



USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

RLBoost: Harvesting Preemptible Cloud Resources for Cost-Efficient Reinforcement Learning on LLMs

Yongji Wu, *UC Berkeley*; Xueshen Liu, *University of Michigan*;
Haizhong Zheng, *Carnegie Mellon University*; Juncheng Gu, *Google*;
Beidi Chen, *Carnegie Mellon University*; Z. Morley Mao, *University of Michigan*;
Arvind Krishnamurthy, *Google and University of Washington*; Ion Stoica, *UC Berkeley*

<https://www.usenix.org/conference/nsdi26/presentation/wu-yongji>

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 عبد الله
للعلوم والتقنية
King Abdullah University of
Science and Technology

RLBoost: Harvesting Preemptible Cloud Resources for Cost-Efficient Reinforcement Learning on LLMs

Yongji Wu^{1,*} Xueshen Liu^{2,*;†} Haizhong Zheng⁴ Juncheng Gu³
Beidi Chen⁴ Z. Morley Mao² Arvind Krishnamurthy^{3,5} Ion Stoica¹

¹UC Berkeley ²University of Michigan ³Google ⁴CMU ⁵University of Washington

Abstract

Reinforcement learning (RL) has become essential for unlocking advanced reasoning capabilities in large language models (LLMs). RL workflows involve interleaving rollout and training stages with fundamentally different resource requirements. Rollout typically dominates overall execution time, yet scales efficiently through multiple independent instances. In contrast, training requires tightly-coupled GPUs with full-mesh communication. Existing RL frameworks fall into two categories: co-located and disaggregated architectures. Co-located frameworks fail to address this resource tension by forcing both stages to share the same GPUs. Disaggregated architectures, without modifications of well-established RL algorithms, suffer from resource under-utilization. Meanwhile, preemptible GPU resources, i.e., spot instances on public clouds and spare capacity in production clusters, present significant cost-saving opportunities for accelerating RL workflows, if efficiently harvested for rollout.

In this paper, we present RLBoost, a framework for cost-efficient RL training that harvests preemptible GPU resources. Our key insight is that rollout’s stateless and embarrassingly parallel nature aligns perfectly with preemptible and often fragmented resources. To efficiently utilize these resources despite frequent and unpredictable availability changes, RLBoost adopts a hybrid architecture with three key techniques: (1) adaptive rollout offload to dynamically adjust workloads on the reserved (on-demand) cluster, (2) pull-based weight transfer that quickly provisions newly available instances, and (3) token-level response collection and migration for efficient preemption handling and continuous load balancing. Extensive experiments show RLBoost increases training throughput by 1.51x-1.97x while improving cost efficiency by 28%-49% compared to using only on-demand GPU resources. RLBoost is open-sourced at <https://github.com/TerraFlux/PolyRL>.

*Yongji Wu and Xueshen Liu contributed equally.

†Work was done when Xueshen Liu interned at Google.

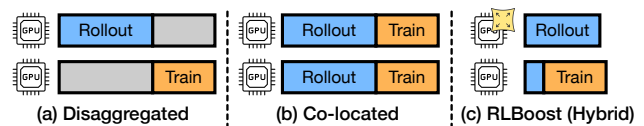


Figure 1: Architectures for on-policy RL frameworks.

1 Introduction

Reinforcement learning (RL) post-training has become the key enabler in unlocking advanced reasoning capabilities for modern large language models (LLMs). RL not only empowers state-of-the-art LLMs like Claude 4 [3] and Grok 4 [4] to achieve leading performance in mathematics, coding, and tool use, but also enables smaller, more efficient models to reach or even surpass the performance of much larger LLMs on specialized tasks [9, 29, 42, 52, 55].

Unlike traditional pre-training, the RL workflow is mainly composed of two interdependent stages: rollout and training. In the rollout stage, input prompts are fed to inference engines, e.g., vLLM [22], to generate a batch of responses. The responses are then used in the training stage to derive reward signals, compute loss, and update model weights. The updated model is subsequently transferred to the inference engines for the rollout stage of the next iteration.

Existing RL frameworks can be divided into two categories: disaggregated and co-located. In disaggregated frameworks, rollout and training stages are assigned to separate sets of GPUs. They either struggle with resource under-utilization (Figure 1(a)) due to bubbles caused by stage dependency [18, 34], or sacrifice model accuracy by using asynchronous (off-policy) algorithms [11, 14, 49, 53] to relax the stage dependency.

To maximize resource utilization under the well-established synchronous (on-policy) RL algorithms, co-located RL frameworks are proposed [24, 35, 54], where training and rollout stages are on the same set of GPUs. The two stages time-share the GPUs, with each GPU alternating between rollout

and training, avoiding idle GPU cycles. However, the two stages exhibit fundamentally different resource requirements. In terms of resource types, the rollout stage partitions available GPUs into multiple independent rollout instances, using tensor parallelism within each instance and requiring no communication across instances. In contrast, the training stage generally employs fully sharded data parallelism (FSDP) [47] and/or 3D parallelism across all available GPUs, involving extensive full-mesh communication between GPUs. In terms of resource quantities, the rollout stage scales efficiently by spawning more independent rollout instances and substantially benefits from allocating more GPUs than the training stage, as generation takes up to 90% of overall RL time under the co-located setting [16].

How can we reconcile this fundamental resource tension under synchronous algorithms without compromising system efficiency or incurring prohibitive monetary costs? Public clouds and private production GPU clusters typically offer their excess capacity in the form of preemptible resources. Spot instances on public clouds provide considerable cost savings (up to 90% cheaper) [38], while production clusters generally have unused GPUs reserved for online workloads [10, 26]. These instances can be preempted at any time by the infrastructure provider. Moreover, these spare GPU resources often suffer from fragmentation at multiple levels, leading to significant communication overhead. At the node level, available GPUs may spread across many nodes, with each node already partially occupied [6, 41]. At the cluster level, available nodes may be topologically scattered across different racks or pods, causing traffic to cross spine and core switches [1, 31]. These preemptible and fragmented resources, while poorly suited for training, align well with the rollout stage's embarrassingly parallel and stateless nature.

Our insight is that through a *hybrid* architecture, we can harvest preemptible resources for high throughput and cost-efficient RL on LLMs. Under the hybrid architecture shown in Figure 1(c), the reserved training cluster performs both training and rollout but opportunistically outsources part of the rollout workload to available preemptible rollout instances.

Still, to efficiently harvest these preemptible resources, there are several key challenges. First, how can we adapt the workloads on the training cluster to dynamic preemptible resource availability? Second, when a new preemptible instance becomes available, how can we quickly provision it with the latest model weights for it to begin rollout, while minimizing progress loss when an instance is preempted? Third, how can we balance the load across rollout instances? The output length of rollout requests in RL exhibits high non-determinism [46], which is further complicated by instance elasticity. Without careful scheduling, tail requests cause severe load concentration on a small subset of instances.

To address these challenges, we propose RLBoost, an RL framework with a hybrid architecture that harvests preemptible resources. To adapt to dynamic resource availability,

RLBoost employs an adaptive rollout offload mechanism. At each step, the training cluster starts from a "seeding" stage, where it is temporarily repurposed for rollout. During this stage, it pre-computes a part of rollout responses that serve as "seeds" for remote rollout instances to continue, before switching to training mode and overlapping with remote rollout through dynamic micro-batch pipelining. RLBoost adaptively tunes the seeding time window based on current workloads and preemptible instance availability.

To quickly provision weights to newly available instances, we decouple weight transfer logic from the training and inference frameworks. We design a pull-based transfer agent that asynchronously transfers model weights, enabling new instances to join and contribute to rollout at any point during a training step.

To minimize preemption overheads and enable fine-grained load balancing, RLBoost collects rollout results at token granularity rather than request level, allowing flexible request migration between rollout instances at any point without progress loss. Building on this token-level stream redirection mechanism, RLBoost incorporates a real-time load balancer that continuously monitors queue depths across rollout instances and redistributes in-flight requests.

We evaluate RLBoost using H100 GPU instances from a public cloud. Extensive evaluations ranging from 8B to 32B models with various spot instance traces show that RLBoost increases overall RL training throughput by 1.51x-1.97x while improving cost efficiency, i.e., the total tokens trained with the same monetary budget, by 28-49%.

In summary, we make the following contributions:

- We identify the fundamental resource tension between the rollout and training stages in RL workflow, and propose a hybrid architecture to harvest preemptible resources for high throughput and cost-efficient RL.
- We design an adaptive rollout offload mechanism to dynamically adapt the training cluster's workloads to real-time resource availability, while adhering to well-established synchronous (on-policy) RL algorithms.
- We develop pull-based weight transfer to quickly provision weights to new instances, complemented by token-level response collection and migration to handle preemptions.
- We conduct extensive experiments to evaluate RLBoost and demonstrate its performance and cost efficiency against state-of-the-art RL frameworks.

