answering, where responses are evaluated using rule-based logic, code sandboxes, and LLM-as-a-Judge reward workers (using Qwen2.5-7B-Instruct), respectively.

Configurations and Metrics. By default, we adopt a synchronous RL training setting with $P_0 = 128$ prompts and $R_0 = 8$ responses per prompt. The actor and reference models share the same parameter size. Resource allocation and parallelism strategies scale with model size: the 7B, 14B, and 32B models are trained on 16, 32, and 64 GPUs, respectively. Their rollout TP is set to 1, 2, and 2, while the training stage parallelism dimensions (TP, PP, CP) are configured as (2,1,1), (2,2,2), and (4,1,4), respectively. To comprehensively assess system capability, we report validation accuracy across training steps as our metric for effectiveness, and end-to-end training time per step as our metric for efficiency.

6.1 End-to-End System Performance

We compare the end-to-end performance of `RollPacker` with two state-of-the-art synchronous RL post-training systems:

- **veRL [34]:** A popular framework that utilizes a hybrid programming model and an optimized 3D-HybridEngine to enhance the rollout and training efficiency.
- **RLHFuse [55]:** A pioneering RL system that pipelines reward computation and reference model inference with the rollout stage. To ensure a strong baseline, we strengthen RLHFuse by integrating our *asynchronous reward computation* and *stream trainer* optimizations.

Model Accuracy and Convergence. Figure 9 illustrates the average validation scores of `RollPacker` and veRL. The results demonstrate that tail batching strictly preserves training accuracy across varying model sizes and response lengths. Furthermore, `RollPacker` consistently exhibits faster convergence during the early stages of training. We hypothesize this accelerated convergence is a direct benefit of the more balanced response length distribution produced by tail batching.

Training Step Time. Figure 10 plots the training step time for each model over the initial 40 steps. Overall, `RollPacker` consistently outperforms veRL and the strengthened RLHFuse across all three LLM sizes. Compared to veRL, `RollPacker` achieves significant end-to-end speedups of 2.03×, 2.22×, and 2.56× for the 7B, 14B, and 32B models, respectively. When evaluated against RLHFuse, `RollPacker` delivers speedups of 1.14×, 1.68×, and 2.24×.

Thanks to the integration of the reward scheduler and stream trainer, both `RollPacker` and RLHFuse maintain a performance edge over veRL during long rounds. However, as the maximum response length increases (e.g., up to 32k tokens, as shown in Figure 10c), the relative benefits of stage overlapping begin to diminish because rollout generation increasingly dominates the total step time. Conversely, during short rounds, tail batching completely eliminates the long-tail generation bottleneck, enabling `RollPacker` to achieve speedups of 2.1× to 3.6× over veRL, and 1.2× to 3.2× over the strengthened RLHFuse.

6.2 System Performance Breakdown

Ablation Study. To quantify the performance contributions of `RollPacker`'s core techniques, we conduct an incremental ablation study. Starting from the veRL baseline, we sequentially integrate each system component to evaluate its impact on end-to-end training speedup across varying model sizes and response lengths, as detailed in Table 2.

- **Tail Batching:** By mitigating the long-tail rollout overhead, tail batching provides foundational speedups that become increasingly pronounced at longer response lengths. For the Qwen2.5-32B/32k configuration, it achieves a 2.21× speedup over the veRL baseline.

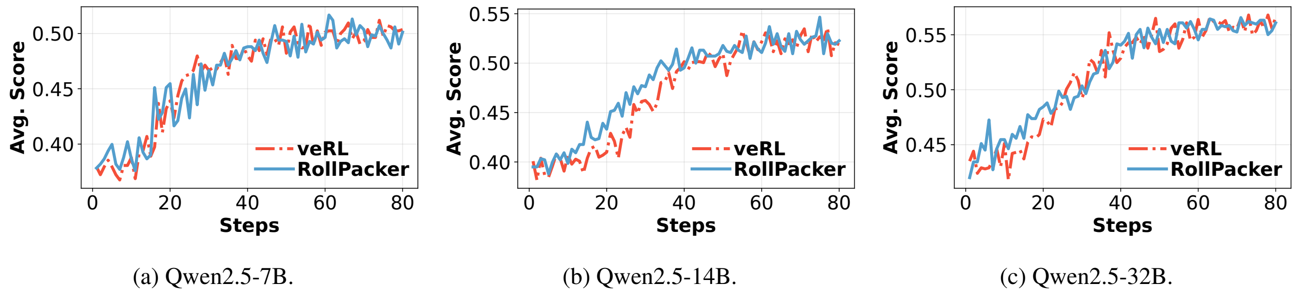


Figure 9: The average validation score for training Qwen2.5-7B, 14B and 32B model with veRL and RollPacker.

