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

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 often suffers from significant GPU underutilization, referred to as "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 that systematically consolidates prompts leading to long-tail responses into a small subset of rollout steps (*long rounds*), while ensuring that the majority of steps (*short rounds*) involve only balanced, short rollouts. By excluding long responses from short rounds and rescheduling them into a few designated long rounds, tail batching effectively reduces GPU idle time during rollouts and significantly accelerates RL training without sacrificing accuracy. We present Flexes, 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. Empirical results show that Flexes achieves a 2.03×-2.56× end-to-end training time reduction compared to veRL [45], and up to 2.24× speedup compared to RLHFuse [63] for the Qwen2.5 family of LLMs on up to 128 H800 GPUs.

1 Introduction

Advanced Large Language Models (LLMs) [1, 3, 11, 40] critically rely on Reinforcement Learning (RL) post-training to enhance reasoning capabilities in complex tasks, such as mathematics [2], code generation [34], and tool use [12, 35]. The standard RL post-training workflow for LLM reasoning models comprises repeated cycles across three stages [11, 43]: *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 [47] 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 [21, 29, 44, 51], 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 are completed.

A common approach to mitigating idle bubbles is to overlap the long rollout stage with reward computation (e.g., ROLL [50] and MiMO [52]) and reference model inference (e.g., RLHFuse [63]). 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 DeepScaler [28], StreamRL [62], and AsyncFlow [17], wherein rollouts generated in a previous step are used for subsequent training. Some frameworks, such as AReaL [14], even adopt fully asynchronous RL training that continuously performs rollouts and training in parallel. Although these approaches 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

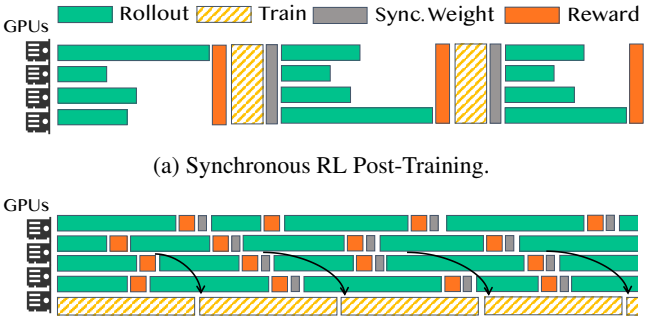
of balanced, short responses, thereby reducing idle bubbles in GPU utilization. Importantly, because tail batching alters only the order of training samples, it preserves training accuracy, as indicated by recent algorithmic research [37, 57, 59].

To implement this approach, we present `Flexes`, a system engineered to unlock the full potential of tail batching for on-policy RL training. `Flexes` initiates rollout in a *short round* to sample P_0 prompts, each producing R_0 responses. To collect balanced, short responses, `Flexes` 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, `Flexes` batch-executes them in a dedicated *long round*, without speculative execution.

`Flexes` 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 [11]. A fixed parallelization scheme cannot accommodate this variability. To address this, `Flexes` 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, `Flexes` 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 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, `Flexes` 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, `Flexes` 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.



(a) Synchronous RL Post-Training.

(b) Asynchronous RL Post-Training.

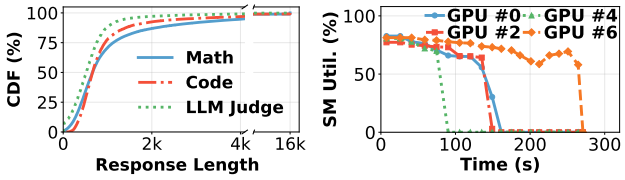
Figure 1: Execution workflows of synchronous and asynchronous RL post-training.

We implemented `Flexes` in 6.6k lines of Python code atop an in-house RL framework. We train models from the Qwen2.5 family (7B–32B) [38] with real-world datasets [20, 54] on a cluster of 128 H800 GPUs using `Flexes`. Evaluation results show that `Flexes` substantially outperforms state-of-the-art RL systems, achieving 2.03×-2.56× end-to-end training speedup over veRL [45] and up to 2.24× speedup compared to RLHFuse [63].

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 [43] are highly effective in enhancing reasoning capabilities across various domains. An RL 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 [47] 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



(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 [45] and GRPO [43] with real-world datasets [20, 54] in three tasks.

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

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 [16]. 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 [38], configured with a maximum response length of 16k tokens, a batch size of 128, and a group size of 8 under the GRPO algorithm [43]. We run math, code, and LLM-as-a-Judge tasks with real-world datasets [20, 54] using veRL [45] 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 [14, 17, 19, 63].

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 [63] overlaps rollout with reward computation and reference model inference, while frameworks like ROLL [50] and MiMO [52] 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 [4], the relative contribution of reward and reference inference diminishes (typically less than 15% of step runtime as reported in Table 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 [48] introduces *partial rollout* by truncating the long-tail responses and preserving generated tokens to continue rollouts in subsequent steps. StreamRL [62], AsyncFlow [17], and RhymeRL[1] [19] allow a one-step staleness, enabling the training stage to proceed with slightly outdated rollouts in a *one-off pipeline*. Pushing further, AReaL [14] introduces *fully asynchronous* RL training, in which rollout and training are completely decoupled,

[1] RhymeRL's HistoPipe scheduling requires one-step off-policyness (see Figure 10 and evaluation in §7.3 of [19]).