2 Background

2.1 Reinforcement Learning for LLMs

Reinforcement Learning (RL) is a technique that predates LLMs but has emerged as the predominant paradigm for fine-tuning LLMs during post-training, aligning them with human

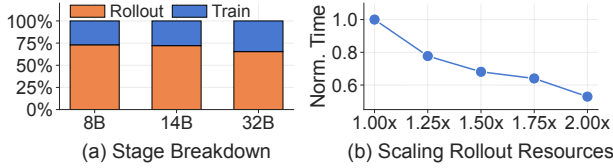


Figure 2: The rollout stage dominates an RL step in the co-located architecture, yet it efficiently scales with more GPU resources as each rollout instance operates independently.

preferences [28] and enhancing their performance on tasks requiring complex reasoning [13, 33]. In a typical RL workflow, the process begins with a pre-trained base model that serves as the agent model to be optimized. The agent learns to take a set of actions given an input and receives different rewards based on the actions it takes. In the context of LLMs, the inputs are initial prompts that present tasks for the model to solve. The model takes actions by generating tokens autoregressively, where each generated token constitutes an individual action. The agent LLM’s behavior is optimized by training it to learn action sequences that maximize the expected reward.

Although there are a variety of RL algorithms, e.g., PPO [32] and GRPO [13], they all revolve around two main stages within each RL step: rollout and training. Each step begins with the rollout stage, where the LLM agent takes actions by processing a batch of prompts and generating a single or a group of responses per prompt, similar to traditional LLM inference. Upon completion, each generated response forms a training sample.

In the training stage, a reward [13] is computed for each response (i.e., training sample) to derive the loss function for model updates. The reward typically comes from a rule-based verifiable function, such as a binary signal indicating whether the response successfully completes a coding task or correctly answers a mathematical question. Alternatively, the reward can be derived from another LLM, referred to as a reward model. RL algorithms may also employ critic models or reference models to provide additional loss signals. However, these auxiliary models remain frozen during RL training and generally introduce insignificant computational overhead. Mainstream LLMs are predominantly trained with synchronous (on-policy) RL algorithms [13, 33, 44, 50]. After the agent LLM is updated, the new model weights are immediately used in the next rollout stage, ensuring that responses are always generated using the latest version of the model.

2.2 RL Frameworks

Existing RL frameworks can be categorized into two architectures: co-located and disaggregated. Early RL frameworks adopted the disaggregated architecture [18, 34] to effectively reuse existing system infrastructures. The training stage is deployed on one set of GPUs using frameworks such as

Table 1: Overview of existing RL frameworks for LLMs.

Systems	On-policy Optimized	Resource Decoupling	Preemptible Resources
veRL [35]	✓	✗	✗
StreamRL [53]	✓	✓	✗
AReaL [11]	✗	✓	✗
AsyncFlow [14]	✗	✓	✗
RhymeRL [16]	✗	✓	✗
RLBoost	✓	✓	✓

Megatron-LM [36], while the rollout stage is deployed with another set of GPUs using frameworks like vLLM [22]. At each step, weights are first transmitted from the training workers to the rollout workers, after which the rollout and training stages execute sequentially. At any given time, one set of GPUs remains idle while waiting for the other set to complete its stage. There are some recent disaggregated frameworks that improve system efficiency by optimizing for asynchronous (off-policy) RL algorithms [11, 14, 16, 53].

To address the resource utilization issue under the widely-adopted synchronous (on-policy) RL algorithms, the co-located architecture is developed [24, 35], which switches between rollout and training on the same set of GPUs. However, there is a fundamental mismatch in resource requirements between the two stages as described in §1. Figure 2(a) presents the step time breakdown for training Qwen3 [37] models using the co-located veRL framework, with experimental details in §6.2. Rollout accounts for up to 73% of the overall time, yet it can be easily accelerated with more GPU resources, as shown in Figure 2(b).

We compare existing RL frameworks in Table 1. None can leverage preemptible resources and adapt to dynamic resource availability, whether they are disaggregated or co-located.

3 Overview

RLBoost is a hybrid RL framework that leverages preemptible instances for high-throughput and cost-efficient RL training. We present the major components of RLBoost in Figure 3. RLBoost employs a fixed (reserved) training cluster, as the training stage requires tightly coupled GPUs, and frequent preemptions would incur significant checkpoint-restart overhead. We note that existing fault-tolerant training techniques [17, 39, 40] are orthogonal to our work, and RLBoost can directly benefit from them. RLBoost also utilizes an elastic pool of preemptible GPU instances to offload rollout workloads from the training cluster, where instances can be dynamically allocated or preempted at any time. We refer to these preemptible instances dedicated to rollout offload as (*remote*) *rollout instances*. These instances can be located either in the same datacenter or cloud region as the training cluster, or distributed across different datacenters or cloud regions. As shown in

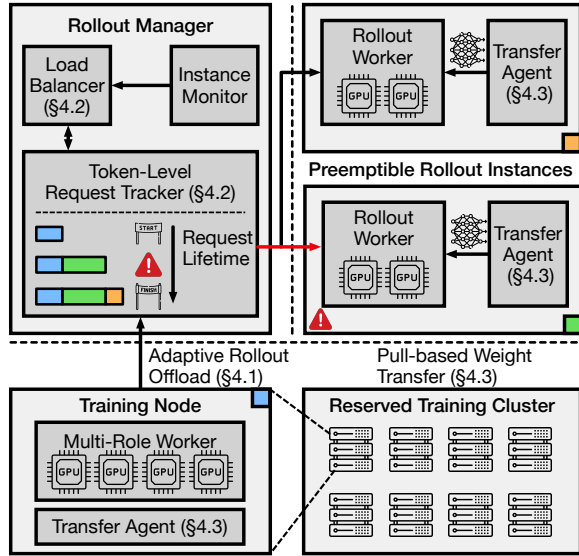


Figure 3: System overview of RLBoost.

Figure 2, rollout typically consumes the majority of step time; therefore, offloading it to more affordable preemptible resources can significantly increase throughput while reducing monetary costs.

The core component connecting the training cluster and rollout instances is the rollout manager. It monitors the health of each rollout instance, handles preemptions, and launches rollout workers when new instances become available.

In every step, the rollout manager sends a part of the rollout requests to rollout instances on behalf of the training cluster. It continuously tracks the status of each request and collects the responses in token granularity. A load balancer distributes requests across rollout instances and redirects in-flight requests upon load variations or instance preemptions.

To adapt the amount of workload offloaded from the training cluster to dynamic rollout instance availability and balance remote and local execution, RLBoost employs multi-role workers on the training nodes, which can be temporarily repurposed for rollout at the beginning of each RL step. While the rollout instances are receiving weights and generating the first stream of responses, the training cluster would handle rollout requests within a specific time window. Such mechanism enables the training cluster to "seed" a part of the responses for remote rollout instances to continue the work. To enable a new rollout instance to join and participate in the rollout at any time during a step, RLBoost decouples the weight transfer logic from the training and rollout workers into dedicated transfer agents. The agents send and receive weights asynchronously while the training node is occupied with either seeding rollout or training tasks.

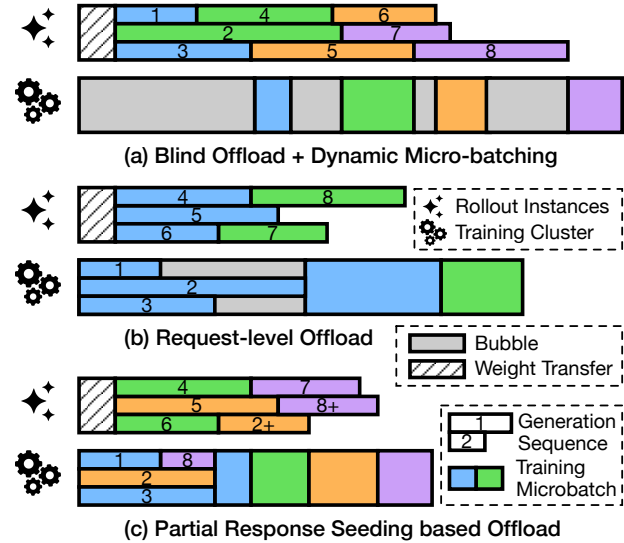


Figure 4: RLBoost minimizes training cluster idling with an adaptive partial response seeding mechanism.

4 Design

4.1 Adaptive Rollout Offload

Because RLBoost offloads rollout from the training cluster to a separate pool of preemptible instances, it faces the same resource idleness issue as shown in Figure 1(a), attributed to the dependency between rollout and training. To overlap the execution of the training cluster and remote rollout instances, we can employ dynamic microbatch pipelining, similar to [53]. The training cluster collects responses from the rollout manager as soon as they are generated, until a minimum microbatch size of m_b is reached, then immediately begins training of the microbatch, as illustrated in Figure 4(a). If more than m_b responses arrive at once, they are gathered in a single microbatch. Since gradients are accumulated across all responses, they can be collected and batched without preserving the original order in which prompts are issued to rollout instances. Notably, even in the co-located architecture, training is already executed in a series of microbatches, because all generated responses cannot fit into a single training batch constrained by GPU memory. Hence, dynamic micro-batching does not hurt compute efficiency.

If we blindly offload all rollout computation, even with dynamic micro-batching, the training cluster still suffers from significant bubbles, especially when insufficient remote rollout instances are available. Specifically, the training cluster must wait for rollout instances to receive model weights at the beginning of each step, and wait between microbatches for responses to be generated.

To balance the execution between the training cluster and remote rollout instances, RLBoost must dynamically adjust the offloaded rollout workload to adapt to preemptible re-

Algorithm 1: Adaptive partial response seeding.

Input : N_{resv} : Number of local rollout engines in the reserved training cluster; η : Adaptation rate; T_{init} : Initial seeding window, S : Total number of training steps.