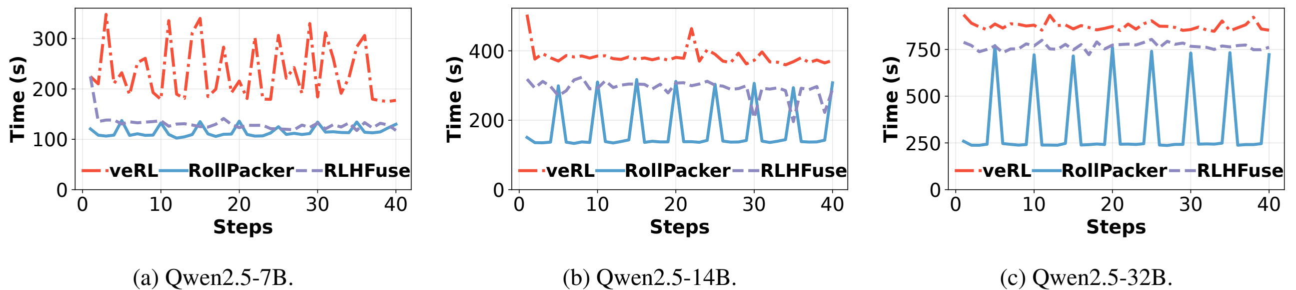


Figure 10: The step time for training Qwen2.5-7B, 14B and 32B model with veRL, RLHFuse and RollPacker.

Table 2: Ablation Study of RollPacker's Core Techniques.

Method	Qwen2.5-7B/8k	Qwen2.5-14B/16k	Qwen2.5-32B/32k
veRL Baseline	1.00	1.00	1.00
+ Tail Batching	1.30×	1.48×	2.21×
+ Reward Scheduler	2.01×	1.99×	2.48×
+ Parallelism Planner	2.01×	2.02×	2.52×
+ Stream Trainer	2.03×	2.22×	2.56×

- **Reward Scheduler:** Asynchronous reward computation and adaptive scheduling yield significant benefits, especially for short rollouts. With an 8k response length, it achieves the overall speedup from 1.30× to 2.01×. Its effectiveness also extends to longer sequences, improving the speedup from 2.21× to 2.48× at a 32k response length.
- **Parallelism Planner:** Dynamic parallelism adaptation proves effective under high memory pressure, a scenario that typically emerges when scaling to large models and extended sequences. In the Qwen2.5-32B/32k setting, the planner provides an additional performance edge, lifting the speedup to 2.52×.
- **Stream Trainer:** By overlapping rollout generation and gradient computation, the stream trainer effectively hides training latency. This pipelining yields a substantial 20% performance improvement (from 2.02× to 2.22×) in the Qwen2.5-14B/16k setting, where rollout and training durations are well balanced. Across other settings, the stream trainer consistently improves the speedup gains, culminating in RollPacker's end-to-end performance.

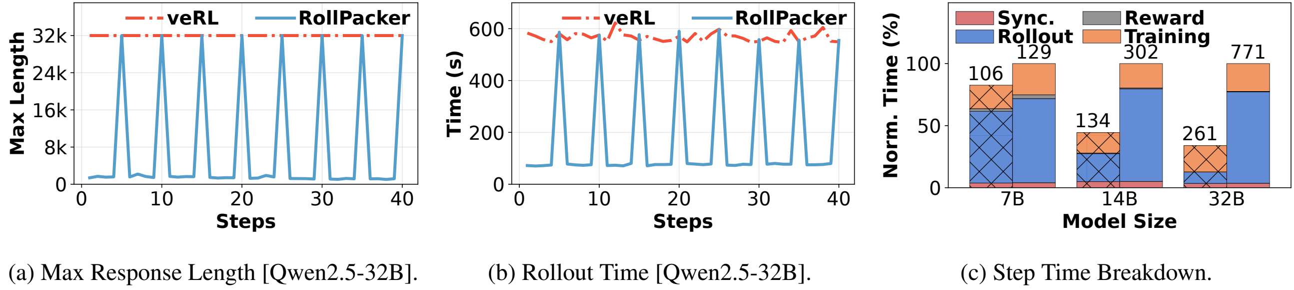
Training Step Breakdown. Figure 11 further breaks down the training time to reveal exactly where RollPacker saves GPU cycles. Figures 11a and 11b highlight the contrast between RollPacker and veRL regarding maximum response length and rollout time. While long rounds exhibit comparable rollout times between the two systems, RollPacker achieves a significant advantage in short rounds. By substantially reducing the maximum response length (Figure 11a), RollPacker delivers up to a 7.8× speedup in average rollout time (Figure 11b). Figure 11c aggregates these stage-wise breakdowns, comparing short rounds (hatched bars) and long rounds (solid bars). In long rounds, the rollout stage dominates the step time. However, the salient time savings harvested during the much more frequent short rounds significantly lower the average step time across the entire training period.

Having established these macro-level benefits, we next investigate the micro-level behavior of each system component to quantify their optimizations (§6.3–§6.6)

6.3 Sensitivity Analysis of Tail Batching

Figure 12 presents the rollout time under various configurations. To isolate the effects of speculative execution, we independently scale the number of prompts (P) and the number of responses per prompt (R). We then compare these independent scaling strategies against RollPacker's joint scaling approach, where both dimensions share a speculation factor of $\eta = 1.25$.

Impact of Response Speculation (R)**.** Fixing the number of prompts at $P = P_0$, we scale the response speculation factor



(a) Max Response Length [Qwen2.5-32B]. (b) Rollout Time [Qwen2.5-32B]. (c) Step Time Breakdown.