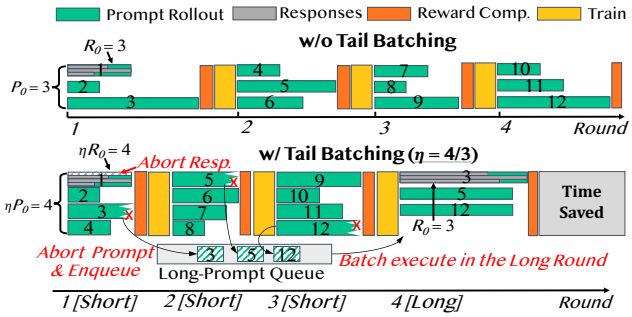
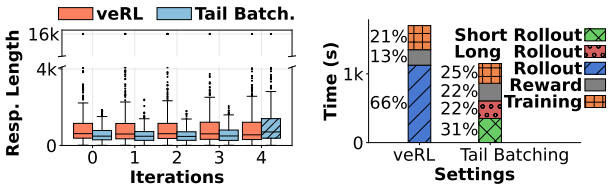


Figure 3: Illustration of Tail Batching.



(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 [54]. (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.

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 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-trail 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* [58] by over-provisioning requests while selectively retaining only the fastest completions. Under the GRPO algorithm [43], 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 only reorders short and long prompts into separate rounds without altering the underlying sample distribution or relaxing synchronization. Prior studies show that changes in training order do not degrade model accuracy [7,9,13]. In fact, several recent RL post-training algorithms explicitly explore prompt reordering as a means of efficiency [37,57,59]. Our empirical results further validate this claim: as shown in Figure 8, tail batching achieves accuracy curves nearly identical to those of standard synchronous RL training.

Rollout Efficiency. We empirically validate tail batching's effectiveness in enhancing rollout efficiency by training a Qwen2.5-14B model [38] on the code dataset [54] 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 [45]), tail batching yields shorter and more balanced responses in the first four steps (short 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 be-

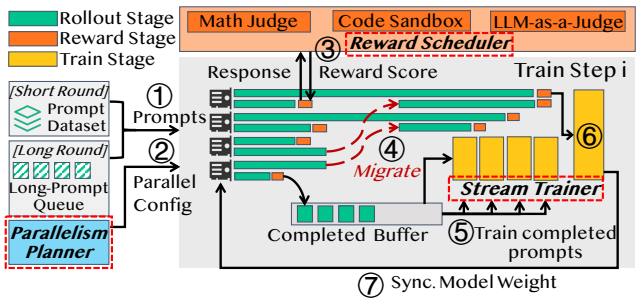


Figure 5: System overview and workflow of `Flexes`.

come 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.

4 `Flexes` System Design

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

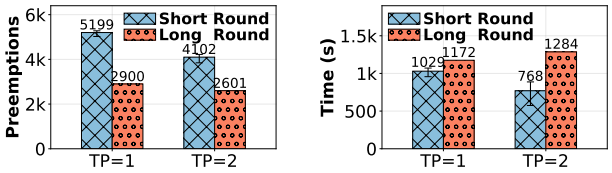
4.1 System Overview

`Flexes` 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. `Flexes` 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, `Flexes` 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 [50, 63] 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.



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

Figure 6: 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.

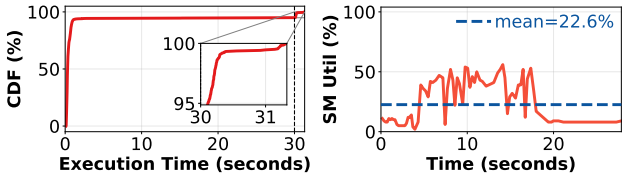
Workflow. `Flexes` operates in two phases: an *offline profiling phase* and *online execution phase*. In the **offline phase**, `Flexes` benchmarks the actor LLM's prefilling and decoding throughput under different TP sizes, batch sizes, and sequence lengths. It also profiles the GPU memory footprint and runtime cost of the judge LLM across varying sequence lengths. These profiled results are used for guiding online decisions.

In the **online phase**, `Flexes` orchestrates rollout, reward, and training in a synchronous RL job (Figure 5). 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.

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 [24, 42] 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 6a 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 6a, configuring a larger TP size partitions model weights across more GPUs, free-



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

Figure 7: 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.

ing 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 6b, 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 [21, 45] adopt a fixed parallelism configuration, which our analysis shows is inefficient for tail batching (Figure 6). Optimal TP sizes differ between short and long rounds, necessitating dynamic adaptation. `Flexes` 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 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, `Flexes` 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 [50] and MiMo [52]. 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 7a, 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.

