USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

DistRS: Disaggregated Reward Service for RLVR with Batch-Level Constraint

Ruidong Zhu, *School of Computer Science, Peking University;* Mingcong Han, *ByteDance Seed;* Yinmin Zhong, *School of Computer Science, Peking University;* Wencong Xiao, *ByteDance Seed;* Xuanzhe Liu and Xin Jin, *School of Computer Science, Peking University*

https://www.usenix.org/conference/nsdi26/presentation/zhu-ruidong

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

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

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

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

DistRS: Disaggregated Reward Service for RLVR with Batch-Level Constraint

Ruidong Zhu[1] *Mingcong Han*[2] *Yinmin Zhong*[1] *Wencong Xiao*[2] *Xuanzhe Liu*[1] *Xin Jin*[1]
[1]*School of Computer Science, Peking University* [2]*ByteDance Seed*

Abstract

Reinforcement Learning with Verifiable Rewards (RLVR) has emerged as a key post-training paradigm for enhancing the capabilities of large language models (LLMs). As the complexity increases and resource consumption grows, reward computation is becoming a critical workload in the RLVR training process.

We present DistRS, a disaggregated reward service framework designed to provide resource-efficient reward computation for RLVR training. Through the analysis of a real RLVR training task, we observe that the reward service faces a highly dynamic workload, motivating the need for elasticity and multi-tenancy. DistRS leverages request-level flexibility from the *request-in, batch-out* characteristic of reward computation to design more resource-efficient scaling and scheduling policies. Specifically, DistRS establishes a batch-level constraint for each training task that relaxes latency requirements at the request level. Building on this foundation, we design a history-based resource scaling policy and a batch-level priority-based request scheduling policy. In addition, DistRS incorporates a timeout-aware mechanism to adjust resource allocation, thereby mitigating the impact of deviations between history and actual execution. We evaluate DistRS with real-world RLVR training tasks and the results demonstrate that DistRS reduces resource consumption by up to 3.79× while incurring minimal overhead on training progress.

1 Introduction

Reinforcement learning (RL) has become an indispensable step in the post-training process of large language models (LLMs). Early applications of RL to LLMs primarily focused on aligning models with human preferences by introducing a reward model to score outputs, a paradigm known as Reinforcement Learning from Human Feedback (RLHF) [1, 2]. More recent studies [3–7] demonstrate that in domains where correctness can be automatically verified—such as mathematics and code—rewards can be directly computed through rule-based methods, substantially improving both reasoning ability and task performance. This paradigm is referred to as *Reinforcement Learning with Verifiable Rewards* (RLVR).

The key distinction between RLVR and RLHF lies in the reward computation process. As shown in Figure 1, the workflow of RLVR largely mirrors that of RLHF, with each iteration consisting of a rollout phase, reward computation, and a training phase. In RLHF, rewards are computed via a forward pass of the *reward model*, and this computation is typically executed in batches on the same resources used for rollout and training, as illustrated in Figure 2(a). However,

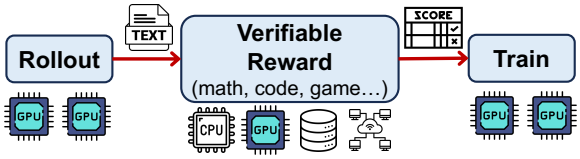


Figure 1: Workflow of RLVR.

verifiable reward computation in RLVR can be considerably more complex and resource-intensive. For example, when training models to generate CUDA code [8], reward computation requires compiling the code on CPUs and executing the code on GPUs to verify correctness if compilation succeeds (Figure 4(a)). In production training tasks, this process can use thousands of CPUs and hundreds of GPUs, with each request taking up to several minutes to complete. For more CPU-intensive workloads, reward computation can simultaneously occupy hundreds of thousands of CPUs during training, making its overall cost comparable to that of training itself.

As reward computation grows increasingly intensive, its underlying paradigm is undergoing rapid evolution. Early frameworks for RLVR [9, 10] adopted an approach similar to that of RLHF, performing reward computation collectively after rollout (Figure 2(b)). However, the substantial complexity of reward computation introduces a range of issues, including mismatches of resource types, resource contention, and limited computational concurrency. Consequently, recent works [11–13], as well as our internal practice, have decoupled reward computation from the RL training framework and deployed it as an independent service. The disaggregated reward service further enables request-level reward computation. As shown in Figure 2(c), once an input sequence completes its rollout, the corresponding reward computation request can be sent to the service and processed in real time.

In this paper, we aim at developing a disaggregated reward service that minimizes resource consumption while introducing negligible disruption to training tasks. However, our analysis of the reward service workload in a real RL training task (§3) reveals that achieving this objective remains highly challenging. We observe that the reward service faces a highly dynamic workload. In terms of arrival patterns, reward computation requests are generated only after an output sequence is fully generated. Consequently, no reward request arises in the early rollout steps or during training phases when collocated training architecture (§2.1) is employed. Furthermore, the arrival pattern exhibits a long-tail characteristic, as a small fraction of input sequences generate relatively long outputs [14–16], resulting in reward computation requests arriving later. In terms of resource demands, they vary signif-

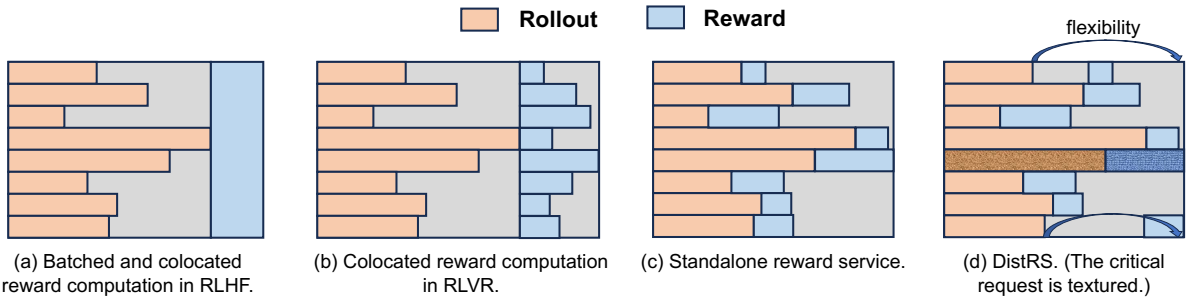


Figure 2: Comparison of four types of reward computation.

icantly as training progresses, owing to the evolving output behavior of the LLM during the process. Different outputs may require vastly different amounts of resources and computation time for reward evaluation. The dynamic nature of the workload renders static resource allocation inefficient, resulting in either wasted resources or increased training delays.