Figure 11: Breakdown of training step time in RollPacker. (a) Maximum response length per training step for Qwen2.5-32B/32k. (b) Total rollout time per training step for Qwen2.5-32B/32k. (c) Breakdown of step time for different models, comparing short rounds (hatched bars) with long rounds (solid bars). The time for each component is normalized to the total time of the long round for that model. Absolute step times are displayed as labels on each bar.

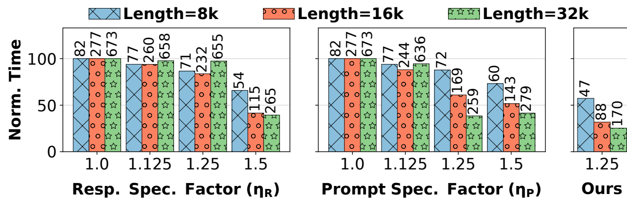


Figure 12: **[Tail Batching]** The rollout time of different configurations. We set P and R fixed to P_0, and R_0, respectively, while changing η for the other parameter. Each bar represents the average iteration time of a full period of consecutive short and long rounds, normalized to the baseline without tail batching, and is annotated with the actual time.

η_R from 1.0 to 1.5. Generating extra responses allows the system to eagerly discard long-tail outliers, thereby reducing rollout time. However, because inherently difficult prompts consistently generate long responses, a substantial reduction in rollout time only materializes when η_R is aggressively increased to 1.5.

Impact of Prompt Speculation (P)**.** Alternatively, fixing the response count at $R = R_0$, we over-provision the number of prompts P. In this configuration, the system collects the fastest P_0 prompts and defers the remaining slow-completing prompts to the long-prompt queue. However, increasing the prompt speculation factor η_P proportionally increases the frequency of long rounds. This frequent triggering of long rounds negates the latency savings achieved during the short rounds. For instance, at a 32k maximum response length, the average rollout time actually increases when η_P is raised from 1.25 to 1.5.

Choosing a Robust Default. Based on these isolated analyses, we configure RollPacker to jointly scale both P and R with $\eta = 1.25$, serving as a robust default for general RL training workloads. Under this balanced setting, tail batching accelerates the average rollout speed by up to 3.9×, outperforming the fixed-P_0 and fixed-R_0 configurations by up to 1.5× and 1.6×, respectively. For practitioners optimizing for specific datasets, we recommend conducting a lightweight grid search over a narrow bound (e.g., $\eta \in [1.1, 1.4]$) during the initial training steps to further fine-tune performance.

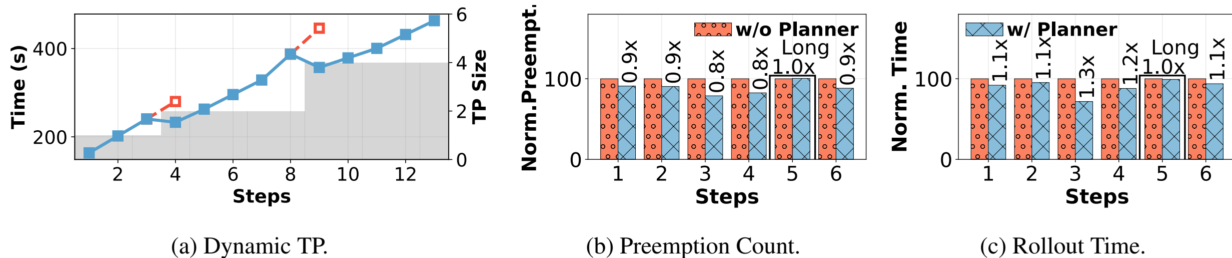
6.4 Parallelism Planner

Dynamic Tensor Parallelism. The parallelism planner adaptively adjusts the TP size during the rollout stage based on the response length distribution. To analyze this behavior, we evaluate the rollout latency of the Qwen2.5-14B model on 16 GPUs over training steps. Initially fixing the TP size to 1, we artificially scale the response length from 8k to 32k in increments of 2k tokens per step. Figure 13a plots the average rollout time (left) and the optimal TP size (right). Specifically, at steps 3 and 8, the planner automatically increases the TP size to 2 and 4, respectively, effectively curbing the latency growth. Comparing the iteration time under dynamic TP scaling (blue solid line) against a fixed-TP baseline (red dashed line) reveals a clear and substantial reduction in rollout time. Overall, adaptive TP selection yields an average 1.9× speedup compared to strictly maintaining a TP size of 1.

Preemptions and Rollout Latency. We further evaluate the planner's impact on KV cache preemptions and step latency by fixing the maximum response length at 32k and the initial TP size at 2. Figures 13b and 13c report the normalized preemption counts and rollout times per step, respectively. As shown in Figure 13b, the parallelism planner successfully mitigates memory pressure, reducing the preemption count in short rounds by an average of 13.8%. This reduction in memory contention translates directly to performance gains, accelerating rollout time in short rounds by 1.11× to 1.28×.