`Flexes` 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 total GPUs) *exclusively for the judge LLM* to avoid interference with other workers. However, this strategy results in poor utilization. As shown in Figure 7b, 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, `Flexes` *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, `Flexes` enables *Multi-Process Service* (MPS) [33], which partitions GPU resources at the warp level and allows both models to run concurrently with minimal interference.

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, `Flexes` introduces a *layer-wise pipeline scheme* that reduces the memory footprint of the judge LLM. Inspired by prior work [6], 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. `Flexes` dynamically adjusts the number of offloaded layers to accommodate varying input sequence lengths, ensuring the judge LLM fits within memory while maximizing utilization.

Algorithm 1 Stream Trainer

```
1:  Input: Requests R, GPUs G
2:  procedure StreamTrainer(R, G)
3:      G_rollout ← G;    G_train ← ∅
4:      R_run ← R;    R_comp ← []
5:      scaled_down ← false
6:      Δ_R ← 0
7:      while |R_run| ≠ 0 do
8:          R_fin ← LLM.generate(R_run, G_rollout)
9:          Δ_R ← Δ_R + |R_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 ≤ |R_comp|/|R| ≤ 0.5 and Δ_R/|R| ≥ 0.05 then
15:                 Δ_R ← 0
16:                 G_free ← PickScaleDownGPUs(G)
17:                 if MeetScaleCriteria(G_free) then
18:                     G_rollout ← G_rollout \ G_free
19:                     G_train ← G_free
20:                     MigrateRequests(G_free, G_rollout)
21:                     scaled_down ← true
22:         if scaled_down then
23:             ComputeGrad(G_train, R_comp)
```

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, `Flexes` 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 [15, 19]. 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 [46] 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 gradient computation and model updates. Since some replicas may have already processed part of the workload, a naive averaging would bias the result. To correct this, we re-normalize local gradients on each replica by the number of samples it processed, ensuring the final update is equivalent to that of standard on-policy training.

5 Implementation

We implemented `Flexes` in ∼6.6k lines of Python code on top of an in-house RL framework, which will be open-sourced. The system integrates existing LLM infrastructure

with lightweight extensions for rollout, reward, and training.

Rollout Stage. `Flexes` uses vLLM v0.8.4 [24] 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 flexibly 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 [46], 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 `Flexes` on Qwen2.5 models with 7B-32B parameters using a diverse benchmark of real-world datasets. In §6.1, we compare `Flexes` against existing RL post-training systems in terms of validation accuracy and training time. We then break down performance across pipeline stages in §6.2 and microbenchmark the three optimization designs in §6.3-§6.6. §6.7 presents a scalability analysis.

Cluster Setup. We deploy `Flexes` on an H800 cluster with 16 nodes (128 GPUs total), connected via 400 Gbps InfiniBand.

Models. We use the Qwen2.5 [5] family with 7B, 14B, and 32B parameters, configured with maximum response lengths of 8k, 16k, and 32k tokens, respectively. End-to-end evaluation is conducted in a multi-task setting with a uniform mix of datasets spanning mathematics [20], code generation [54], and multi-subject question answering, using rule-based, code sandbox, and LLM-as-a-Judge reward workers, respectively.

Training Configurations. Unless otherwise noted, we adopt synchronous RL training with $P_0 = 128$ and $R_0 = 8$. Actor and reference models are of the same size, and Qwen2.5-7B-Instruct is used as the judge. Parallelism strategies and resource allocations vary with model size. The 7B, 14B, and 32B models are trained on 16, 32, and 64 GPUs. Their rollout TP is set to 1, 2, and 2, while training configurations (TP, PP, CP) are (2,1,1), (2,2,2), and (4,1,4), respectively.

Metrics. We report validation accuracy across training steps and measure end-to-end training time to evaluate both effectiveness and efficiency.

6.1 End-to-End Evaluation

We compare the end-to-end performance of `Flexes` with state-of-the-art synchronous RL post-training systems.

Table 2: End-to-end training speedup breakdown.

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	2.01×	1.99×	2.48×
+ Parallelism	2.01×	2.02×	2.52×
+ Trainer	2.03×	2.22×	2.56×

- **veRL** [45] proposes a hybrid programming model for the RL post-training pipeline and provides a optimized 3D-HybridEngine to improve the rollout and training efficiency.
- **RLHFuse** [63] pipelines the reward and reference model inference with the rollout stage. We strength RLHFuse with *stream trainer* and *asynchronous reward computation*.