The highly dynamic workload motivates us to adopt an elastic and multi-tenant service. However, the objective of traditional elastic multi-tenant services is to satisfy the service level objective (SLO) of *individual requests* [17, 18]. Our further analysis shows that, due to the unique *request-in, batch-out* nature of reward service, the objective for reward service can be relaxed to a *batch-level constraint*. Consequently, compared to traditional elastic multi-tenant services, we can exploit greater flexibility. The *request-in* property originates from the autoregressive generation of LLMs, where requests are generated and dispatched to the reward service in a sequential manner. The *batch-out* property arises because the training phase cannot advance until the reward service has returned responses for all requests within the batch. As illustrated in Figure 2(d), the *critical request* is defined as the last request in a batch to complete if no queuing occurs. To minimize the impact on training, only the *critical request* must be executed immediately. In contrast, the other requests can be deferred and scheduled flexibly at any point between their arrival and the completion of the *critical request*.

To this end, we propose DistRS, a disaggregated reward service framework aiming at both resource efficiency and minimal impact on training. In addition to elasticity and multi-tenancy, we build a *batch-level constraint* that relaxes the latency requirement at the request level, and take this as the foundation for designing our scaling and scheduling policies.

We propose a history-based resource scaling algorithm for individual tasks that operates at the batch granularity. By leveraging the history of previous batches, the algorithm predicts the arrival times and execution times of requests in the upcoming batch, and accordingly determines the minimum resources required to complete all requests under the batch-level constraint. The main challenge lies in handling requests whose execution times are significantly underestimated, which may lead to incorrect queuing decisions and extra delays. To mitigate this issue, we design a timeout-aware mechanism that dynamically adjusts resource allocation based on predefined timeout thresholds, thereby ensuring efficient resource utilization while preserving training progress.

We then extend the algorithm to the multi-tenant setting, where scaling decisions are still made at batch boundaries. To further improve the effectiveness of resource sharing, we incorporate the batch-level constraint into the request scheduling policy. In particular, we assign priorities to batches based on the estimated completion time of the batch for each task, and schedule request execution according to these priorities.

We implement DistRS and integrate it with an internal RL training framework. Experimental results on real-world training tasks demonstrate that DistRS reduces GPU consumption in the reward service by up to 3.79× and CPU consumption by up to 2.16×, while imposing minimal impact on training.

In summary, our contributions are as follows:

- We present a systematic workload analysis of reward computation in RLVR, revealing its highly dynamic nature and the unique *request-in, batch-out* characteristic.
- We propose DistRS, a disaggregated reward service framework that leverages request-level flexibility and batch-level constraints to design resource scaling and request scheduling policies that minimize resource consumption while preserving training progress.
- We conduct experiments on real RL training tasks and the results demonstrate up to 3.79× reduction in resource consumption with minimal impact on training.

2 Background and Motivation

In this section, we first introduce the process and system architecture of RLVR in §2.1, followed by a discussion of the emerging paradigm of reward-as-a-service §2.2. Finally, we present our motivation and objectives in §2.3.

2.1 Reinforcement Learning with Verifiable Rewards

RLHF heavily relies on a reward model to evaluate LLM outputs. However, the reward model's judgments are not always reliable, as it can be susceptible to reward hacking [19, 20]. To address this limitation, recent works [3, 5, 6] propose Reinforcement Learning with Verifiable Rewards (RLVR) for LLM training. In domains such as math, code, and knowledge-based question answering, simple rule-based methods can

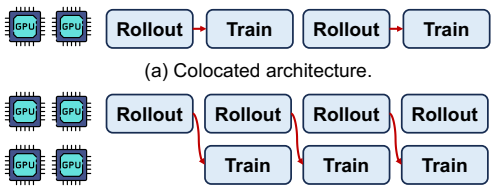


Figure 3: Two architectures of LLM RL training frameworks.

provide reward scores that accurately reflect the correctness of LLM outputs. RLVR has been widely applied in these areas to enhance LLMs' reasoning ability, tool-use capabilities, and task-specific performance.

Figure 1 illustrates a training iteration in RLVR. First, given a batch of inputs, the model performs autoregressive token generation, a phase commonly referred to as the *rollout* in RL algorithms. After the rollout, relevant portions of the model's outputs are extracted for verifiable reward computation. For instance, for a math problem, the model first performs reasoning and thinking, and then outputs an answer; the reward is computed by comparing the generated answer with the reference solution. Through format matching and answer verification, a final score is assigned to the output. This score is then transformed on the training side and used to guide model parameter updates via algorithms such as PPO and GRPO [6, 21–23].

In terms of architecture for LLM RL training frameworks, there are two main paradigms: colocated and disaggregated. We present them in Figure 3, where the weight update and context switching are omitted for simplicity. In the colocated architecture (Figure 3(a)), the rollout and training phases share the same set of resources and execute alternately. In contrast, the disaggregated architecture (Figure 3(b)) allocates separate resources for rollout and training, with the two phases communicating over the network for data transfer and parameter synchronization. To optimize resource utilization in disaggregated setups, techniques such as one-step staleness and streaming [15, 24] have been proposed to enable pipeline execution of rollout and training.