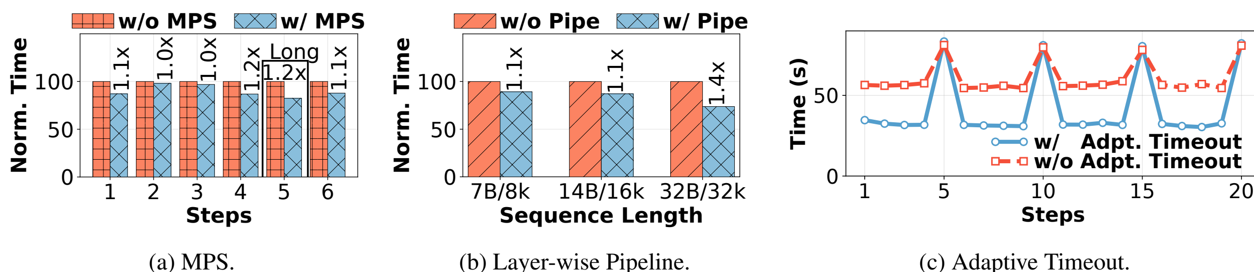
6.5 Reward Scheduler

Compared to synchronous execution, asynchronous reward computation yields end-to-end speedups of 1.48×, 1.35×, and 1.18× across the three LLMs, respectively. We analyze



(a) Dynamic TP. (b) Preemption Count. (c) Rollout Time.

Figure 13: **[Parallelism Planner]** (a) Rollout time (line, left axis) and corresponding TP size (bar, right axis) when training Qwen2.5-14B. The response length increases linearly from 8k to 32k in 2k increments. Red dashed lines indicate rollout time without TP adjustment. (b)–(c) Normalized preemption count and rollout time per step with and without the parallelism planner. The fifth step corresponds to the long round.



(a) MPS. (b) Layer-wise Pipeline. (c) Adaptive Timeout.

Figure 14: **[Reward Scheduler]** *LLM-as-a-Judge*: (a) Normalized time for each step w/ and w/o MPS. The fifth step is the long round. (b) Normalized time for reward computation with different sequence lengths w/ and w/o pipelined execution. *Code sandbox*: (c) The combined time of rollout and asynchronous reward computation w/ and w/o adaptive timeout.

the reward scheduler for LLM-as-a-Judge and code tasks through microbenchmarks.

GPU Sharing for the Judge LLM. To improve resource utilization, `RollPacker` colocates the judge LLM and the actor LLM on the same GPU. However, naive concurrent execution suffers from severe interference due to resource contention. To mitigate this, we leverage `MPS`. Figure 14a reports the step time with and without `MPS` when training Qwen2.5-7B/8k. `MPS` effectively isolates the concurrent computations, reducing step time and yielding up to a 1.25× speedup. These results demonstrate the effectiveness of `MPS` in GPU sharing.

Pipelined Judge Execution. When colocating the two models, we reserve a strict GPU memory budget for the judge LLM, which requires offloading at least half of its weights to the CPU to perform reward computation for maximum-length responses. While this approach satisfies memory constraints, naive offloading incurs substantial weight transfer overhead. To hide this latency, `RollPacker` employs a layer-wise pipelined execution scheme that overlaps weight transmission with GPU activation computation. Figure 14b compares the reward computation overhead of pipelined versus non-pipelined execution across varying response lengths. As response lengths grow to 32k tokens, which demands larger memory and thus forces more weights to be offloaded, the pipelined scheme yields up to a 1.4× speedup. These results demonstrate that pipelining successfully reduces both memory consumption and reward computation time.

Adaptive Timeout for Code Sandboxing. To alleviate the bottleneck of code sandbox evaluation, `RollPacker` employs an adaptive timeout mechanism. We evaluate this optimization on the Qwen2.5-7B/8k model by comparing the reward overhead under adaptive versus fixed timeouts. Figure 14c illustrates the combined duration of rollout and asynchronous reward computation per step. The adaptive timeout fast-fails doomed execution paths, substantially reducing unnecessary timeout waits during short rounds. During long rounds, the system overlaps prolonged reward computations with the generation of long-tail rollouts. Consequently, the adaptive timeout mechanism drastically improves reward evaluation efficiency, achieving an average speedup of 1.6× across all steps.

6.6 Stream Trainer

We evaluate the effectiveness of the stream trainer using Qwen2.5-7B/8k model on mathematical datasets to isolate its training dynamics.

Adaptive GPU Scaling Criteria. The stream trainer monitors the response length distribution to adaptively determine the optimal moment for GPU scaling. To highlight the advantages of this adaptive approach, we evaluate it against a

Table 3: **[Stream Trainer]** The impact of GPU scaling.

	w/o	20%	30%	40%	Adpt.
Step Time (s)	124.2 1.00×	122.7 1.01×	118.2 1.05×	119.8 1.04×	115.2 1.08×

Table 4: **[Stream Trainer]** The impact of async fetching.

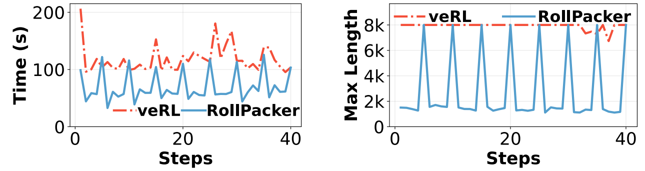
	Stream	20%	30%	40%	50%
Step Time (s)	115.2 1.00×	128.6 0.90×	129.5 0.89×	133.8 0.86×	132.0 0.87×

baseline without GPU scaling, as well as fixed-trigger baselines that initiate scaling only when the fraction of completed prompts reaches 20%, 30%, or 40%. Because short rounds trigger GPU scaling less frequently, we report the average speedup over five long rounds. With asynchronous fetching enabled across all configurations, we observe that even fixed-criteria scaling successfully reduces end-to-end training time. Importantly, the request migration overhead remains under 3 seconds, and the resulting decrease in decoding throughput is strictly contained to within 1%. Furthermore, adaptive GPU scaling outperforms all fixed-criteria baselines, achieving a 1.08× speedup over the baseline without GPU scaling as depicted in Table 3.