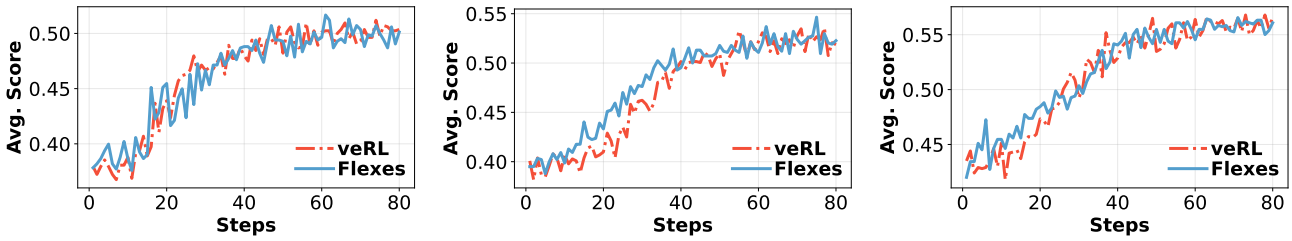
Validation Performance. Figure 8 presents the average validation scores of `Flexes` and veRL, demonstrating that tail batching does not compromise training accuracy across different model sizes and response lengths. Moreover, `Flexes` achieves faster convergence at early stage, and we hypothesize it is due to the more balanced response length distribution.

End-to-End Latency. Figure 9 reports the training step time of each model in the first 40 steps for a clear illustration. Overall, `Flexes` outperforms veRL and RLHFuse across all three LLMs. Compared with veRL, `Flexes` achieves speedups of 2.03×, 2.22×, and 2.56× for three LLMs, respectively. Against RLHFuse, the speedups are 1.14×, 1.68×, and 2.24×. Owing to the reward scheduler and stream trainer, both `Flexes` and RLHFuse maintain advantages over veRL in long rounds. However, the overlapping benefits diminish as response length increases (see Figure 9c), since the proportion of rollout time grows in long rounds. In short rounds, `Flexes` achieves 2.1×-3.6× speedup over veRL and 1.2×-3.2× speedup over RLHFuse, owing to tail batching.

6.2 Performance Breakdown

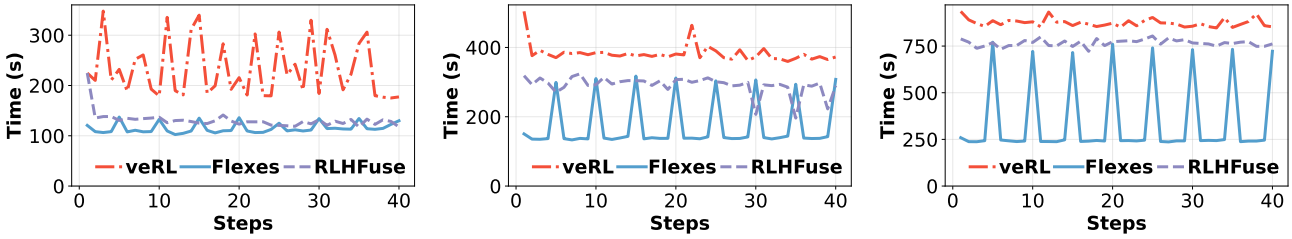
Improvement Breakdown. Table 2 presents a detailed breakdown of the cumulative speedup from our proposed techniques across different model sizes and response lengths.

- The tail batching effectively reduces rollout overhead, and its benefits become more pronounced as response length increases. In particular, `Flexes` achieves up to 2.21× speedup for Qwen2.5-32B/32k setting.
- The reward scheduler is particularly beneficial for short rollouts, providing a 71% performance uplift (from 1.30× to 2.01×) with an 8k response length. It also retains its advantages for longer responses, contributing a 27% improvement (from 2.21× to 2.48×) at a 32k response length.
- The parallelism planner is effective under high memory pressure, a scenario typical for large models and long sequences. In the Qwen2.5-32B/32k setting, it provides an additional 4% speedup (from 2.48× to 2.52×).



(a) Qwen2.5-7B (b) Qwen2.5-14B (c) Qwen2.5-32B

Figure 8: The average validation score for training Qwen2.5-7B, 14B and 32B model with veRL and `Flexes`.



(a) Qwen2.5-7B (b) Qwen2.5-14B (c) Qwen2.5-32B

Figure 9: The step time for training Qwen2.5-7B, 14B and 32B model with veRL, RLHFuse and `Flexes`.

- The stream trainer reduces step time by overlapping rollout and gradient computation. It delivers a substantial 20% performance improvement (from 2.02× to 2.22×) for the Qwen2.5-14B/16k, where rollout and training time are well balanced for pipelining. For other settings, the stream trainer consistently yields positive speedup gains.

Training Step Breakdown. Figure 10 breaks down the training time to analyze the performance of `Flexes`. Figures 10a and 10b compare `Flexes` against veRL on maximum response length and rollout time. While rollout times are comparable in long rounds, `Flexes` demonstrates a significant advantage in short rounds. It substantially reduces the maximum response length (Figure 10a), leading to up to a 7.8× speedup in average rollout time (Figure 10b). The aggregated step time breakdown is shown in Figure 10c, with hatched bars for short rounds and solid bars for long rounds. In long rounds, the rollout stage progressively dominates the total step time. In contrast, the time savings from the shorter rollouts become more significant, effectively lowering the average step time across all rounds. Next, we investigate each system component and quantify its individual contribution under various conditions. The results of these microbenchmarks are presented from §6.3 to §6.6.