- 1 $\mathcal{M} \leftarrow \emptyset$ // scheduler memory
- 2 $T_{\text{seed}} \leftarrow T_{\text{init}}$ // rollout time window on the training cluster
- 3 $N_{\text{prem}} \leftarrow N_{\text{resv}}$ // initialize max amount of preemptible instances the same as reserved rollout engines
- 4 **for** $s \leftarrow 1$ **to** S **do**
- 5 EXECUTESTEP($T_{\text{seed}}, N_{\text{prem}}$)
- 6 $\bar{n}_{\text{prem}}, \hat{n}_{\text{prem}} \leftarrow \text{MONITORINSTAVAIL}()$
- 7 $t_{\text{train}}^{\text{wait}}, T_{\text{remote}}^{\text{wait}} \leftarrow \text{GETIDLETIME}()$
- 8 $t_{\text{train}}, t_{\text{remote}} \leftarrow \text{GETCOMPUTETIME}()$
 // update schedule
- 9 $T_{\text{seed}} \leftarrow T_{\text{seed}} + \frac{t_{\text{train}}^{\text{wait}} - t_{\text{remote}}^{\text{wait}}}{\eta}$
- 10 $N_{\text{prem}} \leftarrow \frac{t_{\text{remote}} \bar{n}_{\text{prem}} + T_{\text{seed}} N_{\text{resv}}}{t_{\text{train}}}$
- 11 **if** $\bar{n}_{\text{prem}} = \hat{n}_{\text{prem}}$ **then**
 // memorize schedule optimized under \hat{n}_{prem}
- 12 $\mathcal{M}[\hat{n}_{\text{prem}}] \leftarrow T_{\text{seed}}$
- 13 **if** $\hat{n}_{\text{prem}} \in \mathcal{M}$ **then**
 // retrieve latest schedule optimized for \hat{n}_{prem}
- 14 $T_{\text{seed}} \leftarrow \mathcal{M}[\hat{n}_{\text{prem}}]$

source availability. A straightforward approach is to assign a specified number of rollout requests for the training cluster to generate locally while offloading the rest to remote instances, as is shown in Figure 4(b). However, this offloading strategy is too coarse-grained. Since response lengths are highly unpredictable, the training cluster may be stuck generating long-tail responses even after receiving sufficient responses for training.

Our insight is that instead of controlling the number of rollout samples to offload, we can bound the trainer’s rollout work in *time* to make progress predictable. To this end, we design a partial response seeding mechanism: RLBoost allows training cluster to rollout only within a fixed time window at the beginning of each step, and then it transitions to training. For long-tail responses, the training cluster “seeds” a part of the response for rollout instances to continue from, as illustrated in Figure 4(c) for response 2. Since rollout instances only need to compute a single prefill over the already generated tokens, migrating partially generated responses introduces minimal overhead (see §4.2).

However, determining the optimal seeding duration remains non-trivial. If set too long, training is unnecessarily delayed; if too short, training cluster still experience bubbles waiting for responses. Moreover, the optimal setting is dynamic due to two key factors. In addition to the fluctuating number of available preemptible instances for rollout, the average response length tends to grow as RL training pro-

gresses [46]. These factors cause unpredictable changes in rollout and training times throughout the RL training process.

Beyond the challenge of identifying the optimal seeding window, another question is how many preemptible instances should we actually use, even when availability is unlimited. Given their cost advantages, we can follow the established practices in [10, 25, 38] and use as many instances as available to maximize the generation speedup. However, the training stage still imposes a lower bound on step time. Hence, we must avoid over-provisioning remote rollout instances.

We present an adaptive scheduling algorithm in Algorithm 1 that addresses both initial idling on the training cluster and resource waste of preemptible instances. Each remote rollout instance uses the same number of GPUs as one local rollout engine’s tensor parallel size. The algorithm dynamically adjusts the seeding window T_{seed} and enforces a maximum number of allowed remote rollout instances N_{prem} by monitoring step time statistics. In each step, RLBoost tracks idle time on both the training cluster $t_{\text{train}}^{\text{wait}}$ and remote rollout instances $t_{\text{remote}}^{\text{wait}}$. $t_{\text{remote}}^{\text{wait}}$ represents the idle time on the training cluster, waiting for sufficient responses to fill a microbatch. $t_{\text{remote}}^{\text{wait}}$ measures how long remote instances wait for the training cluster to complete the current step, after they generate the last response. Ideally, to minimize the total step time, we should minimize $T_{\text{seed}} + t_{\text{train}}^{\text{wait}}$, as the training cluster’s completion marks the step completion. However, due to the unpredictable nature of responses arrivals, generation lengths, and instance availability, $t_{\text{train}}^{\text{wait}}$ and $t_{\text{remote}}^{\text{wait}}$ are highly indeterministic. They are also intertwined and are both correlated with T_{seed} . Hence, RLBoost employs a feedback-driven mechanism to incrementally tune T_{seed} , maintaining stability across steps under fluctuations while adapting to evolving workload patterns. RLBoost should increase T_{seed} when observing a significant $t_{\text{train}}^{\text{wait}}$. Yet, T_{seed} cannot grow indefinitely as it would delay the overall step time, which will be reflected in a longer $t_{\text{remote}}^{\text{wait}}$. As shown in line 9 of Algorithm 1, RLBoost adjusts T_{seed} by balancing between the two objectives, with a scale factor η applied to the adjustment delta.

The tuning in line 9 needs gradual progression to converge after the number of remote instances changes. To mitigate the re-tuning overhead when many instances join or are preempted during a step, RLBoost employs a memorization mechanism in line 14 to directly start from the latest T_{seed} optimized under \hat{n}_{prem} instances, where \hat{n}_{prem} is the number of active rollout instances available before the start of the subsequent step. The scheduler memory \mathcal{M} is continuously updated after each step in line 12, provided no instance changes occurred during the step, i.e., only when $\hat{n}_{\text{prem}} = \bar{n}_{\text{prem}}$. \bar{n}_{prem} is the number of instances averaged over the duration of a step.

To prevent RLBoost from over-allocating remote rollout instances that would yield no further performance improvement, in line 10, RLBoost sets the upper bound N_{prem} by computing how many instances are required for the rollout stage to take less time than t_{train} . t_{train} is the effective time the training

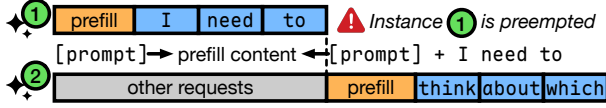


Figure 5: RLBoost collects responses at token granularity and migrates requests upon instance preemption, incurring only the cost of an additional prefill.

cluster spent on training in a step, i.e., excluding idle periods. To preclude the impacts of T_{seed} , we assume rollout is solely processed by remote instances when computing N_{pre} , where $t_{\text{remote}}\bar{n}_{\text{pre}} + T_{\text{seed}}n_{\text{resv}}$ is the total rollout workload. n_{resv} is the number of rollout engines (instances) the training cluster is divided into during rollout seeding. The rollout manager in RLBoost keeps tracks of instance availability and allocates new instances upon availability. If there are already N_{pre} remote instances, RLBoost will not allocate a new instance even if more are available.

With adaptive rollout offload minimizing training cluster idle time, RLBoost falls back to colocated rollout when spot capacity is unavailable or frequently preempted, without stalling progress. Next, we explore how RLBoost enables no-waste preemption handling and continuous load balancing with token-level response collection.

4.2 Live Request Tracking and Migration

Since a rollout instance can be preempted at any instant, requests routed to it may not complete generation when preemption occurs. To preserve the correctness of the RL workflow, we cannot drop any of preempted requests. However, simply retrying the request on another rollout instance from the original prompt would result in significant progress loss, particularly when most tokens of a sample has been generated. To minimize lost progress and redundant computation upon a preemption, our insight is to collect the response at token granularity. For each request, the rollout manager spawns an asynchronous task to track and receive the response tokens in a streaming manner. When an instance is preempted, RLBoost still preserves partially generated responses for requests routed to the instance. For each partially generated sample, RLBoost migrates the request to one of the healthy instances to continue generation (Figure 5). Similar to §4.1, the redirected instance only performs a prefill operation on the concatenated prompt and previously generated tokens, incurring negligible overhead compared to generating from the beginning (original prompt).

4.2.1 Continuous Load Balancing

Such token-level response collection not only reduces the costs of a preemption, it also empowers RLBoost with the ability to flexibly migrate and redistribute samples across instances, allowing continuous load monitoring and balancing.

Algorithm 2: RLBoost’s load balancer.