Asynchronous Fetching. To compute gradients concurrently with rollout generation, the stream trainer asynchronously fetches completed prompts. To quantify the benefits of this design, we compare it against static baseline approaches that perform a single fetch of all available completed prompts, capped at 20%, 30%, 40%, or 50% of the total batch. Table 4 reports the end-to-end training step time for these fixed fetch ratios versus asynchronous fetching. The stream trainer achieves up to a 14% reduction in end-to-end step time when compared to the static, fixed-size fetching baselines.

6.7 Applicability to Asynchronous Pipelines

While our primary focus is synchronous training, our core optimizations are complementary to asynchronous paradigms (e.g., one-off pipelines [23], partial rollouts [39], or fully asynchronous training [9]) which relax synchronization to improve throughput but often still suffer from the inefficiency of long-tail generation. To demonstrate this, we integrated `RollPacker` with a one-off pipeline training policy—a widely used asynchronous mode where rollouts generated in step t are used for training in step $t+1$. We compared this against a baseline veRL implementation using the same one-off policy on the Qwen2.5-7B model with a maximum sequence length of 8k. As shown in Figure 15a, `RollPacker` achieves a 1.71× average speedup over the one-off veRL baseline. Figure 15b further reveals that `RollPacker` significantly reduces the maximum response length in most rollout steps, confirming that tail batching effectively mitigates long-tail bottlenecks even when synchronization constraints are relaxed. Furthermore, our system-level optimizations, specifically the reward scheduler and stream trainer, remain fully applicable in asynchronous settings, as they independently reduce the latency of the reward and training stages.



(a) Step Time. (b) Max Response Length.

Figure 15: Performance analysis of `RollPacker` with the one-off pipeline [23] and Qwen2.5-7B model. We use eight H800 GPUs for rollout and eight GPUs for training. The batch size is adjusted to 32 accordingly.

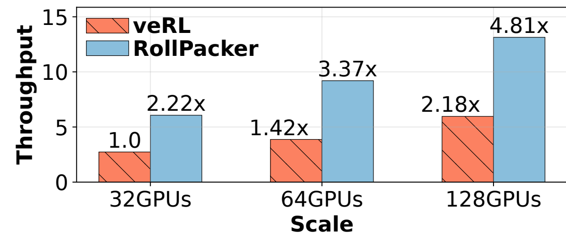


Figure 16: End-to-end throughput (samples/second) of training a Qwen2.5-14B model at scale.

6.8 Scalability

We conduct a scalability analysis for the Qwen2.5-14B/16k. We scale the batch size from 128 to 512 along with the corresponding computational resources. Throughput is measured following [55], defined as the average number of samples processed per second, and is averaged over 20 consecutive training steps. Figure 16 shows that `RollPacker` maintains strong performance at large scale, utilizing up to 128 GPUs. Compared with veRL, `RollPacker` consistently achieves a 2.2× throughput increase. When doubling the resources, `RollPacker` delivers ∼1.5× throughput, as larger global batch sizes increase the computational load and time required for the training stage.

7 Discussion

In this section, we discuss the broader implications and extensibility of `RollPacker`'s design.

General Validity of Offline Profiling. Our offline profiling (§4.1) serves as a foundational reference for online scheduling. While the distribution of prompts and response lengths evolves during training, the fundamental hardware performance characteristics of the LLM, such as memory usage per token and decoding latency per batch, remain stable. Thus,

the initial profile typically remains valid throughout the training lifecycle. To handle potential corner cases where extreme distribution shifts might push the model into an unprofiled performance regime, RollPacker continuously monitors runtime metrics against expected baseline values. If significant discrepancies arise, the system triggers a targeted, lightweight online re-profiling of the relevant configurations. Given its low computational cost (§4.1), even occasional profiling updates introduce minimal overhead (less than 2% of total training time), thereby preserving strict training efficiency.

Extend to Other Policy Optimization Algorithms. Many algorithmic studies [30,49,53] have extended GRPO to improve sample efficiency. A representative variant is DAPO [49], which performs oversampling and discards prompts with zero reward variance. Similar to tail batching, DAPO launches more prompts than the required batch size during rollout. To integrate tail batching with DAPO, we can establish a maximum number of active requests per LLM instance and continuously stream new requests. The termination criteria follow DAPO specifications. Prompts with zero reward variance are excluded from the long-prompt queue, while the remaining unfinished prompts are retained for subsequent processing.

MoE and Agentic RL. RollPacker's design naturally extends to more advanced model architectures and training paradigms. For Mixture-of-Experts (MoE) models, computational variance is compounded by expert routing decisions. To support MoE, the parallelism planner (§4.2) can be extended to dynamically optimize expert parallelism (EP) alongside TP. This requires augmenting the offline profiling phase to characterize the computational cost of different expert routing patterns. By dynamically adjusting EP size or expert placement during rollout, RollPacker could further reduce the specific overheads tied to MoE generation.