6.3 Sensitivity Analysis of Tail Batching

Figure 11 shows the rollout time of different configurations of tail batching. We first fix the number of prompts to P_0 and set different η for the number of responses per prompt R. We then fix the number of responses per prompt and discover the impact of R. We compare them with our chosen configuration of $\eta = 1.25$ in `Flexes`.

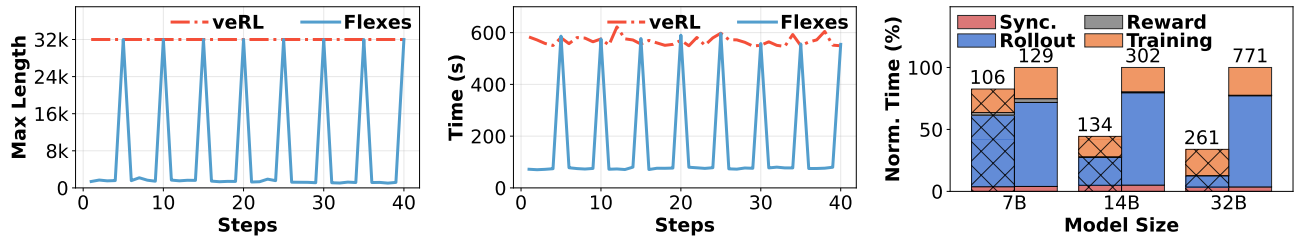
Impact of R. With a fixed $P = P_0$, we increase η for R from 1.0 to 1.5. As the number of responses per prompt increases, we can discard long-tail responses to reduce rollout time. However, some difficult prompts consistently produce long responses. yield long responses. A substantial reduction in rollout time is observed only when η is increased to 1.5.

Impact of P. With a fixed $R = R_0$, we increase the number of prompts P. We collect the first first P_0 prompts and drops the remaining prompts with long responses to the long-prompt queue. When we increase η for P, the frequency of long rounds increases accordingly, which negates the benefits of time reduction from short rounds. For example, with a response length of 32k, we observe that the average rollout time increases when η is raised from 1.25 to 1.5.

Based on the above analyses of P and R, we fix $\eta = 1.25$ for both. Under this setting, tail batching improves the average rollout speed by up to 3.9× and outperforms the fixed-P_0 and fixed-R_0 settings by up to 1.5× and 1.6×, respectively.

6.4 Parallelism Planner

Dynamic TPs in LLM Generation. The parallelism planner adaptively adjusts the TP size for LLM generation in the rollout stage based on the response length distribution. To analyze this behavior, we measure the rollout time of training Qwen2.5-14B on 16 GPUs across training steps. Initially, we fix the TP size to 1, then gradually increase the response length from 8k to 32k by 1k per training step. We present the average rollout time (left) and the optimal TP size (right)



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

Figure 10: Breakdown of training step time in `Flexes`. (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.

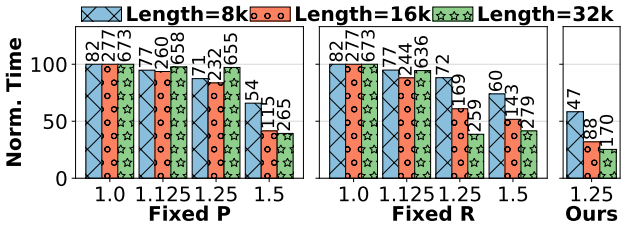


Figure 11: **[Tail Batching]** The rollout time of different configurations. We set P and R fixed to P_0, and R_0, respectively, while changing η for the another parameter. Each bar represents the average iteration time of a full period of consecutive short and long rounds, normalized to the baseline without tail batching, and is annotated with the actual time.

in Figure 12a. Specifically, at steps 3 and 8, the parallelism planner increases the TP size to 2 and 4, respectively, to reduce rollout time. We also compare the iteration time when changing the TP size (blue solod line) versus keeping it fixed (red dashed line), and observe a clear reduction in rollout time due to adaptive TP selection. Overall, the parallelism planner achieves an average 1.9× speedup compared to a baseline with a fixed TP size as 1.

Preemptions and Rollout Latency. Figures 12b-12c show the number of preemptions and the rollout time per step, with and without the parallelism planner, when the maximum response length is fixed at 32k and the initial TP size is set to 2. Both the preemption count and rollout time are normalized to the values obtained without the parallelism planner. As shown in Figure 12b, the parallelism planner reduces the preemption count in short rounds by an average of 13.8%. Figure 12c shows that the parallelism planner can speedup the rollout time in short rounds by 1.11×-1.28×.