Input : I : Set of rollout instances; \mathcal{P} : Inference batching profile table; Θ : Maximum pending requests threshold.

```

1 function SELECTINSTANCE( $I$ )
2   while true do
3      $C \leftarrow \emptyset$ 
4     foreach  $i \in I$  do
5        $m_i^{\text{pending}} \leftarrow \text{QUERY PENDING}(i)$ 
6       if  $m_i^{\text{pending}} < \Theta$  then
7          $C \leftarrow C \cup \{i\}$ 
8     if  $C \neq \emptyset$  then
9        $i \leftarrow \arg \min_{i \in I} m_i^{\text{pending}}$ 
10      return  $i$ 
11    else
12       $\text{WAIT ANY COMPLETION}()$ 
13 procedure CONTINUOUSLB( $I, \mathcal{P}$ )
14   while true do
15     foreach  $i \in I$  do
16        $m_i^{\text{pending}} \leftarrow \text{QUERY PENDING}(i)$ 
17        $m_i^{\text{exec}} \leftarrow \text{QUERY EXECUTING}(i)$ 
18     if  $\exists i, m_i^{\text{pending}} = 0$  and  $\exists k, m_k^{\text{pending}} > 0$  then
19        $j \leftarrow \arg \max_{k \in I} m_k^{\text{pending}}$ 
20       // migrate a single request
21        $\text{MIGRATE REQS}(j \rightarrow i, 1)$ 
22     else if  $\exists i, m_i^{\text{exec}} = 0$  then
23        $j \leftarrow \arg \max_{k \in I} m_k^{\text{exec}}$ 
24        $B \leftarrow \text{GET BATCHING PLATEAU}(\mathcal{P})$ 
25        $r \leftarrow \max(m_j^{\text{exec}} - B, 0)$ 
26       // migrate  $r$  requests
27        $\text{MIGRATE REQS}(j \rightarrow i, r)$ 

```

We present the load balancer logic for RLBoost in [Algorithm 2](#). It is composed of two main components: SELECTINSTANCE is used for initial candidate instance selection when a generation request is first scheduled, and re-routing when the previously selected instance is preempted. CONTINUOUSLB is a background monitor task to continuously migrate requests from overloaded instances to underloaded ones as needed.

SELECTINSTANCE endorses the classical join the shortest queue (JSQ) scheduling policy widely used in web servers. It routes the generation request to the instance with the minimum number of pending requests (line 9), i.e., requests that are already sent to the instance but have not been scheduled to execute yet. In the traditional JSQ policy, a request is immediately dispatched to an instance upon receiving it. Such a strategy works well in typical web servers of CPU-based processing, where requests are mostly homogeneous in the way that they take roughly the same amount of time to process. However, in LLM generation, instances with the most pending requests could complete the earliest due to variance in

generation lengths. If all requests are immediately dispatched, we may need to frequently migrate requests to balance the load, causing unnecessary overhead. Instead, RLBoost adopts a *delayed dispatch* approach, where we limit the number of outstanding pending requests to Θ for each instance. If all instances are already occupied with more than Θ pending requests, RLBoost waits for any of the in-flight request to finish (line 12) and rechecks the pending status (line 2), holding the request until one of the instances becomes available.

Once all requests are dispatched, RLBoost monitors and dynamically rebalances load with CONTINUOUSLB. In lines 16–17, RLBoost tracks both the number of pending requests m_i^{pending} and the number of currently executing requests m_i^{exec} for each instance. RLBoost first checks if any instance i has no pending requests while other instances have (line 18). RLBoost migrates pending requests from the most overloaded instance j to i , one request at a time (line 20). If instance i has enough capacity, the migrated request will be immediately scheduled. In this case, RLBoost keeps migrating more requests to instance i until it is saturated, i.e., subsequent requests to i will queue up.

If there are no pending requests on all instances, RLBoost then checks if any instance i is completely idle (line 21), i.e., is not executing any request. In this scenario, RLBoost finds the most loaded instance j with the largest m_j^{exec} (line 22). Different from the scenario with pending requests, migrating executing requests may not lead to earlier completion due to the batching effects of LLMs. If m_j^{exec} is small enough, the generation is completely memory-bound, removing requests from j leads to no improvement in inter-token latency (ITL), but instead a linear decrease in generation throughput. However, if m_j^{exec} is beyond the point where further increases in batch size yield only marginal throughput gains, migrating a part of the requests out of j helps speed-up the overall generation. In line 24, RLBoost determines the number of requests r to migrate from j to i by clamping m_j^{exec} to the batch size B where the generation throughput plateaus, where B is computed from a profile table \mathcal{P} of throughput under different batch sizes (line 23). Instead of offline profiling, \mathcal{P} is *online captured* by RLBoost during the previous step’s rollout, and is *continuously calibrated* to account for the current average context length. We also tried directly incorporating both batch size and the context length into \mathcal{P} , but found it difficult to fit the performance model across two dimensions, resulting in worse estimates. We note that since \mathcal{P} is only established after the first step, CONTINUOUSLB begins to migrate executing requests from the second step onward.

At this point, through adaptive rollout offload and migration-based load balancing, RLBoost can maximize effective compute on the training cluster and remote rollout instances, while efficiently handling preemptions. Next, we discuss how RLBoost decouples the weight transfer logic from the training and generation workers.

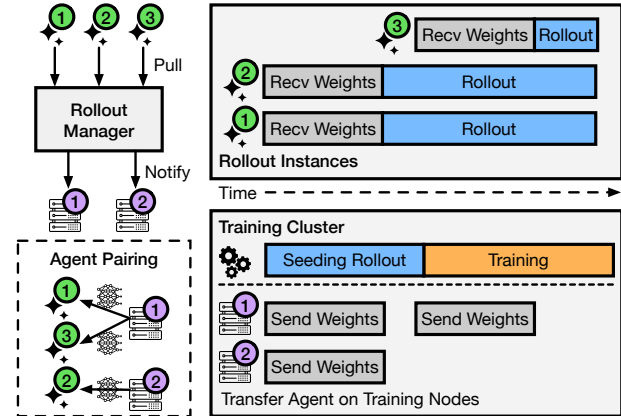


Figure 6: Pull-based weight transfer enables newly allocated rollout instances to be quickly provisioned with the latest model weights without blocking existing workers.

4.3 Pull-based Weight Transfer

After the training stage and the model is updated, RLBoost will reshard model weights for seeding rollout on training cluster, in the same way as co-located RL frameworks. The resharding within the training cluster is carried out over fast interconnects like NVLink and RDMA, which can be significantly faster than the bandwidth between training cluster and rollout instances. In modern GPU clusters, the frontend and backend networks are typically separated, with the high-capacity backend network dedicated for GPU data traffic within the cluster [8, 12]. On public clouds, even if training cluster and rollout instances are located in the same datacenter, they can be limited by the slower frontend network [12].

Besides the asymmetric network bandwidth problem, if we use the synchronized weight update approach in co-located frameworks that transfers weights only after each step, an instance joined midway through a step cannot process requests until the next step. Also, the completion of weight update can be blocked by rollout instances with poor network bandwidth.

To unblock the training cluster for rollout seeding and to immediately transfer the latest weights to a rollout instance once they are allocated, RLBoost employs a pull-based transfer agent to asynchronously transfer weights, as shown in Figure 6. The transfer agent is a separate process residing on each training node and rollout instance. During the intra-cluster all-gather, each training node copies the full model weights from GPU to a pre-allocated CPU buffer managed by the transfer agent. After that, the training cluster immediately starts seeding rollout, instead of waiting for the weight delivery to all rollout instances. Each rollout instance is paired with a weight transfer agent in a round-robin way and establishes a peer-to-peer connection. On initial registration or model update, a rollout instance will independently pull the latest weight and start generation once the transfer finishes, without affecting other rollout instances and training cluster.

5 Implementation

We implement RLBoost based on a derived version of veRL [35] in 2.7K lines of Python and 1.7K lines of Rust. RLBoost supports PyTorch FSDP [47] and Megatron [36] for training and uses SGLang [51] for rollout.

Rollout manager. We implement the rollout manager as a RESTful API web service using Rust’s asynchronous framework with Tokio [7] and Axum [2]. The manager monitors instance availability and allocates new rollout instances when permitted, ensuring the total count does not exceed the upper bound N_{prem} . It keeps track of idle waiting time and effective compute time reported by the rollout instances and the training cluster, which are used to compute T_{seed} and configure the training cluster for the next step. The rollout manager also periodically probes each rollout instance’s m_i^{pending} and m_i^{exec} for load balancing. For each rollout request, an asynchronous task is launched to track the request’s entire lifetime, collecting response tokens as they are generated. If the routed instance is preempted, the tracking task would detect a connection-closed error, it then immediately requests another healthy instance from the load balancer and migrates the request.

When a new rollout instance is allocated, it registers with the manager, which then assigns it to a weight sender agent, as described in §4.2. The manager maintains the weight version of each instance and notifies the paired sender agent to transfer weights if an instance is not up-to-date. The manager only routes requests to instances that have loaded the latest weights.

Trainer workers. Using components from veRL [35], we implement the trainer worker to support dynamic micro-batching as discussed in §4.1, with an asynchronous task running on the CPU collects complete responses from the rollout manager and packs them into micro-batches.

Transfer agents. We assume weight transfer is conducted over the frontend network (§4.3) and thus does not contend with training or inference traffic on the backend network. Since RDMA networks between rollout instances and the training cluster may not be available, particularly in public cloud environments, we implement a TCP-based weight transfer engine in our prototype. To fully utilize the bandwidth of all available frontend NICs, RLBoost uses multiple I/O threads with each handling a different weight shard, while each sender agent transfers weights to multiple rollout instances simultaneously. The design of RLBoost’ pull-based weight transfer is agnostic to the underlying transport. When RDMA interconnects are available, we can easily integrate RDMA optimized transport engines (e.g., Mooncake [30] and NIXL [5]) into RLBoost.

6 Evaluation

In this section, we evaluate RLBoost against co-located and disaggregated RL frameworks with models from 8B to 32B,

Table 2: Specification of public cloud instances.