Furthermore, in agentic workflows, models frequently interact with external environments, introducing high variance in rollout latency due to unpredictable environment wait times and multi-turn interactions. Recent systems like ROLL-Flash [22] and RollArt [12] have adopted speculative execution to prevent environment failures from stalling entire batches. Tail batching can be applied to isolate these long-latency interaction trajectories into dedicated rounds, preventing them from blocking faster, purely computational rollouts.

8 Related Works

RL Post-Training Frameworks. Many frameworks have been proposed to accelerate RL post-training. Early efforts [15, 18, 21, 47] aim to orchestrate the complex workflow of RL post-training. Later systems introduced architecture optimizations: veRL [34] introduces a hybrid-controller design to improve resource utilization, while DistFlow [42] adopts a multi-controller approach to enhance scalability. Other systems targets specific bottlenecks: RLHFuse [55] fuses the generation and inference stages to reduce training time, and ReaLHF [24] optimizes parallelism strategies to improve system throughput. To mitigate long-tail rollouts, systems like AReaL [9], ROLL-Flash [22], ROME [41], StreamRL [54], and RhymeRL [16] introduce asynchronous RL post-training with tailored system optimizations to increase throughput. RollMux [43] exploits resource bubbles induced by long-tail rollouts, improving resource utilization via multi-tenant scheduling, similar to [11,48]. RollPacker can alleviate long-tail rollouts for a single job in synchronous RL training.

Rollout Optimization. Many recent works aim to optimize the rollout stage of RL post-training to improve training efficiency. For example, DAPO [49] proposes a dynamic sampling technique to filter out prompts with zero reward variance and terminates the rollout stage after collecting enough responses. SPEED-RL [51] estimates the difficulty of each prompt, then selects those with desirable pass rates for further response generation. Similarly, GRESO [53] leverages reward dynamics to remove zero-variance prompts before rollout, while MoPPS [30] models prompt success rates to predict prompt difficulty. These algorithmic techniques expedite model convergence in RL post-training by prioritizing high-quality prompts. In contrast, RollPacker takes a systems-oriented approach that aims to improve raw rollout speed, thus directly reducing end-to-end training latency without altering the underlying prompt distribution.

9 Conclusion

This paper presents RollPacker, a novel RL post-training system designed to expedite synchronous RL training by mitigating the severe GPU underutilization caused by long-tail rollouts. We have proposed tail batching, a scheduling strategy that enhances resource utilization by consolidating long-tail responses into dedicated long rounds, ensuring the majority of rollout steps remain balanced and efficient. In conjunction with tail batching, we have designed a parallelism planner, reward scheduler, and stream trainer that systematically optimize the rollout, reward, and training stages, respectively. Extensive experiments demonstrate the effectiveness of RollPacker in training efficiency, achieving up to a 2.56× end-to-end training time reduction compared to veRL across the Qwen2.5 family of LLMs on up to 128 GPUs.

Acknowledgment

We thank our shepherd, Prof. Arvind Krishnamurthy, and the anonymous reviewers for their valuable comments that help improve the quality of this work. This work was supported in part by the Alibaba Innovative Research (AIR) Grant, RGC CRF Grant (Ref. #C6015-23G), RGC GRF Grant (Ref. #16217124), and NSFC/RGC CRS Grant (Ref. #CRS_HKUST601/24 and Ref. #CRS_PolyU501/23).

References

[1] Chenxin An, Zhihui Xie, Xiaonan Li, Lei Li, Jun Zhang, Shansan Gong, Ming Zhong, Jingjing Xu, Xipeng Qiu, Mingxuan Wang, and Lingpeng Kong. POLARIS: A Post-Training Recipe for Scaling Reinforcement Learning on Advanced Reasoning Models. https://hkunlp.github.io/blog/2025/Polaris, 2025.

[2] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2020.

[3] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML*, 2009.

[4] Ernie Chang, Hui-Syuan Yeh, and Vera Demberg. Does the order of training samples matter? improving neural data-to-text generation with curriculum learning. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume, EACL*, 2021.