2.2 Reward-as-a-Service

Since reward computation in RLHF is essentially a forward pass of a model, its computational load remains relatively fixed across different training tasks, and it primarily relies on GPUs, similar to other training phases. Consequently, reward computation in RLHF often shares resources with other phases, utilizing GPUs freed after the rollout phase to perform computations collectively. RLHFuse [14] fuses the rollout and reward computation phases, but they still share the same set of resources. When RL frameworks such as verl [9] and OpenRLHF [10] began supporting RLVR, they adopted a similar approach to reward computation, placing it after the entire rollout phase completes and performing it in batches, as illustrated in Figure 2(b).

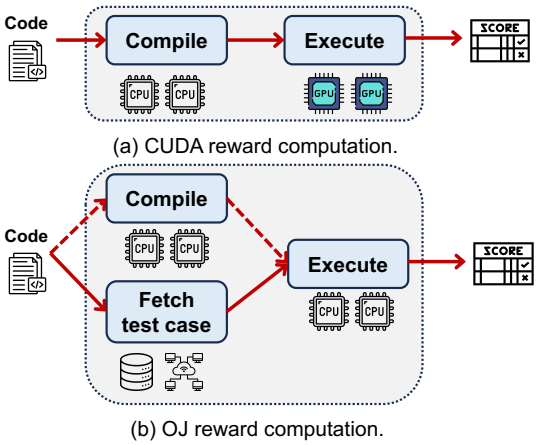
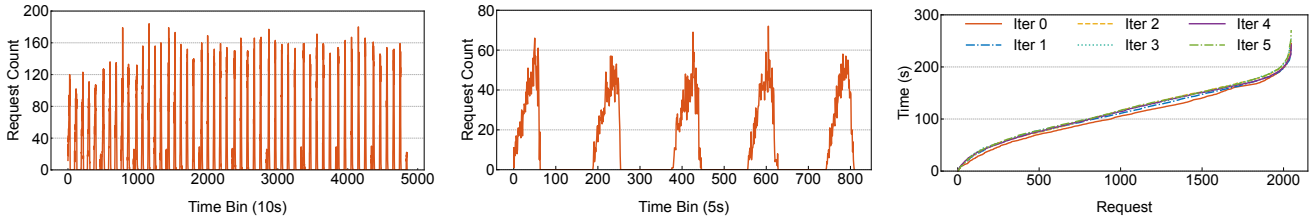


Figure 4: Examples of verifiable reward computation.

The main difference between RLVR and RLHF is the reward computation. In RLVR, reward computation may involve a workflow, with each request potentially requiring different inputs and traversing multiple stages. Furthermore, it consumes a wide variety of resources [8, 25, 26], including CPU, GPU, network, and storage, which may differ from those primarily used in RL training. We illustrate the process of verifiable rewards with two concrete examples. Figure 4(a) shows the reward computation for RL tasks targeting CUDA code generation [8]: once an input sequence completes its rollout, the generated CUDA code is extracted via simple string matching, compiled on the CPU, and, if compilation succeeds, executed on the GPU. Figure 4(b) depicts the reward computation for RL training on online judge (OJ) programming problems [13], which require code to be automatically tested against predefined inputs and outputs [27, 28]. The generated code is first extracted. If compilation is required (e.g., C++), the code is compiled, whereas for interpreted languages (e.g., Python), this step is omitted. The system then retrieves the corresponding test cases from remote storage, executes the code, and verifies its correctness. Colocating reward computation with rollout and training can cause resource contention and limit concurrency, making the reward computation phase a bottleneck.

To support resource-intensive reward computations and improve their scalability, recent works and our internal practice have started deploying reward computation as a standalone service, i.e., reward-as-a-service. We collectively refer to the rollout and training phases and their resources as the trainer, while a disaggregated reward service receives reward computation requests from the trainer and returns the results. Decoupled from the trainer, the reward service can leverage a dedicated resource pool and conveniently provide request-level reward computation, as depicted in Figure 2(c). As model capabilities continue to expand and application domains diversify, reward computation will become increasingly complex. Consequently, we expect disaggregated reward-as-a-service to become the mainstream approach for RLVR.



(a) Arrival rate during 50 iterations with a bin of 10s. (b) Arrival rate during 5 iterations with a bin of 5s. (c) Request arrival time for 6 iterations.

Figure 5: Arrival pattern of reward service during training.

2.3 Motivation and Objectives

As reward computation becomes more complex, the resource demands of the reward service also grow. For instance, in our company's RL tasks that train models to generate CUDA kernels, each reward computation requires tens of seconds for CPU compilation and several additional seconds for GPU execution. Furthermore, to evaluate each kernel accurately and fairly, it must exclusively occupy a GPU during execution. As a result, the resource demand for reward service is approaching that of the trainer. It is foreseeable that training LLMs to handle more complex tasks—such as building distributed systems or generating project-level code—will require even greater resources and longer computation times. Fundamentally, under the RLVR paradigm, a task can only be effectively trained if it can be evaluated efficiently and accurately, with timely reward feedback [29].

As reward computation grows into a critical component of the overall training workload, attention must be directed not only to the optimization of the trainer but also to the resource management of the reward service. In this work, our objective is to design a disaggregated reward service that (i) imposes minimal disruption to training progress and (ii) consumes as few resources as possible.

To achieve this goal, we first conduct an in-depth analysis of the reward computation workload generated by a real RL task in §3. It shows that the reward service faces highly dynamic workloads. The cyclic execution of training phases leads to dynamic variations in the request arrival pattern, while the evolving output behavior of the model during training further causes the resource requirements for reward computation to change accordingly. Consequently, static resource management either slows down training significantly due to insufficient allocation or causes severe resource waste due to overallocation.