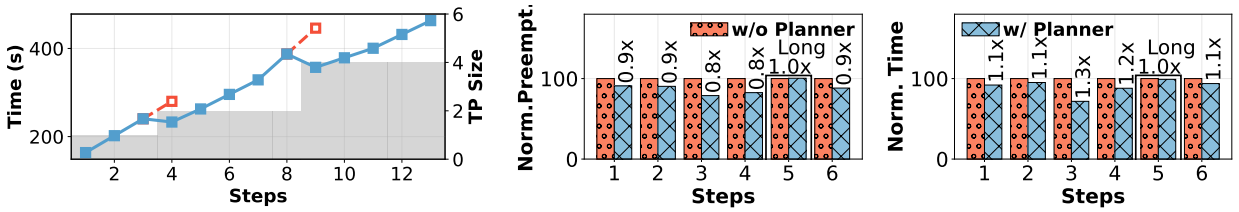
6.5 Reward Scheduler

Compared with synchronous reward computation, asynchronous reward computation yields speedups of 1.48×, 1.35×, and 1.18× on three LLMs, respectively. We analyze the reward scheduler for LLM-as-a-Judge and code tasks. To isolate and evaluate its effectiveness, each subsequent experiment is conducted on a specific task rather than a mixture.

GPU Sharing in Judge LLM. In the reward scheduler, we colocate the judge LLM and actor LLM on the same GPU to improve the resource utilization. However, concurrent executions on a single GPU can suffer interference due to contention for shared computational resources, resulting in degraded performance. To mitigate this, we leverage `MPS`. Figure 13a reports the step time with and without `MPS` when training Qwen2.5-7B/8k. We observe that `MPS` consistently reduces step time, yielding up to a 1.25× speedup. These results demonstrate the effectiveness of `MPS` in GPU sharing.

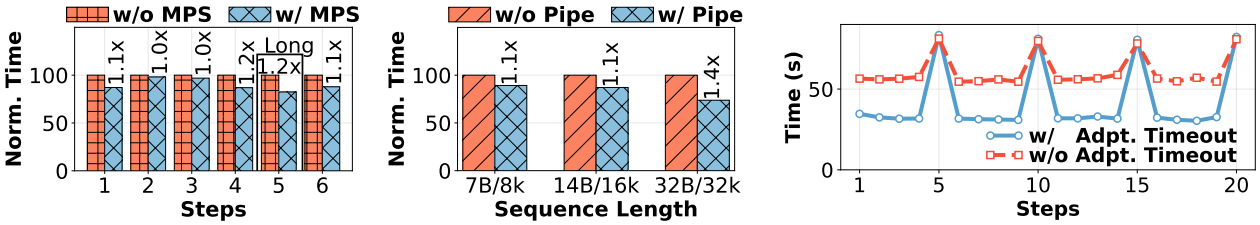
Pipelined Judge LLM Execution. We employ a layer-wise pipelined execution scheme that offloads the weights of the judge LLM to CPU memory and overlaps weight transmission with GPU activation computation. When colocating the judge LLM and actor LLM, we reserve GPU memory for the judge LLM, which requires offloading at least half of its weights to the CPU in order to perform reward computation for responses of maximum length. Although the layer-wise scheme enables execution of the judge LLM under memory constraints, it incurs substantial weight transmission overhead. Figure 13b compares pipelined and non-pipelined execution in terms of reward computation overhead across different sequence lengths. Pipelined execution yields up to a 1.4× speedup when the maximum response length reaches 32k tokens, as the larger activations demand more GPU memory and necessitate offloading more LLM weights. Overall, these results demonstrate that pipelining simultaneously reduces memory consumption and reduces reward computation time.

Adaptive Timeout for Code. we employ an adaptive timeout to alleviate the code sandbox execution overhead. We compare reward computation overhead under adaptive and fixed timeout with Qwen2.5-7B/8k. Figure 13c shows the combined duration of rollout and asynchronous reward computation at each step. The adaptive timeout substantially reduces the unnecessary timeouts in the short rounds. In the long round, the



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

Figure 12: **[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 13: **[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.

reward computation can be overlapped with long-tail rollouts. These results highlight the effectiveness of the adaptive timeout in improving reward computation efficiency, achieving an average speedup of 1.6× across all steps.

6.6 Performance Analysis of Stream Trainer

We evaluate the effectiveness of the stream trainer using Qwen2.5-7B/8k on mathematical datasets for simplicity.

Adaptive Criteria for GPU Scaling. The stream trainer monitors the response length distribution and decides when to perform GPU scaling adaptively. To highlight the advantages of this adaptive criterion, we use a baseline without GPU scaling and fixed-trigger baselines where scaling is applied when the number of completed prompts reaches 20%, 30%, or 40%. Since short rounds involve less frequent GPU scaling, we report the average speedup over five long rounds. Asynchronous fetching is enabled for all GPU scaling baselines. We observe that even with fixed criteria, GPU scaling reduces end-to-end training time and the migration overhead does not exceed 3 seconds, and GPU scaling decreases the decoding throughput ranges 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. The stream trainer asynchronously fetches completed prompts from the rollout stage to compute gradients. To highlight the benefits of this streaming behavior, we compare it against baseline approaches that fetch all available completed prompts only once, with the number of fetched samples capped at 20%, 30%, 40%, or 50% of the total. Table 4 reports the end-to-end training step time for different fixed fetch ratios versus asynchronous prefetching. Compared to fixed-size fetching, the stream trainer achieves up to a 14% reduction in end-to-end step time.