[5] Jiaze Chen, Tiantian Fan, Xin Liu, Lingjun Liu, Zhiqi Lin, Mingxuan Wang, Chengyi Wang, Xiangpeng Wei, Wenyuan Xu, Yufeng Yuan, Yu Yue, Lin Yan, Qiying Yu, Xiaochen Zuo, Chi Zhang, Ruofei Zhu, Zhecheng An, Zhihao Bai, Yu Bao, Xingyan Bin, Jiangjie Chen, Feng Chen, Hongmin Chen, Riwei Chen, Liangqiang Chen, Zixin Chen, Jinsong Chen, Siyan Chen, Kaiyuan Chen, Zhi Chen, Jin Chen, Jiecao Chen, Jinxin Chi, Weinan Dai, Ning Dai, Jiahui Dai, Shihan Dou, Yantao Du, Zhengyin Du, Jianhui Duan, Chen Dun, Ting-Han Fan, Jiazhan Feng, Junda Feng, Ziyuan Feng, Yuwei Fu, Wenqi Fu, Hanjie Fu, Hao Ge, Hongyi Guo, Mingji Han, Li Han, Wenhao Hao, Xintong Hao, Qianyu He, Jerry He, Feng He, Wen Heng, Zehua Hong, Qi Hou, Liang Hu, Shengding Hu, Nan Hu, Kai Hua, Qi Huang, Ziyue Huang, Hongzhi Huang, Zihao Huang, Ting Huang, Wenhao Huang, Wei Jia, Bin Jia, Xiaoying Jia, Yuhua Jiang, Haobin Jiang, Ziheng Jiang, Kaihua Jiang, Chengquan Jiang, Jianpeng Jiao, Xiaoran Jin, Xing Jin, Xunhao Lai, Zheng Li, Xiang Li, Liyi Li, Hongkai Li, Zheng Li, Shengxian Wan, Ya Wang, Yunshui Li, Chenggang Li, Niuniu Li, Siyu Li, Xi Li, Xiao Li, Aoyan Li, Yuntao Li, Nianning Liang, and Xinnian Liang. Seed1.5-Thinking: Advancing superb reasoning models with reinforcement learning. *CoRR*, 2025.

[6] DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *CoRR*, 2025.

[7] Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-Group policy optimization for LLM agent training. *CoRR*, 2025.

[8] Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. Reverse curriculum generation for reinforcement learning. In *1st Annual Conference on Robot Learning, CoRL*, 2017.

[9] 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. *CoRR*, 2025.

[10] Shiwei Gao, Youmin Chen, and Jiwu Shu. Fast state restoration in LLM serving with hcache. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys*, 2025.

[11] Wei Gao, Zhisheng Ye, Peng Sun, Tianwei Zhang, and Yonggang Wen. Unisched: A unified scheduler for deep learning training jobs with different user demands. *IEEE Transactions on Computers*, 2024.

[12] 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. RollArt: Scaling agentic rl training via disaggregated infrastructure. *CoRR*, 2025.

[13] Shixiang Gu, Tim Lillicrap, Richard E. Turner, Zoubin Ghahramani, Bernhard Schölkopf, and Sergey Levine. Interpolated policy gradient: Merging on-policy and off-policy gradient estimation for deep reinforcement learning. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, NeurIPS*, 2017.

[14] 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. AsyncFlow: An asynchronous streaming RL framework for efficient LLM post-training. *CoRR*, 2025.

[15] 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. NeMo: A Toolkit for Conversational AI and Large Language Models. https://github.com/NVIDIA/NeMo, 2024.

[16] Jingkai He, Tianjian Li, Erhu Feng, Dong Du, Qian Liu, Tao Liu, Yubin Xia, and Haibo Chen. History rhymes: Accelerating LLM reinforcement learning with RhymeRL. *CoRR*, 2025.

[17] Zhiwei He, Tian Liang, Jiahao Xu, Qiuzhi Liu, Xingyu Chen, Yue Wang, Linfeng Song, Dian Yu, Zhenwen Liang, Wenxuan Wang, Zhuosheng Zhang, Rui Wang, Zhaopeng Tu, Haitao Mi, and Dong Yu. DeepMath-103K: A large-scale, challenging, decontaminated, and verifiable mathematical dataset for advancing reasoning. *CoRR*, 2025.

[18] Jian Hu, Xibin Wu, Weixun Wang, Xianyu, Dehao Zhang, and Yu Cao. OpenRLHF: An easy-to-use, scalable and high-performance RLHF framework. *CoRR*, 2024.

[19] Minghui Jia. AIME 2024 Dataset. Hugging Face, https://huggingface.co/datasets/Maxwell-Jia/AIME_2024, 2025.

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph 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*, 2023.

[21] Kinman Lei, Yuyang Jin, Mingshu Zhai, Kezhao Huang, Haoxing Ye, and Jidong Zhai. PUZZLE: efficiently aligning large language models through light-weight context switch. In *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC*, 2024.

[22] 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. Part II: ROLL flash - accelerating RLVR and agentic training with asynchrony. *CoRR*, 2025.

[23] Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. DeepCoder: A fully open-source 14b coder at o3-mini level. https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-mini-Level-1cf81902c14680b3bee5eb349a512a51, 2025.

[24] Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. ReaLHF: Optimized RLHF training for large language models through parameter reallocation. *CoRR*, 2024.

[25] NVIDIA Corporation. NVIDIA Multi-Process Service (MPS) Documentation. https://docs.nvidia.com/deploy/mps/index.html, 2024.

[26] NVIDIA Corporation. Cuda driver api – context management (green contexts). https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__GREEN__CONTEXTS.html, 2025.

[27] Open-R1. Codeforces Dataset. Hugging Face, https://huggingface.co/datasets/open-r1/codeforces, 2025.

[28] OpenAI. Introducing openai o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/, 2024.

[29] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. In *Forty-second International Conference on Machine Learning, ICML*, 2025.

[30] Yun Qu, Qi Wang, Yixiu Mao, Vincent Tao Hu, Björn Ommer, and Xiangyang Ji. Can prompt difficulty be online predicted for accelerating RL finetuning of reasoning models? *CoRR*, 2025.