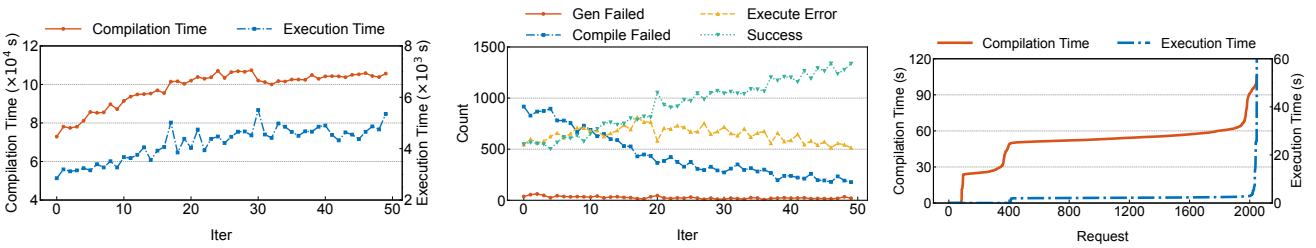
3 Workload Analysis and Insight

In this section, we analyze the workload of reward computation in a real RL training task for CUDA code generation and present our key insights. The task adopts a colocated architecture [9], runs for 50 iterations with a batch size of 2048 per iteration, and is trained using GRPO [22]. The maximum rollout generation length is set to 16K tokens.

3.1 Workload Characteristics

Characteristics of arrival pattern. First, we analyze the arrival pattern of reward computation requests during training. Figure 5(a) shows the overall arrival rate of requests throughout the training process, using a bin size of 10 seconds. The arrival rate exhibits substantial variation, peaking at over 180 requests per 10 seconds while also dropping to zero. Figure 5(b) zooms in on five iterations with finer temporal granularity. Because this task adopts a colocated architecture, no reward computation requests are generated during the training phase. Furthermore, at the beginning of each rollout, reward requests are sparse or absent. As a result, as shown in Figure 5(b), roughly half of the time within each iteration sees no reward requests arriving. Once the rollout progresses, the number of requests rapidly rises to a peak and then gradually decays, forming a long tail. Figure 5(c) normalizes each iteration by aligning the first reward request to the same start time and plotting subsequent request arrivals in order. It demonstrates the similarity in the arrival patterns of reward requests across adjacent iterations, as their curves overlap closely. The reason for this similarity is that the distribution of output lengths during the rollout phase evolves relatively slowly across adjacent iterations [6, 30]. The figure also reveals the long-tail property of the rollout phase, as the arrival interval of the final requests, i.e., the slope of the curve, increases significantly.

Characteristics of resource consumption. Next, we characterize the resource demand of reward computation during training. We measure the total CPU node time and GPU node time (defined as the number of CPUs/GPUs multiplied by their running time) consumed per iteration. Figure 6(a) reports the results of each iteration, where CPU node time corresponds to compilation and GPU node time to execution. As training progresses, both CPU and GPU node times per iteration exhibit an increasing trend. To explain this phenomenon, we further analyze the distribution of requests across different states in each iteration, as shown in Figure 6(b). The request states are categorized as follows: *Generation failed*: the model fails to produce CUDA code that passes simple text-based checks; *Compile failed*: the generated code cannot be compiled; *Execute error*: the code compiles successfully but fails during execution or produces incorrect results; *Success*: the



(a) Total compilation and execution time of each iteration.

(b) State distribution of each iteration.

(c) Compilation and execution time of each request in an iteration.

Figure 6: Resource consumption of reward service during training.

code compiles and executes successfully, and produces correct outputs. As the LLM gradually learns to generate valid CUDA code during training, the likelihood that the generated code can be compiled and executed increases. Therefore, the proportion of requests in the *Generation failed* and *Compile failed* categories steadily decreases. Meanwhile, *Execute error* requests first increase and then decline, and *Success* requests grow significantly. Therefore, the observed increase in resource consumption in Figure 6(a) stems from the LLM's evolving output behavior during training. Although we focus here on the CUDA code generation task, we argue that this workload variation is representative of RLVR in general. Many reward computations involve multiple stages, where later stages are only reached if earlier ones succeed, and LLMs typically learn to progress through these stages gradually.

We also analyze the compilation and execution times of individual requests within a batch, as shown in Figure 6(c). Since some requests fail to generate code or pass compilation, their compilation or execution times are recorded as zero. For requests that successfully enter compilation or execution, most exhibit relatively consistent runtimes, although a small fraction take significantly longer to complete. In fact, many of these long-running requests terminate only upon reaching the timeout thresholds set during training. This occurs because the LLM may output arbitrary content—for example, code containing infinite loops—that fails to terminate during execution. Unlike data analytics and similar workloads, where execution time can be accurately estimated by profiling [31, 32], in our setting, it is infeasible to reliably predict request runtime prior to execution. However, the reward service cannot allow unbounded execution. It terminates the reward computation if the running time exceeds a timeout threshold and assigns the request a negative reward. In the CUDA reward service we studied, normal requests typically require around 50 seconds for compilation and a few seconds for execution, while the timeout thresholds are set to 120 seconds for compilation and 60 seconds for execution.

Summary and outlook. The above analysis demonstrates that during RLVR training, both the workload arrival patterns and the resource demands of the reward service vary dynamically over time. Therefore, a statically configured reward service not only incurs significant resource idleness when no requests are present, but also fails to adapt to changing demands, leading to either wasted resources or stalled training progress. To address this, we envision an *elastic, multi-tenant* reward service. Elasticity allows the service to dynamically adjust resource allocation in response to workload fluctuations, while multi-tenancy further promotes resource sharing and reuse. Despite the promise of these mechanisms, a key challenge remains: determining what kind of policy should govern their application to effectively achieve our goals.

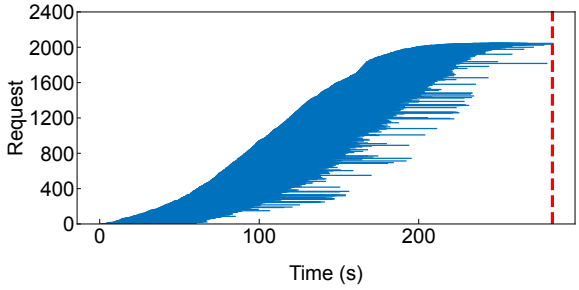


Figure 7: *Request-in, batch-out* nature of reward service.