	On-Demand Training	Spot Rollout
vCPUs	208	52
GPU	8xH100	2xH100
Network	(4+1)x200 Gbps	50 Gbps
Cost	\$83.79/h	\$5.32/h

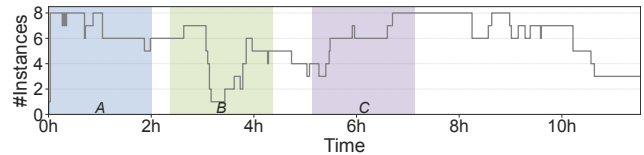


Figure 7: The complete 12-hours availability trace for 2xH100 instances and the three 2-hours segments (A, B, C) extracted.

comparing both performance and cost efficiency. We then breakdown the benefits of different components of RLBoost.

6.1 Setups

Hardware settings. We evaluate RLBoost using H100 GPU instances on a public cloud. For the training cluster, we target on-demand (reserved) instances each fully equipped with 8 H100 GPUs. For preemptible rollout instances, we target spot instances each equipped with 2 H100 GPUs, since fragmented instances generally have better availability [10]. We list the detail specifications of both instance types in Table 2. Notably, the 8xH100 instances feature 4 backend NICs with an RDMA featured networking stack, whereas the 2xH100 instances are limited to a single frontend vNIC. Consequently, 8xH100 must rely on its single 200 Gbps frontend NIC to communicate with 2xH100. All GPUs in a single instance are fully connected with 900 GB/s NVLink. For better representativeness, in Table 2, we calculate the average cost of on-demand and spot instances with the same GPUs across different cloud providers in different regions. These per-instance costs are used to compute cost efficiency of compared systems.

Workloads. We utilize models from the popular Qwen 3 family [37] as base models, spanning from 8B to 32B. We provide their configurations in Appendix B. We employ the synchronized (on-policy) GRPO algorithm, the current mainstream RL algorithm for LLMs. We note that many newly proposed algorithms, e.g., DAPO [46] and GMPO [48] are derived from GRPO and share similar workload patterns.

We use a math dataset OpenR1-Math [27] to train the models, which has a maximum response length of 14K tokens. Following the practice in [35, 53], we use a global batch size of 128 prompts, with a GRPO group size of 8. The models are trained with FSDP [47]. To demonstrate RLBoost across different cluster scales, we use a single on-demand 8xH100 instance for training 8B and 14B models, while for 32B models

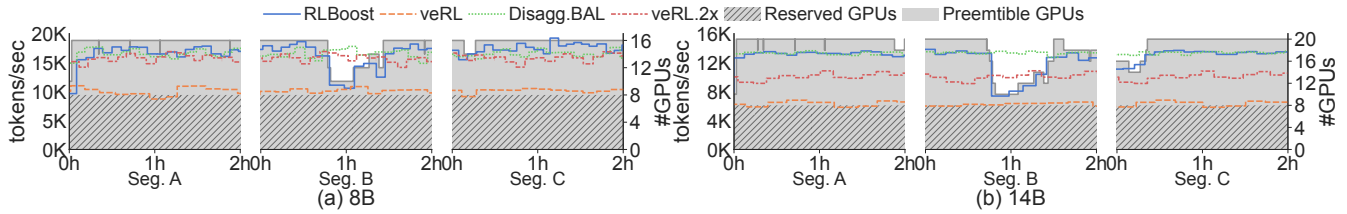


Figure 8: **[Overall evaluation]:** Throughput over each trace segment for Qwen-8B and Qwen-14B. The number of reserved GPUs and the number of preemptible GPUs allocated and used by RLBoost is also shown. veRL, veRL.2x and Disagg.BAL only use reserved GPUs, with veRL.2x and Disagg.BAL use more reserved GPUs than RLBoost.

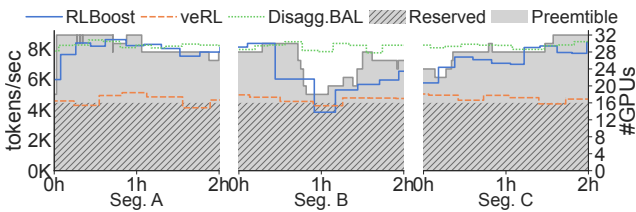


Figure 9: **[Overall evaluation]:** Throughput over each trace segment for Qwen-32B.

we use two $8 \times H100$ instances.

Traces. For evaluation reproducibility, we follow [10, 19, 25] to take real spot instances trace from [38] and replay them on on-demand instances. Following the practice in [10, 25], we extract three representative 2-hours segments in as the preemption traces for the $2 \times H100$ instances, and describe their characteristics in Appendix C. We note that all $8 \times H100$ instances are reserved and will not be preempted.

Metrics. Following [11, 35], we report the system performance in terms of effective training throughput. The throughput of a step is measured as the total number of tokens generated and trained in the step, divided by the time of the step.

6.2 Overall Evaluation

We compare the end-to-end performance of RLBoost with the following systems:

- **veRL** [35]: A state-of-the-art RL system under the co-located architecture. It features an optimized execution engine to efficiently manage and execute RL workflows for LLMs. We run veRL on the training cluster in each setup, i.e., a single $8 \times H100$ instance for Qwen3-8B and Qwen3-14B, and two $8 \times H100$ for Qwen3-32B.
- **veRL.2x**: To evaluate how the cost efficiency varies for the co-located architecture when scaling up, we also run veRL with 2x more hardware resources, e.g., two $8 \times H100$ instances for 8B and 14B models. veRL.2x is not evaluated on Qwen3-32B, as we do not have additional reserved $8 \times H100$ instances.

- **Disagg.BAL (Balanced)**: StreamRL [53] is a state-of-the-art disaggregated framework that targets asynchronous RL. However, it also includes several optimizations that can be applied to the on-policy setting. Since it is not open sourced, we implement a disaggregated framework using techniques from [53]. In particular, it features a resource optimizer that determines the number of GPUs allocated to each stage that balances the workloads and process rollout results in microbatches to reduce bubbles. We use it to calculate the optimal number of reserved $2 \times H100$ instances for rollout, given the number of GPUs used in the training cluster.

Note that none of these systems can take advantage of preemptible instances. We present the training throughput and RLBoost’s GPU usage over the duration of each segment in Figure 8 and 9. We see that RLBoost’s throughput fluctuates as $2 \times H100$ instance availability changes, while throughput of other compared methods remain relatively stable as they only use reserved GPUs. We observe that in segment A there are many tiny spikes in preemptible GPU usage, which are also shown in Figure 7. These spikes occur at the timestamps where a running $2 \times H100$ instance is preempted, but a new one can be immediately allocated (also observed in [23, 38]). RLBoost shows negligible throughput drops in these cases, demonstrating that RLBoost can effectively handle request failures and quickly set up new instances as they are allocated.

We note that the preemption patterns are different across 8B to 32B models. Limited by N_{prem} , RLBoost does not allocate all available instances, hence a preempted instance may not be in use. The throughput is computed at the end of each RL step. For Qwen3-32B, each step takes significantly longer time than 8B and 14B, therefore the throughput changes are not immediately reflected in the curves. Nevertheless, the performance boost RLBoost brings with preemptible GPU resources tightly matches resource availability.

We show the average throughput over each segment and training costs in Figure 10. Compared to veRL that only uses the reserved training cluster, RLBoost significantly increases throughput. Across three segments, RLBoost outperforms veRL by 1.66x, 1.97x and 1.51x for 8B, 14B and 32B models, respectively, in terms of average throughput. RLBoost even achieves up to 24% higher throughput than veRL.2x, which

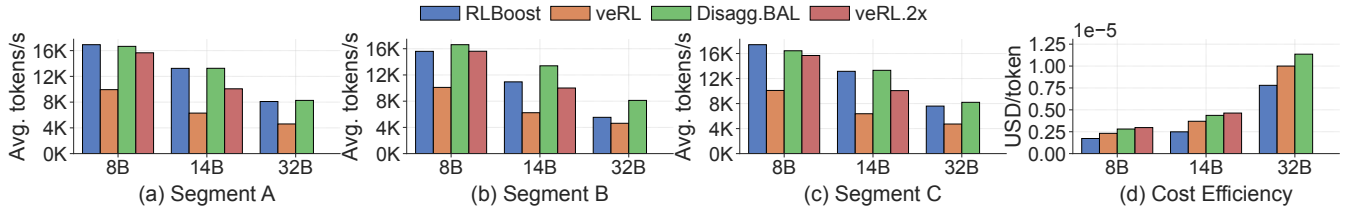


Figure 10: **[Overall evaluation]**: Average throughput and cost efficiency across all three trace segments.

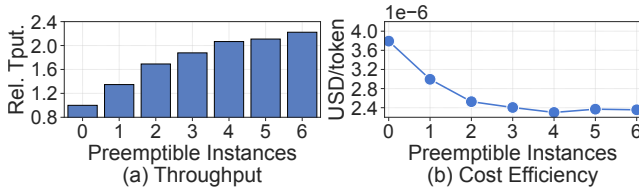


Figure 11: **[Cost efficiency]**: Throughput and cost efficiency of RLBoost on Qwen-14B with a static number of preemptible rollout instances. 0 refers to only use the training cluster.