[31] Qwen Team. QwQ-32B: Embracing the power of reinforcement learning. https://qwenlm.github.io/blog/qwq-32b/, 2025.

[32] SGLang Team. SGLang: Fast serving framework for large language models. https://github.com/sgl-project/sglang, 2025.

[33] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, 2024.

[34] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. verl: Volcano engine reinforcement learning for LLM. https://github.com/volcengine/verl, 2024.

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

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

[37] Guijin Son, Hyunwoo Ko, Hoyoung Lee, Yewon Kim, and Seunghyeok Hong. LLM-as-a-Judge & reward model: What they can and cannot do. *CoRR*, 2024.

[38] Petru Soviany, Radu Tudor Ionescu, Paolo Rota, and Nicu Sebe. Curriculum learning: A survey. *Int. J. Comput. Vis.*, 2022.

[39] Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, Chuning Tang, Congcong Wang, Dehao Zhang, Enming Yuan, Enzhe Lu, Fengxiang Tang, Flood Sung, Guangda Wei, Guokun Lai, Haiqing Guo, Han Zhu, Hao Ding, Hao Hu, Hao Yang, Hao Zhang, Haotian Yao, Haotian Zhao, Haoyu Lu, Haoze Li, Haozhen Yu, Hongcheng Gao, Huabin Zheng, Huan Yuan, Jia Chen, Jianhang Guo, Jianlin Su, Jianzhou Wang, Jie Zhao, Jin Zhang, Jingyuan Liu, Junjie Yan, Junyan Wu, Lidong Shi, Ling Ye, Longhui Yu, Mengnan Dong, Neo Zhang, Ningchen Ma, Qiwei Pan, Qucheng Gong, Shaowei Liu, Shengling Ma, Shupeng Wei, Sihan Cao, Siying Huang, Tao Jiang, Weihao Gao, Weimin Xiong, Weiran He, Weixiao Huang, Wenhao Wu, Wenyang He, Xianghui Wei, Xianqing Jia, Xingzhe Wu, Xinran Xu, Xinxing Zu, Xinyu Zhou, Xuehai Pan, Y. Charles, Yang Li, Yangyang Hu, Yangyang Liu, Yanru Chen, Yejie Wang, Yibo Liu, Yidao Qin, Yifeng Liu, Ying Yang, Yiping Bao, Yulun Du, Yuxin Wu, Yuzhi Wang, Zaida Zhou, Zhaoji Wang, Zhaowei Li, Zhen Zhu, Zheng Zhang, Zhexu Wang, Zhilin Yang, Zhiqi Huang, Zihao Huang, Ziyao Xu, and Zonghan Yang. Kimi k1.5: Scaling Reinforcement Learning with LLMs. *CoRR*, 2025.

[40] 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. *CoRR*, 2025.

[41] 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. Let it flow: Agentic crafting on rock and roll, building the rome model within an open agentic learning ecosystem. *CoRR*, 2026.

[42] Zhixin Wang, Tianyi Zhou, Liming Liu, Ao Li, Jiarui Hu, Dian Yang, Jinlong Hou, Siyuan Feng, Yuan Cheng, and Yuan Qi. DistFlow: A fully distributed RL framework for scalable and efficient LLM post-training. *CoRR*, 2025.

[43] 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. RollMux: Phase-level multiplexing for disaggregated rl post-training. *CoRR*, 2025.

[44] 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, QingKai 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. MiMo: Unlocking the reasoning potential of language model - from pretraining to posttraining. *CoRR*, 2025.

[45] Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. KodCode: A diverse, challenging, and verifiable synthetic dataset for coding. In *Findings of the Association for Computational Linguistics, ACL*, 2025.

[46] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong

Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 Technical Report. *CoRR*, 2024.

[47] 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. *CoRR*, 2023.

[48] Zhisheng Ye, Peng Sun, Wei Gao, Tianwei Zhang, Xiaolin Wang, Shengen Yan, and Yingwei Luo. Astraea: A fair deep learning scheduler for multi-tenant gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[49] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, 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, Weinan Dai, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: an open-source LLM reinforcement learning system at scale. *CoRR*, 2025.

[50] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 2016.

[51] Ruiqi Zhang, Daman Arora, Song Mei, and Andrea Zanette. SPEED-RL: faster training of reasoning models via online curriculum learning. *CoRR*, 2025.

[52] Yiqi Zhang, Huiqiang Jiang, Xufang Luo, Zhihe Yang, Chengruidong Zhang, Yifei Shen, Dongsheng Li, Yuqing Yang, Lili Qiu, and Yang You. SortedRL: Accelerating RL training for LLMs through online length-aware scheduling. In *ES-FoMo III: 3rd Workshop on Efficient Systems for Foundation Models*, 2025.

[53] Haizhong Zheng, Yang Zhou, Brian R. Bartoldson, Bhavya Kailkhura, Fan Lai, Jiawei Zhao, and Beidi Chen. Act only when it pays: Efficient reinforcement learning for LLM reasoning via selective rollouts. *CoRR*, 2025.

[54] 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. StreamRL: Scalable, heterogeneous, and elastic RL for LLMs with disaggregated stream generation. *CoRR*, 2025.

[55] 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*, 2025.

[56] 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. *CoRR*, 2025.