3.2 Key Insight

A deeper analysis of the interaction between the reward service and the trainer yields our key insight: the reward service interacts with the trainer in a *request-in, batch-out* manner. Figure 7 illustrates this property. Each request is represented by a horizontal blue line, where the left endpoint marks its arrival time at the service. Requests arrive at different times, reflecting the *request-in* aspect, since during rollout the LLM generates outputs autoregressively, leading to staggered completion times for each input. For each request, we measure its compilation time and execution time to obtain its completion time under the assumption of no queuing (i.e., the earliest possible completion time). This time is indicated by the right endpoint of the corresponding line. We then take the maximum of these earliest completion times across all requests, which represents the earliest possible completion time for the entire batch, shown as the red vertical line. The *batch-out* aspect arises because the trainer consumes reward results in batches. Consequently, the trainer can only proceed

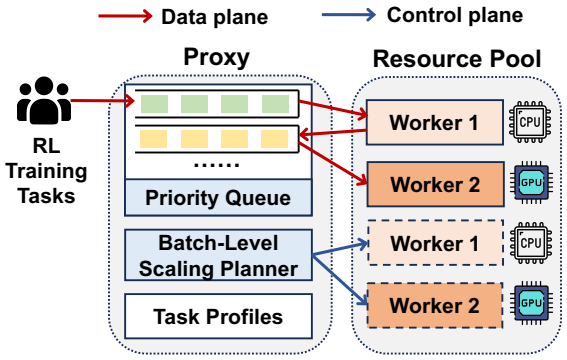


Figure 8: System overview of DistRS.

to the training phase once the reward service has finished computing rewards for the entire batch, i.e., after the red line.

The key insight from the request-in, batch-out characteristic is that each batch has a *critical request*—the request that finishes last under the assumption of no queuing. The completion time of this request determines the completion time of the entire batch. To avoid delaying training progress, the critical request must be completed without queuing. In contrast, other requests only need to finish before the critical request; as long as this condition is met, they impose no additional impact on training progress. Even though prior work such as StreamRL [15] enables streaming data transfer between inference and training to reduce synchronization granularity, training still requires at least one mini-batch to proceed. Moreover, reinforcement learning algorithms such as GRPO [22] require all samples within a group to complete reward computation before training can begin. Consequently, the *request-in, batch-out* characteristic fundamentally persists. This insight provides greater *flexibility* in scheduling request execution (Figure 2(d)) and motivates the design of more resource-efficient scaling and scheduling policies that operate under batch-level constraints.

4 Overview

Inspired by the above insight, we propose DistRS, a disaggregated reward service framework with elasticity and multi-tenancy to provide efficient reward computation for RLVR workloads. Any RL training task requiring the reward computation can connect to DistRS as a client. DistRS aims to minimize resource consumption while ensuring the progress of all training tasks. It fully leverages the *request-in, batch-out* characteristic of reward computation during training to define the Service Level Objective (SLO) at the batch level, which relaxes the latency requirement at the request level. Based on the batch-level constraint, DistRS designs efficient resource scaling and request scheduling policies. As illustrated in Figure 8, DistRS mainly consists of two modules: the proxy and the resource pool.

Elastic resource pool. The resource pool provides the computational resources required for reward computation in a microservice manner. It hosts workers that execute the different stages of reward computation, and workers of different stages may use different types of resources. Each worker can process one request at a time. We denote the worker executing the i-th stage as worker i. In the scenario of RL training for CUDA code generation, reward computation involves two stages: compiling the generated code and executing the compiled code. Therefore, the resource pool contains two types of workers: worker 1 with CPUs for compilation and worker 2 with GPUs for execution. This architecture can be easily extended to support more complex reward computations involving additional stages and resource types. The resource pool supports dynamic scaling of workers for each stage.

Proxy. The proxy serves as the entry point for training tasks to access the reward service. The proxy maintains a request queue for each stage. Requests from the training tasks or the previous stage are stored in the queue, waiting for the corresponding stage workers to fetch and process them. Each request in the queue is assigned a batch-level priority, reflecting the urgency determined by the estimated completion time of the batch it belongs to. Requests with the same priority, i.e., from the same batch, are processed in a first-come, first-served (FCFS) manner. In addition, the proxy incorporates a batch-level resource scaling planner, which periodically makes scaling decisions based on the historical request profiles it maintains. Specifically, when a new batch arrives or an existing batch completes, it estimates each batch's completion time as well as the expected arrival and running times of future requests, and then makes scaling decisions with the goal of minimizing resource consumption while preserving the training process. Moreover, the trainer can provide additional hints: the start of a new rollout batch indicates that a new batch of reward computation requests will arrive shortly. This enables us to proactively trigger resource scaling decisions and warm up new workers during the interval between the onset of rollout and the arrival of the first reward request, thereby effectively hiding the overhead of resource scheduling and cold starts.

5 Policy Design

In this section, we present the design of DistRS, with a focus on the policies for resource scaling and request scheduling. We begin by modeling the scaling objective in the single-task setting and introducing our scaling policy in Section 5.1. We then extend the design to the multi-task setting and describe the request scheduling strategies required to enable efficient multi-tenancy in Section 5.2.