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

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

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

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

6.7 Performance 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 [63], defined as the average number of samples pro-

cessed per second, and is averaged over 20 consecutive training steps. Figure 14 shows that Flexes maintains strong performance at large scale, utilizing up to 128 GPUs. Compared with veRL, Flexes consistently achieves a 2.2× throughput increase. When scaling up resources by 2×, Flexes delivers ∼1.5× throughput improvement, with the smaller gain attributed to the increased training time for larger batch sizes.

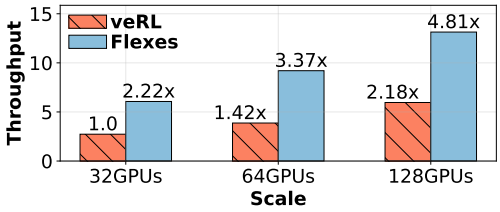


Figure 14: End-to-end throughput (samples/second) of training a Qwen2.5-14B model at scale.

7 Discussion

Here, we discuss how Flexes's design benefits to other policy optimization algorithms and off-policy RL algorithms. We also discuss the potential limitations of Flexes.

Extend to Other Policy Optimization Algorithms. Many algorithmic studies [37, 57, 61] have extended GRPO to improve sample efficiency. A representative variant is DAPO, which performs oversampling and discards prompts with zero reward variance. Similar to tail batching, DAPO launches more prompts than the batch size during the rollout stage. To integrate tail batching with DAPO, we set a maximum number of active requests for each LLM instance and continuously issue new requests. The termination criteria follow DAPO specifications. Prompts with zero reward variance are excluded from the long-prompt queue, while other unfinished prompts are retained in the queue for subsequent processing.

Extend to Asynchronous Systems. In asynchronous off-policy post-training, synchronous RL training is not required to optimize the samples in the long-prompt queue. We can instead leverage existing off-policy algorithms, including one-off pipeline [28], partial rollouts [48], and fully asynchronous training [14], to process the prompts in the queue. For example, rollouts for unfinished prompts can simply be continued in the next training step. In addition, the reward scheduler and stream trainer can reduce the overhead of reward computation and LLM training for asynchronous training, respectively.

Potential Limitations. We utilize MPS [33] to enable spatial GPU sharing, which does not guarantee error isolation. As future work, we plan to explore Green Contexts [32] to improve fault tolerance. The parallelism planner currently focuses only on TP and does not optimize for expert parallelism. Once expert parallelism is enabled, the optimization space can be further expanded.

8 Related Works

RL Post-training Frameworks. Many frameworks have been proposed to accelerate RL post-training. Early efforts [18, 18, 21, 25, 55] aim to orchestrate the complex workflow of RL post-training. Later, veRL [45] introduces a hybrid-controller design to improve resource utilization, while DistFlow [51] adoptes a multi-controller approach to enhance scalability. RLHFuse [63] fuses the generation and inference stages to reduce training time, and Realhf [29] optimizes parallelism strategies to improve system throughput. To mitigate long-tail rollouts, AReal [14], StreamRL [62], and RhymeRL [19] introduce asynchronous RL post-training with tailored system optimizations to increase throughput. Unlike existing RL frameworks, Flexes can alleviate long-tail rollouts even in synchronous RL training.

LLM Training and Inference. LLM training necessitates a range of parallelism strategies. Data parallelism (DP) [36, 39, 41, 60] replicates model weights across GPUs and partitions training samples among model replicas. Tensor parallelism (TP) [8,46,49,53] partitions computation within a model layer, while pipeline parallelism (PP) [22, 30] partitions the model across layers. Context parallelism (CP) [23, 26, 27, 31] partitions input sequences and requires dedicated attention-layer optimizations. In LLM inference, TP and DP are typically used to reduce latency overhead. Many inference optimization techniques focus on accelerating attention computation [10, 24] and optimizing KV cache management [24, 42, 56]. These strategies reduce memory consumption from different dimensions and achieve significant throughput improvements.

Rollout Optimization. Many recent works aim to optimize the rollout stage of RL post-training to improve training efficiency. DAPO [57] proposes a dynamic sampling technique to filters out prompts with zero reward variance and terminates the rollout stage after collected enough responses. SPEED-RL [59] estimates the difficulty of each prompt, then selects those with desirable pass rates for further response generation. GRESO [61] leverages reward dynamics to remove zero-variance prompts before rollout, while MoPPS [37] models prompt success rates to predict prompt difficulty. These techniques expedite the model convergence in RL post-training by prioritizing high-quality prompts. Flexes aims to improve the rollout speed, thus reducing end-to-end training latency.