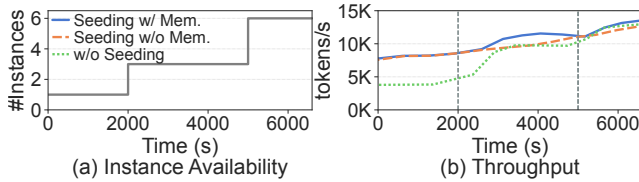


Figure 12: **[Ablation study]**: The impacts of adaptive rollout offload with partial response seeding on Qwen-3-14B.

uses two $8 \times H100$, since the FSDP training in veRL.2x spans across two nodes and suffer additional overheads.

The throughput of RLBoost matches Disagg.BAL when there are sufficient preemptible $2 \times H100$ available. In this case, RLBoost offloads virtually all rollout computation to $2 \times H100$ instances, and thus exhibits comparable performance to Disagg.BAL. For instance, RLBoost’s average throughput is just 2% higher than Disagg.BAL over segment A. However, Disagg.BAL completely disaggregates rollout from the training cluster, and thus is unable to dynamically adapt the training cluster’s workload in response to preemptible instance availability. It also lacks efficient request migration and pull-based weight transfer mechanisms, preventing it from handling dynamic instance changes.

Across all segments, RLBoost improves the training cost efficiency by 34%, 49% and 28% for 8B, 14B and 32B models, compared to veRL. In segment A where the overall instance availability is high, RLBoost’s achieves higher cost savings compared to veRL, improving cost efficiency by 36%, 53% and 45% across 8B-32B models. The cost efficiency in segment B is relatively poor due to frequent preemptions and low availability, where the improvement of RLBoost over veRL drops to 37% even for 14B.

Since Disagg.BAL can only use reserved instances, it suf-

fers from poor cost efficiency. Across all three segments, its per-token training cost is 62%, 75% and 45% higher than RLBoost, for 8B, 14B and 32B models.

6.3 Analysis of Cost Efficiency

In Figure 10(d), we show the cost efficiency of RLBoost under fluctuating instance availability. We now break down how the throughput and cost efficiency of RLBoost change under a steady setting with different numbers of preemptible rollout instances. We show the results in Figure 11.

The throughput is relative to the case with no preemptible instances for rollout, where RLBoost falls back to the same workflow as veRL, in which rollout is fully executed on the training cluster. Both the throughput and cost efficiency improves until they reach the saturation point where rollout is already accelerated enough to match the training speed of the training cluster. Even with only one instance, the throughput increases by 37% and the per-token training cost reduces by 22%. With 6 instances, throughput further increases by 64% compared to a single instance, while the cost further reduces by 21%. We note that the trend in cost efficiency depends on the relative resource (GPU) ratio between the training cluster and the rollout instance pool, rather than the absolute number of rollout instances. RLBoost efficiently scales to more rollout instances as the training cluster scales. Besides, our evaluation across different maximum response lengths in Appendix E shows that RLBoost can utilize more and more preemptible instances as length grows to improve throughput and cost efficiency.

6.4 Ablation Study

6.4.1 Impacts of Adaptive Rollout Offload

We break down how RLBoost adapts the workloads on the training cluster to changes in preemptible instance availability. As described in §4.1, RLBoost adaptively offloads rollout to remote instances using a partial response seeding mechanism. In Figure 12, we illustrate how RLBoost tunes T_{seed} with Algorithm 1 when instance availability changes. We construct a scenario where 5 out of 6 $2 \times H100$ rollout instances are initially preempted, with substitute instances gradually becoming available after a period of time. Note that the initial preemptions are not displayed in Figure 12(a).

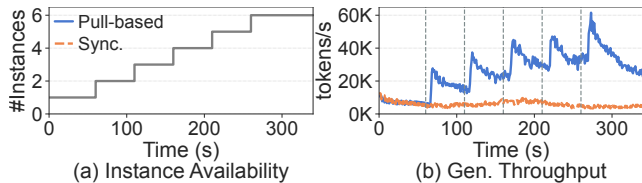


Figure 13: **[Ablation study]**: Comparing pull-based and synchronized weight transfer as new instances are allocated within a step. We use Qwen3-14B.

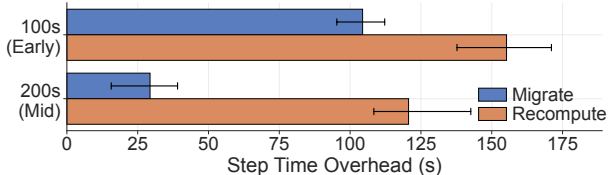


Figure 14: **[Ablation study]**: Comparing different strategies for handling request failures upon preemptions on Qwen-14B. Error bars represent 95% percentile intervals.

We compare three solutions: response seeding using the complete [Algorithm 1](#) (full solution), a variant without scheduler memory, and a variant that disables response seeding. We observe that without seeding, RLBoost has to blindly offload all rollout requests to remote instances. Consequently, the training throughput is significantly lower during the initial stage, when only a single instance remains after preemptions. However, with more instances added, remote rollout is fast enough that T_{seed} becomes negligible. Hence, *w/o seeding* matches the performance of the full solution after all 6 instances become available. Over the duration of [Figure 12](#), *w/o seeding* decreases the average throughput by 19% compared with the full solution. Comparing seeding with and without memory, we find that the scheduler memory reduces the convergence time of T_{seed} when new instances are allocated, resulting in a further 6% average throughput increase.

6.4.2 Impacts of Weight Transfer Paradigm

The pull-based weight transfer agents in RLBoost decouple the transfer logic from training and rollout workers, allowing newly allocated preemptible instances to be quickly provisioned with the latest weights and participate in the current step’s rollout. We construct a scenario where new instances are progressively allocated within a step. This represents the case that can be observed in [Figure 7](#): after simultaneous preemptions of many instances, new instances become available within a short period of time.

We illustrate the effects in [Figure 13](#). Since we focus on the performance within a step, we cannot report effective training throughput, as it is computed per step. Instead, we report the total generation throughput aggregated over all rollout instances. The pull-based weight transfer enables RLBoost to immediately use a new instance for rollout, while the tra-

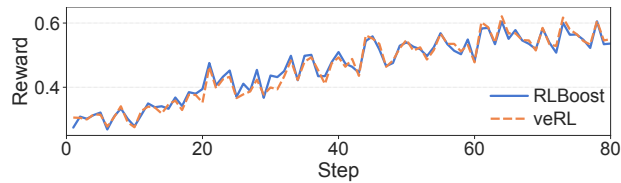


Figure 15: **[Algorithm integrity]**: Rewards on Qwen-8B.

ditional synchronized weight transfer only makes use of new instances in the next step, causing substantial resource waste. We note that the generation throughput gradually drops after the initial surge when a new instance is added, resulting from the growing context length in the continuous batch as tokens are generated [\[22\]](#). We also demonstrate the impact of pull-based weight transfer agents when instances are preempted and immediately restart on another available node in [Appendix D](#).

6.4.3 Impacts of Fault Handling

To study how efficiently RLBoost handles request failures upon preemptions, we construct a scenario where 3 out of 6 rollout instances are preempted simultaneously at different points during the rollout of a step. We compare two fault handling strategies for requests routed to the preempted instances: our solution in [§4.2](#) that collects responses at token granularity and migrates partially generated rollout requests (denoted as *migrate*); and the traditional strategy that only collects complete responses, resulting in recomputing the entire request on a healthy instance (denoted as *recompute*).

We analyze how different strategies impact step time. In [Figure 14](#), we report the step time overhead (increase) compared to the case with no preemption. We demonstrate two settings: preemptions at 100s (early point) and 200s (mid point) after the start of a step. For earlier preemptions at 100s, a limited number of tokens are generated for most requests, hence the cost of recomputation is not significant and migrate only reduces the overhead by 33%. However, for preemptions at 200s, where many requests have already generated a large number of tokens, migrate reduces the overhead by 76%.

6.5 Algorithm Integrity

Unlike many disaggregated frameworks [\[11, 14, 16, 53\]](#), RLBoost maintains the well established synchronous RL algorithms. We compare the training reward curve of RLBoost with veRL in [Figure 15](#), where RLBoost is trained under the $2 \times H100$ instance availability in [Figure 7](#). We present the curve for 8B since more steps can be trained within the same amount of time. The 14B and 32B models exhibit similar patterns.

As RLBoost makes no modifications to the synchronous GRPO algorithm and uses the same training settings, the reward curve of RLBoost closely matches that of veRL. We note that the reward eval at each step are not exactly the same. This is due to the temperature-based sampling in rollout, which

is further complicated by the well-known nondeterministic behaviors in current LLM inference frameworks [15].

7 Discussion

Supporting heterogeneous hardware. Since rollout instances operate independently from one another, RLBoost can exploit heterogeneous computing resources to further improve cost efficiency. Each rollout instance can be configured with different accelerators (varying GPU models or even TPUs) and parallelism strategies (different TP sizes). By leveraging real-time load and step time statistics in [Algorithm 1](#) and [2](#), RLBoost can match each instance’s computing capability, maximizing effective compute while ensuring load balance across heterogeneous instances.

Weight transfer optimization. RLBoost currently implements a bipartite point-to-point weight transfer mechanism, where each rollout instance directly fetches weights from one of the training nodes. This strategy already fully utilizes all frontend NICs on training nodes within the same data-center setting, where multiple network routes exist between rollout instances and the training cluster. Scaling to larger models would require more and larger GPU instances for both training and rollout. Hence, the per-instance and aggregated bandwidth increase accordingly. Consequently, weight transfer remains a small fraction of per-step time, which itself grows with model size. In addition, RLBoost can incorporate weight compression techniques [45] to transfer compressed weight deltas between consecutive steps.