5.1 Scaling Policy for Single Task

Objective and constraint formulation. We begin with the simpler case in which only a single training task utilizes the reward service. We model the resource scaling problem at the granularity of batches, meaning that resource allocation decisions are made only when the first requests of a new batch

Symbol	Description
B	Batch size
M	Number of stages in reward computation
Rt	Total resource consumption of trainer
$\mathbf{C_j}$	Cost of a worker at stage j
$\underline{r_i}$	Rollout completion time of request i
$\underline{s_{ij}}$	Running time of reward request i at stage j
$\underline{t_i}$	Earliest possible completion time of request i
$\underline{T}$	Earliest possible completion time of the batch
d	Extra delay
N_j	Number of workers at stage j
Rr	Total resource consumption of reward service

Table 1: Boldface variables denote inputs with known values; underlined variables denote inputs that are unpredictable in practice; all other variables represent either outputs determined by the policy or variables affected by these outputs.

arrive. Once decided, the allocated resources remain fixed until all requests in the batch complete, after which the resources are released. Table 1 summarizes the key notations used in this formulation. Our scaling decision determines the number of workers N_j allocated to each stage. Given the rollout completion time r_i for each request and the execution time s_{ij} of each stage in the corresponding reward computation, the earliest possible completion time of request i is

$$t_i = r_i + \sum_j s_{ij}. \tag{1}$$

Therefore, the earliest completion time of the entire batch is

$$T = \max_i t_i. \tag{2}$$

We denote the actual completion time of the entire batch as $T + d$ ($d \geq 0$), where d represents the additional delay introduced by queuing in reward computation beyond the theoretical earliest completion time T. For each batch, the objective function can be formulated as:

$$\min_{N_1,\ldots,N_M} (Rt + Rr) \cdot (T + d), \quad \text{s.t. } d \ll T. \tag{3}$$

This objective function aims to minimize the total resources consumed by the trainer and the reward service during rollout and reward computation, while ensuring that the impact of queuing in reward computation is negligible. Here, Rt denotes the trainer-side configuration, and T depends solely on the characteristics of rollout (r_i) and reward computation requests (s_{ij}), both of which are independent of the reward service. Combined with the constraint $d \ll T$, the objective can be simplified to:

$$\min_{N_1,\ldots,N_M} Rr, \quad \text{s.t. } d \ll T. \tag{4}$$

Expanding Rr as $\sum_j N_j \cdot C_j$, we obtain the final form of the objective:

$$\min_{N_1,\ldots,N_M} \sum_j N_j \cdot C_j, \quad \text{s.t. } d \ll T. \tag{5}$$

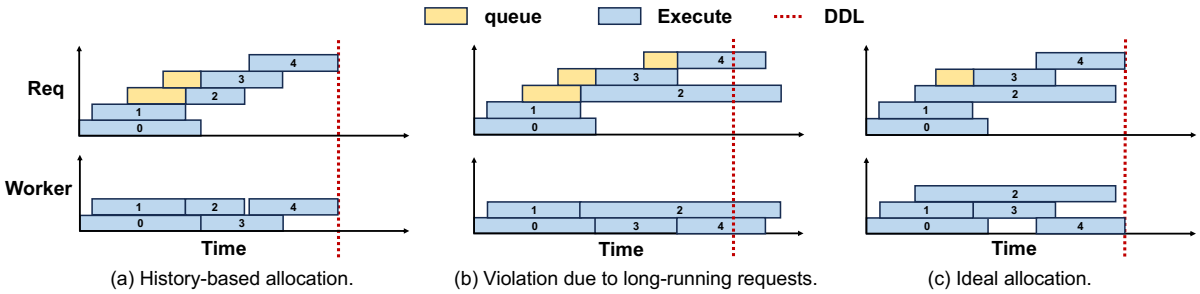
Algorithm 1 Simulation-Based Searching for Scaling

```
1:  Input: Information of the previous batch, i.e.,
2:        R = {r_i} and S = {s_ij}, where 0 ≤ i < B and 0 ≤ j < M.
3:        Cost of each worker at each stage C_1, C_2, ..., C_M
4:        the maximum tolerable extra delay D
5:  Output: The number of workers at each stage N_1, N_2, ..., N_M
6:  Initialize output list Num[M] = [B] * M
7:  sort stage j by C_j in descending order
8:  for each stage j in the sorted order do
9:      N_j^low = 1, N_j^high = B
10:     while N_j^low < N_j^high do
11:         Num[j] = (N_j^low + N_j^high)/2
12:         sat = SIMULATE(Num, R, S, D)
13:         if sat == True then
14:             N_j^high = Num[j]
15:         else
16:             N_j^low = Num[j] + 1
17:     Num[j] = N_j^low
18: return Num
```

In practical RL training, the rollout phase typically lasts several hundred seconds, placing T on the order of hundreds of seconds or more. Consequently, the additional delay d permitted by the constraint amounts to at most a few seconds. In summary, we formulate an objective that minimizes resource consumption under a batch-level delay constraint, which naturally aligns with the *request-in, batch-out* characteristic of reward computation.

Solving the problem. Ideally, we would express d as a function of the number of workers at each stage, N_j, to directly solve the optimization problem. Unfortunately, while d depends on the rollout completion time r_i of each request and the execution time s_{ij} at each stage—and is monotonically decreasing with respect to N_j—a general closed-form expression is not available. Moreover, for a future batch, r_i and s_{ij} cannot be predicted in advance. However, we observe that the arrival patterns of requests in adjacent batches are often similar (Figure 5(c)). Based on this observation, we adopt a simulation-based sequential binary search algorithm that leverages historical information to determine the number of workers at each stage.

Algorithm 1 presents the pseudocode of our scaling decision procedure. It takes as input the historical request information r_i and s_{ij}, the cost of each worker at each stage C_j, and the maximum tolerable delay D, and outputs the number of workers N_j for each stage. Initially, each N_j is set to a maximum value B (Line 6), ensuring that every request in the batch can be immediately served by an available worker, thereby satisfying the delay constraint. The stages are then sorted in descending order of worker cost C_j (Line 7). For each stage, a binary search between 1 and B determines the minimal number of workers that still satisfies the delay constraint (Lines 9-17). At each step, the current allocation is evaluated using



(a) History-based allocation. (b) Violation due to long-running requests. (c) Ideal allocation.