9 Conclusion

This paper presents Flexes, a novel RL post-training system designed to expedite synchronous RL training. We propose *tail batching* to alleviate long-tail rollouts and enhance resource utilization. In conjunction with the tail batching, we design parallelism planner, reward scheduler, and stream trainer that optimize the rollout, reward, and training stages respectively. Extensive experiments demonstrate the effectiveness of Flexes in training efficiency against baselines.

References

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

[2] AIME 2024 Dataset, 2025.

[3] Qwq-32b: Embracing the power of reinforcement learning. https://qwenlm.github.io/blog/qwq-32b/, 2025.

[4] 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, 2025.

[5] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[6] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: fast pipelined context switching for deep learning applications. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[7] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09. ACM, 2009.

[8] Zhengda Bian, Qifan Xu, Boxiang Wang, and Yang You. Maximizing parallelism in distributed training for huge neural networks. *arXiv preprint arXiv:2105.14450*, 2021.

[9] 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, Online, April 19 - 23, 2021*, 2021.

[10] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[11] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[12] Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for llm agent training. *arXiv preprint arXiv:2505.10978*, 2025.

[13] 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, Mountain View, California, USA, November 13-15, 2017, Proceedings*, Proceedings of Machine Learning Research. PMLR, 2017.

[14] 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, 2025.

[15] Shiwei Gao, Youmin Chen, and Jiwu Shu. Fast state restoration in llm serving with hcache, 2024.

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

[17] 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, 2025.

[18] 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, 2025.

[19] 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, 2025.

[20] Zhiwei He, Tian Liang, Jiahao Xu, Qiuzhi Liu, Xingyu Chen, Yue Wang, Linfeng Song, Dian Yu, Zhenwen Liang, Wenxuan Wang, et al. Deepmath-103k: A large-scale, challenging, decontaminated, and verifiable mathematical dataset for advancing reasoning. *arXiv preprint arXiv:2504.11456*, 2025.

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

[22] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[23] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5, 2023.

[24] 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 ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[25] 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 *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 127–140, 2024.

[26] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2391–2404, 2023.

[27] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *International Conference on Learning Representations(ICLR)*, 2024.

[28] Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Y. Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Li Erran Li, Raluca Ada Popa, and Ion Stoica. Deepscaler: Surpassing o1-preview with a 1.5b model by scaling rl, 2025. Notion Blog.

[29] Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. Realhf: Optimized rlhf training for large language models through parameter reallocation. *arXiv preprint arXiv:2406.14088*, 2024.

[30] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.

[31] NVIDIA. Megatron context parallelism, 2023.

[32] NVIDIA Corporation. Cuda driver api – context management (green contexts). https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__GREEN__CONTEXTS.html, 2025. Accessed: 2025-09.

[33] NVIDIA Corporation. NVIDIA Multi-Process Service (MPS) Documentation, 2025. Accessed: 2025-09.

[34] Open-R1. Codeforces Dataset, 2025.

[35] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym, 2024.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[37] 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?, 2025.

[38] Qwen, :, 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, Tianyi Tang, 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, 2025.

[39] 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*, pages 1–16. IEEE, 2020.

[40] ByteDance Seed, Yufeng Yuan, Yu Yue, Mingxuan Wang, Xiaochen Zuo, Jiaze Chen, Lin Yan, Wenyuan Xu, Chi Zhang, Xin Liu, et al. Seed-thinking-v1. 5: Advancing superb reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.13914*, 2025.

[41] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[42] SGLang Team. Sglang: Fast serving framework for large language models. https://github.com/sgl-project/sglang, 2025. Version 0.4.

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

[44] 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.

[45] 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.

[46] 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.

[47] Guijin Son, Hyunwoo Ko, Hoyoung Lee, Yewon Kim, and Seunghyeok Hong. Llm-as-a-judge and reward model: What they can and cannot do, 2024.

[48] Kimi Team. Kimi k1.5: Scaling reinforcement learning with llms, 2025.

[49] Boxiang Wang, Qifan Xu, Zhengda Bian, and Yang You. Tesseract: Parallelize the tensor parallelism efficiently. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–11, 2022.

[50] 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, 2025.

[51] 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, 2025.

[52] LLM-Core Xiaomi. Mimo: Unlocking the reasoning potential of language model – from pretraining to posttraining, 2025.

[53] Qifan Xu and Yang You. An efficient 2d method for training super-large deep learning models. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 222–232. IEEE, 2023.

[54] Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding. *arXiv preprint arXiv:2503.02951*, 2025.

[55] 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, 2023.

[56] 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 *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, pages 521–538, 2022.

[57] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale, 2025.

[58] 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*, 59(11):56–65, 2016.

[59] Ruiqi Zhang, Daman Arora, Song Mei, and Andrea Zanette. Speed-rl: Faster training of reasoning models via online curriculum learning. *arXiv preprint arXiv:2506.09016*, 2025.

[60] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[61] 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, 2025.

[62] 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, 2025.

[63] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, and Xin Jin. Rlhfuse: Efficient rlhf training for large language models with inter- and intra-stage fusion, 2024.