RLBoost can be further optimized by building a dynamic broadcast tree [56], where only a subset of rollout instances retrieves weights from the training cluster while others receive them from peers. This optimization is beneficial when rollout instances are in a different datacenter, where cross-datacenter bandwidth is the bottleneck. We leave that for future work.

Asynchronous RL Although RLBoost mainly targets well-established synchronous RL algorithms, it can be easily extended to support asynchronous RL algorithms, e.g., one-step off-policy [53] or fully asynchronous [11], by not enforcing instances to use the latest weights in the rollout manager.

8 Related Work

LLM training and serving on preemptible instances. Recent systems use preemptible instances to cut the cost of LLM pre-training and inference. For training, prior work achieves resilience via redundancy, migrating across parallelization strategies after preemptions and assuming spare replicas preserve lost model states [10, 19, 38, 43]. Under frequent preemptions, they often revert to checkpoint restarts and can stall when all replicas are lost, and they do not support fully non-redundant strategies such as FSDP [47]. Hence, preemptible-instance training remains inefficient and unstable.

There is also a series of systems that leverage preemptible instances for online LLM serving [23, 25], but they focus primarily on optimizing service availability and latency SLOs. In contrast, RL workloads require optimizing the total execution time of each training step, which encompasses both interdependent rollout and training stages.

RL frameworks for LLMs. A number of RL frameworks have been specifically designed for LLMs. NeMo-Aligner [34] and OpenRLHF [18] are among the earliest. They apply the disaggregated architecture and suffer from resource under-utilization due to serial dependencies between stages. Co-located frameworks like veRL [35], ReaL [24], and RLHFuse [54] are hence proposed to improve resource efficiency. veRL [35] combines single-controller and multi-controller paradigms to efficiently drive the execution of different stages. RLHFuse [54] introduces stage fusion to further improve GPU compute efficiency. Recently, the resource coupling issue of co-located frameworks has led to resurgent interests in the disaggregated architecture [11, 14, 16, 53]. StreamRL [53] proposes a one-step off-policy training pipeline, where rollout uses stale weights that are one step behind. AReaL [11] further relaxes this to fully asynchronous training. RhymeRL [16] adopts the one-step off-policy paradigm and leverages speculative decoding to accelerate rollout. Notably, StreamRL [53] also proposes dynamically adjusting GPU resources allocated to training and rollout to elastically maintain balanced execution. However, it still assumes a fixed resource pool and cannot support preemptible resources, where resource availability is unpredictable.

9 Conclusion

In this paper, we present RLBoost, a hybrid RL framework that harvests preemptible GPU resources for high-throughput and cost-efficient RL on LLMs. RLBoost maintains a reserved (on-demand) training cluster while opportunistically offloading rollout workloads to preemptible instances. Through adaptive rollout offload with partial response seeding, RLBoost dynamically balances workloads between the training cluster and remote instances based on real-time resource availability. The pull-based weight transfer mechanism enables newly allocated instances to quickly join ongoing rollout, while token-level response collection minimizes preemption overhead and enables continuous load balancing. Experiments show RLBoost accelerates RL training by up to 1.97× while improving cost efficiency by up to 49% compared to using only on-demand GPU resources, all while maintaining synchronous RL algorithms.

Acknowledgements We thank the anonymous NSDI reviewers and our shepherd, Hitesh Ballani, for their constructive feedback and suggestions. We also thank Google Cloud Platform and Kaiyuan Zhang for their support and infrastructure resources.

References

- [1] Cluster fragmentation. <https://www.rc.fas.harvard.edu/blog/cluster-fragmentation/>, 2022.
- [2] Axum. <https://github.com/tokio-rs/axum>, 2025.
- [3] Claude 4. <https://www.anthropic.com/news/claude-4>, 2025.
- [4] Grok 4. <https://x.ai/news/grok-4>, 2025.
- [5] Nixl. <https://github.com/ai-dynamo/nixl>, 2025.
- [6] Practical tips for preventing gpu fragmentation for volcano scheduler. <https://developer.nvidia.com/blog/practical-tips-for-preventing-gpu-fragmentation-for-volcano-scheduler/>, 2025.
- [7] Tokio. <https://tokio.rs/>, 2025.
- [8] Amazon Web Services. Amazon EC2 P5 Instances. <https://aws.amazon.com/ec2/instance-types/p5/>, 2024. Accessed: September 17, 2025.
- [9] Udbhav Bamba, Minghao Fang, Yifan Yu, Haizhong Zheng, and Fan Lai. Xrpo: Pushing the limits of grpo with targeted exploration and exploitation. *arXiv preprint arXiv:2510.06672*, 2025.
- [10] Jiangfei Duan, Ziang Song, Xupeng Miao, Xiaoli Xi, Dahua Lin, Harry Xu, Minjia Zhang, and Zhihao Jia. Parcae: Proactive, {Liveput-Optimized} {DNN} training on preemptible instances. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1121–1139, 2024.
- [11] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, et al. Areal: A large-scale asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025.
- [12] Google Cloud. GPU network bandwidth. <https://cloud.google.com/compute/docs/gpus/gpu-network-bandwidth>. Accessed: August 26, 2025.
- [13] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
- [14] Zhenyu Han, Ansheng You, Haibo Wang, Kui Luo, Guang Yang, Wenqi Shi, Menglong Chen, Sicheng Zhang, Zeshun Lan, Chunshi Deng, et al. Asyncflow: An asynchronous streaming rl framework for efficient llm post-training. *arXiv preprint arXiv:2507.01663*, 2025.
- [15] Horace He and Thinking Machines Lab. Defeating nondeterminism in llm inference. *Thinking Machines Lab: Connectionism*, 2025. <https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/>.
- [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. *arXiv preprint arXiv:2508.18588*, 2025.
- [17] Tao He, Xue Li, Zhibin Wang, Kun Qian, Jingbo Xu, Wenyuan Yu, and Jingren Zhou. Unicorn: Economizing self-healing llm training at scale. *arXiv preprint arXiv:2401.00134*, 2023.
- [18] Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Zilin Zhu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, et al. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. *arXiv preprint arXiv:2405.11143*, 2024.
- [19] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 382–395, 2023.
- [20] Nils Knieling. Amazon EC2 Instance Types. <https://aws-pricing.com/instances.html>. Accessed: September 12, 2025.
- [21] Nils Knieling. GCE Machine Types in Google Cloud Platform. <https://gcloud-compute.com/instances.html>. Accessed: September 12, 2025.
- [22] 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*, pages 611–626, 2023.
- [23] Ziming Mao, Tian Xia, Zhanghao Wu, Wei-Lin Chiang, Tyler Griggs, Romil Bhardwaj, Zongheng Yang, Scott Shenker, and Ion Stoica. Skyserve: Serving ai models across regions and clouds with spot instances. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 159–175, 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. *arXiv e-prints*, pages arXiv–2406, 2024.

- [25] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1112–1127, 2024.
- [26] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, et al. Ras: continuously optimized region-wide datacenter resource allocation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 505–520, 2021.
- [27] OpenR1 Team. OpenR1-Math-220k. <https://huggingface.co/datasets/open-r1/OpenR1-Math-220k>, 2025.
- [28] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.
- [29] Vignesh Prabhakar, Md Amirul Islam, Adam Atanas, Yao-Ting Wang, Joah Han, Aastha Jhunjhunwala, Rucha Apte, Robert Clark, Kang Xu, Zihan Wang, et al. Omniscience: A domain-specialized llm for scientific reasoning and discovery. *arXiv preprint arXiv:2503.17604*, 2025.
- [30] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, 2025.
- [31] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. {CASSINI}:{Network-Aware} job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1403–1420, 2024.
- [32] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [33] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, and Junxiao Song. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *ArXiv preprint*, 2024.
- [34] Gerald Shen, Zhilin Wang, Olivier Delalleau, Jiaqi Zeng, Yi Dong, Daniel Egert, Shengyang Sun, Jimmy Zhang, Sahil Jain, Ali Taghibakhshi, et al. Nemo-aligner: Scalable toolkit for efficient model alignment. *arXiv preprint arXiv:2405.01481*, 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*, pages 1279–1297, 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. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19*, 2019.
- [37] Qwen Team. Qwen3 technical report, 2025.
- [38] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, 2023.
- [39] Borui Wan, Gaohong Liu, Zuquan Song, Jun Wang, Yun Zhang, Guangming Sheng, Shuguang Wang, Houmin Wei, Chenyuan Wang, Weiqiang Lou, et al. Robust llm training infrastructure at bytedance. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 186–203, 2025.
- [40] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 364–381, 2023.
- [41] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of fragmentation: Scheduling {GPU-Sharing} workloads with fragmentation gradient descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 995–1008, 2023.
- [42] Junde Wu, Jiayuan Zhu, Yuyuan Liu, Min Xu, and Yueming Jin. Agentic reasoning: A streamlined framework for enhancing llm reasoning with agentic tools. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 28489–28503, 2025.