Figure 9: Constraint violation due to long-running requests.

the simulation function SIMULATE(Num, R, S, D), which returns true if the batch's extra delay d remains below D. In the simulation, requests are processed in an FCFS manner, with each worker handling one request at a time. The algorithm finally outputs a worker allocation that satisfies the batch-level constraint. While this approach cannot theoretically guarantee the global optimum, experiments in §6 show that it achieves satisfactory results on the CUDA reward service with a short runtime, demonstrating its practical effectiveness.

At this stage, our algorithm can find a near-optimal solution under the assumption that future batches closely resemble historical ones. However, in most cases, historical information is reliable only at the level of aggregate distributions, such as the overall arrival pattern, while the execution time of individual requests is difficult to predict accurately. Figure 9 illustrates such an example. For simplicity, we consider a reward computation scenario with a single stage and five requests per batch, with a delay requirement of $D = 0$. In Figure 9(a), the running time of each request is derived from the data of the previous batch. Algorithm 1 directly uses historical information to determine that two workers suffice to meet the constraint. Requests 2 and 3 are queued, starting execution after requests 0 and 1 complete. However, as shown in Figure 9(b), request 2 in this batch has a longer-than-expected execution time, and with only two workers, queuing causes its completion time to violate the constraint. To satisfy the requirement, three workers are needed, as illustrated in Figure 9(c), where request 2 runs without queuing. As shown in Figure 6(c), in the CUDA reward service and similar code-execution services, unexpectedly long execution times are common. To bound the runtime of each request, such services typically set a timeout threshold. When long-running requests occur, particularly those that only terminate upon reaching the timeout, the amount of queuing they can tolerate is far less than estimated. Critically, it is impossible to predict in advance which requests will be long-running. These cases can only be detected once the request starts execution, at which point no corrective action can be taken. Furthermore, long-running requests exhibit little correlation with any observable features or system metrics prior to execution. As a result, general-purpose autoscalers (e.g., utilization-driven approaches) are unaware of the batch-level constraint and therefore cannot effectively address this issue. Consequently, purely reactive approaches are insufficient, and a proactive strategy is required.

Based on the above analysis, we propose a more conservative, timeout-aware, proactive adjustment scheme. Within the *SIMULATE* function, we not only check whether the overall batch completion time satisfies the constraint, but also introduce an additional constraint to control whether certain requests can be queued. During the *SIMULATE* process, if a request i is found to be queued at stage k, we compute its latest possible completion time as

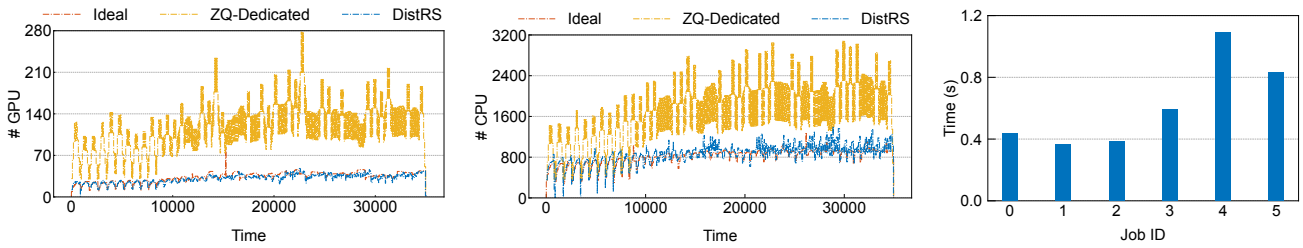
$$LT = ts + \sum_{j=k}^{M} T_j.$$

Where ts is the current timestamp and T_j is the timeout threshold for each subsequent stage. If $LT > T + D$, then this request may cause the overall batch to violate the extra delay constraint due to queuing. In this case, function *SIMULATE* immediately returns False, indicating that the current number of workers is insufficient. The timeout-aware constraint effectively prevents potentially long-running requests from being queued, resulting in a slightly higher allocation of workers. Conceptually, this trades additional resources for reduced extra delay. As demonstrated in §6, the extra resource cost is modest, while the reduction in delay is substantial.

5.2 Scaling and Scheduling Policy for Multiple Tasks

We next consider the more complex scenario in which multiple training tasks share a single reward service, with each task subject to the batch-level constraint described in §5.1.

Scaling policy. Regarding the scaling policy, the key difference between multi-tenant and single-tenant scenarios lies in the independence of batches. In the single-tenant case, each batch can be treated independently. In contrast, in the multi-tenant scenario, requests from different tasks are interleaved. Consequently, when a batch from one task arrives or completes, it is highly likely that batches from other tasks are still in progress. When continuing to make scaling decisions at the start or end of a batch, we categorize requests into three types: $Future$, requests that are expected to arrive at some point in the future; $Queuing_k$, requests that have already arrived at the service but are waiting in the queue at stage



(a) Number of GPU nodes. (b) Number of CPU nodes. (c) Extra delay.

Figure 10: Performance under colocated RL training.

Algorithm 2 Request Information Collection

1: **Input:** Information of the previous batch, i.e.,
2: $R = \{r_i\}$ and $S = \{s_{ij}\}$, where $0 \le i < B$ and $0 \le j < M$.
3: Current batch's arrival time AT
4: Elapsed time of running requests at each stage e_{ik}
5: **Output:** Estimated request information
6: Initialize output list $Reqs = []$
7: // Collect $Future$ requests
8: **for** each request i in the batch **do**
9: **if** $AT + r_i > cur\_ts$ **then**
10: $Reqs.append((AT + r_i, \{s_{i1}, s_{i2}, ..., s_{iM}\}))$
11: **for** each stage k **do**
12: // Collect $Queuing_k$ requests
13: **for** each queuing request i at stage k **do**
14: Sample a request i' from historical data
15: $Reqs.append((cur\_ts, \{s_{i'k}, s_{i'(k+1)}, \ldots, s_{i'M}\}))$
16: // Collect $Running_k$ requests
17: **for** each running request i at stage k **do**
18: Sample a request i' with $s_{i'k} > e_{ik}$ from historical data
19: $Reqs.append((cur\_ts, \{s_{i'k} - e_{ik}, s_{i'(k+1)}, \ldots, s_{i'M}\}))$
20: **return** $Reqs$