- [43] Yongji Wu, Wenjie Qu, Xueshen Liu, Tianyang Tao, Yifan Qiao, Zhuang Wang, Wei Bai, Yuan Tian, Jiaheng Zhang, Z Morley Mao, et al. Lazarus: Resilient and elastic training of mixture-of-experts models. *arXiv preprint arXiv:2407.04656*, 2024.
- [44] Kaizhuo Yan, Yingjie Yu, Yifan Yu, Haizhong Zheng, and Fan Lai. Oppo: Accelerating ppo-based rlhf via pipeline overlap. In *International Conference on Learning Representations*, 2026.
- [45] Xiaozhe Yao, Qinghao Hu, and Ana Klimovic. Deltazip: Efficient serving of multiple full-model-tuned llms. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 110–127, 2025.
- [46] Qiyang Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, et al. Dapo: An open-source llm reinforcement learning system at scale. *ArXiv preprint*, 2025.
- [47] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel. In *Proceedings of the VLDB Endowment*, volume 16, pages 3848–3860, 2023.
- [48] Yuzhong Zhao, Yue Liu, Junpeng Liu, Jingye Chen, Xun Wu, Yaru Hao, Tengchao Lv, Shaohan Huang, Lei Cui, Qixiang Ye, Fang Wan, and Furu Wei. Geometric-mean policy optimization. *ArXiv preprint*, 2025.
- [49] Haizhong Zheng, Jiawei Zhao, and Beidi Chen. Prosperity before collapse: How far can off-policy rl reach with stale data on llms? In *International Conference on Learning Representations*, 2026.
- [50] 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. In *Advances in neural information processing systems*, 2025.
- [51] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.
- [52] Yuxiang Zheng, Dayuan Fu, Xiangkun Hu, Xiaojie Cai, Lyumanshan Ye, Pengrui Lu, and Pengfei Liu. Deep-researcher: Scaling deep research via reinforcement learning in real-world environments. *arXiv preprint arXiv:2504.03160*, 2025.
- [53] Yinmin Zhong, Zili Zhang, Xiaoni Song, Hanpeng Hu, Chao Jin, Bingyang Wu, Nuo Chen, Yukun Chen, Yu Zhou, Changyi Wan, et al. Streamrl: Scalable, heterogeneous, and elastic rl for llms with disaggregated stream generation. *arXiv preprint arXiv:2504.15930*, 2025.
- [54] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, et al. Optimizing rlhf training for large language models with stage fusion. *arXiv preprint arXiv:2409.13221*, 2024.
- [55] Yifei Zhou, Song Jiang, Yuandong Tian, Jason Weston, Sergey Levine, Sainbayar Sukhbaatar, and Xian Li. Sweet-rl: Training multi-turn llm agents on collaborative reasoning tasks. *arXiv preprint arXiv:2503.15478*, 2025.
- [56] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. Hoplite: efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 641–656, 2021.

Table 3: Configurations of machines on public clouds for cost efficiency analysis.

Provider	Machine Type	#H100	Provision	Cost/hour
GCP	a3-highgpu-8g	8	Standard	\$101.62
	a3-highgpu-8g	8	Spot	\$23.78
	a3-highgpu-2g	2	Spot	\$6.75
AWS	p5.48xlarge	8	Standard	\$65.96
	p5.48xlarge	8	Spot	\$20.24
	p5.4xlarge	1	Spot	\$1.76

A Cloud Instance Cost

We calculate the average cost of instances with H100 across different regions on both AWS and GCP following [20, 21]. The results are listed in Table 3. Both providers offer standard and spot provision options with large price gaps, and users can run RLBoost to boost RL throughput.

A full a3-highgpu-8g is equipped with a 200 Gbps frontend NIC and four 200 Gbps backend NICs, while a a3-highgpu-2g can only access a 50 Gbps frontend vNIC [12]. On the other hand, a p5.48xlarge supports 3200 Gbps EFA network, while a p5.4xlarge only supports 100 Gbps EFA network [8]. The limited bandwidth on fragmented instances makes them less feasible for distributed training, but their high availability and affordable price make them a perfect fit for rollout.

For cost efficiency calculation, the average hourly cost of standard instance with 8 H100s on public clouds is

$$$(101.62 + 65.96)/2 \approx \$83.79,$$

and the average hourly cost of 2 spot H100s is

$$$(23.78/4 + 6.75 + 20.24/4 + 1.76 \times 2)/4 \approx \$5.32$$

Figure 16 quantifies how spot GPU discounts affect RLBoost’s cost efficiency across the three trace segments. We define spot GPU discount as the normalized per-GPU discount relative to on-demand: 0% means equal price, and 100% means effectively free GPUs (e.g., on-premise infrastructure). With no discount (0%), RLBoost’s cost efficiency lies between veRL and veRL.2x because RLBoost provisions additional (rollout) resources beyond veRL. With as little as ~30% savings, RLBoost improves cost efficiency while also increasing training throughput.

B Model Configuration

In our evaluation, we use 8B/14B/32B models from Qwen3 family [37]. The models’ details are listed in Table 4.

Table 4: Model configurations.

	Layers	Q heads	K/V heads	Hidden size
Qwen3-8B	32	32	8	4096
Qwen3-14B	48	48	8	5120
Qwen3-32B	64	40	8	5120

Table 5: Overview of the 3 segments of the spot instance trace.

Traces	Segment A	Segment B	Segment C
Availability	High	Low	High
Preemption Intensity	High	High	Low
#Avg. Instances	6.53	4.58	6.06
#Allocations	13	8	6
#Preemptions	8	9	2

C Characteristics of Preemptible Instance Traces

Our preemptible instance trace is based on [38]. To match our resource constraints, we randomly sampled 50% of the instances from the original trace while preserving their individual allocation and preemption event histories. We evaluate RLBoost on three representative segments shown in Figure 7. Their characteristics are listed in Table 5. For best cost efficiency, we control the maximum number of running preemptible instances, N_{prem} , according to Algorithm 1, instead of allocating all available instances. When an instance is preempted, i.e., remove event in the trace, the trace replayer will shut down the instance and immediately start a new one if available.

D Impact of Weight Transfer Paradigm on Availability Spikes

Besides allowing a newly available instance to quickly participate in the current step’s rollout, the pull-based weight transfer agents also stabilize the throughput on availability spikes. As we observed in Figure 7, instances can be occasionally preempted, but a new one can be immediately allocated. In Figure 17, we construct a scenario where three rollout instances are preempted and restart consecutively within a step. Because the synchronous weight transfer logic updates weights between each step, a restarted instance cannot join the current step rollout, and the throughput drops accordingly. In contrast, with our pull-based weight transfer agents, the restarted instances immediately pull latest model weights from the agents and begin rollout, and we can observe the throughput quickly recovers.

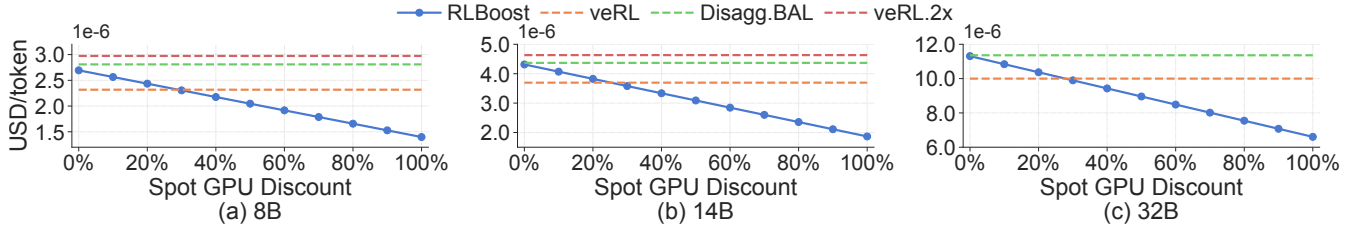


Figure 16: **[Cost sensitivity]**: The impact of spot GPU cost saving to the overall cost efficiency in Figure 10(d).

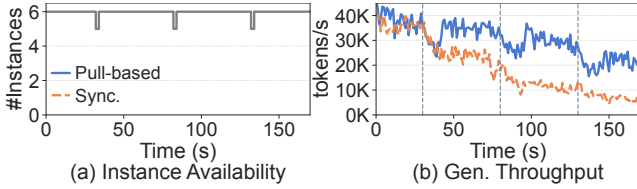


Figure 17: **[Ablation study]**: Comparing pull-based and synchronized weight transfer as instances restart within a step. We use Qwen3-14B.

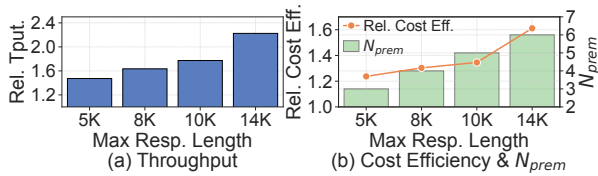


Figure 18: **[Cost efficiency]**: Relative throughput and cost efficiency of RLBoost w.r.t. veRL on Qwen-14B using a single 8xH100 instance as the training cluster, under different max response length with corresponding optimal N_{preem} .

E Impact of Maximum Response Length on Cost Efficiency

We evaluate RLBoost under different maximum response lengths from 5K to 14K, and record the relative throughput and cost efficiency over veRL running on reserved instances in Figure 18. Due to the autoregressive computation pattern [22] of LLM inference, rollout becomes more time-consuming than training as length grows. RLBoost automatically scales the number of preemptible instances to match the workload as in Algorithm 1. As the optimal number of preemptible instances (N_{preem}) increases from 3 to 6, RLBoost boosts the relative throughput by 1.47x–2.22x and improves the relative cost efficiency by 1.24x–1.61x.