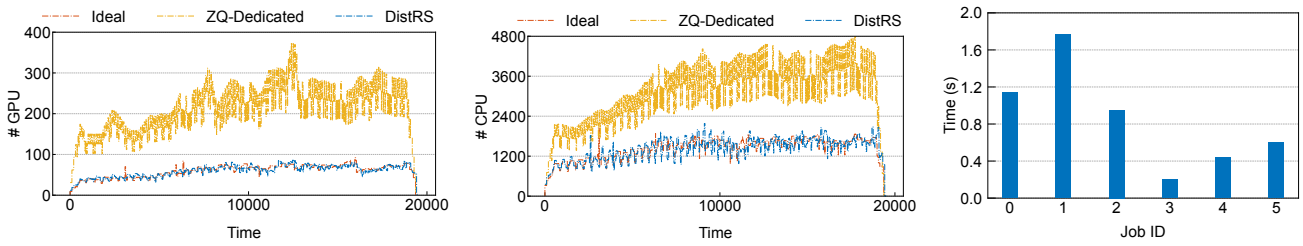
k; and $Running_k$, requests that are currently being executed at stage k. Algorithm 2 describes how requests from each currently active batch are selected or sampled and how their associated information is determined. For $Future$ requests, the only requirement is that their estimated arrival time occurs after the current timestamp. Historical requests satisfying this condition are selected, and their corresponding information can be directly used (Line 8-10). For $Queuing_k$ and $Running_k$ requests, since they have already arrived at the service, their arrival time is set to the current timestamp. Their running time at each stage is sampled from historical data. For $Running_k$ requests specifically, both the sampling and the estimation of remaining execution time must account for the time already spent executing in the current stage. Whenever a batch arrives or completes, we collect information for all active batches using Algorithm 2 and feed it into Algorithm 1 to determine the resource allocation for the next time window.

Scheduling policy. In the single-task scenario, since our constraint is defined at the batch level, all requests share the same constraint. When the execution times of individual requests are unknown, an FCFS scheduling policy is already sufficiently effective. In contrast, in a multi-task scenario, the expected earliest completion times of the current batches vary across tasks, so requests originating from different batches are subject to different constraints. However, FCFS cannot account for this difference. As a result, under the same amount of resources, requests with tighter constraints may be blocked by earlier-arriving requests with looser constraints, leading to increased queuing delays for the latter. To this end, we propose a simple yet effective scheduling policy called Earliest Batch First (EBF). Specifically, we manage a priority queue for each stage, where requests are prioritized based on the expected completion time of their corresponding batches. Requests with the same priority are sorted by their arrival time. When a worker at stage k becomes available, it selects from the queue the request whose associated batch has the earliest estimated completion time. The expected completion time of a batch is estimated using Equation 1 and 2. It is worth noting that we also apply EBF within the *SIMULATE* function, enabling the scaling policy to be aware of the scheduling policy and thereby make more informed scaling decisions.

6 Evaluation

In this section, we evaluate DistRS using the CUDA reward service illustrated in Figure 4(a), focusing on both its resource allocation efficiency and its impact on training progress. We first present an end-to-end evaluation of DistRS in a multi-tenant setting (§6.2), and then highlight the effectiveness of its scaling and scheduling policies (§6.3).

Implementation. The implementation of DistRS involves modifications to the trainer and the development of a disaggregated reward service. The trainer is a production-scale RL training framework. We trigger real-time reward computation requests via a hook attached to the end of the rollout phase. Specifically, when an input sequence completes its rollout, a function annotated with *@ray.remote* is invoked to extract the generated code and send a request to the reward service. This design allows requests to be issued immediately without interfering with ongoing rollouts. In the reward service, each worker runs in a container and is managed centrally by Kubernetes. The proxy, implemented as an HTTP server, handles incoming reward computation requests from trainers and



(a) Number of GPU nodes. (b) Number of CPU nodes. (c) Extra delay.

Figure 11: Performance under disaggregated RL training with one-step staleness.

fetch requests from workers. The scaling planner executes in a separate process, allowing it to run concurrently with the HTTP server.

6.1 Methodology

Testbed. We deploy the CUDA reward service on a cluster comprising both CPU and GPU nodes. Each compiler is allocated 4 CPU cores and 16 GB of memory, while each executor is equipped with an NVIDIA GPU for executing CUDA kernels. The proxy runs on a CPU node provisioned with 80 cores and 900 GB of memory.

Workload. We evaluate DistRS using a trace collected from a real CUDA code generation RL training task, which is the same trace analyzed in §3. For each batch, we record the arrival time of each request along with the corresponding generated code. For end-to-end evaluation, we concurrently run six RL training tasks using this trace, with their start times staggered within a short time window. The original trace corresponds to a colocated training architecture. To generate the arrival pattern for a disaggregated one-step staleness training architecture, we adjust the idle gaps between adjacent batches, reflecting the continuous rollout performed by dedicated resources in a disaggregated setup.

Baseline. Since no existing baseline is specifically designed for the reward service, and a fully static resource configuration cannot guarantee that training tasks remain unblocked, we adopt a baseline resource allocation strategy based on historical information. This strategy targets zero request queuing and allocates resources independently for each training task. We refer to this baseline as *ZQ-Dedicated* (Zero-Queue and Dedicated Resource). Additionally, we consider an idealized configuration in which all training tasks share a single reward service. In this setup, the service makes scaling decisions with perfect knowledge of the arrival times for all requests in the current batch and the execution time of each stage, while scheduling requests using EBF.

Metrics. Following the objective in Equation 5, we evaluate the performance of DistRS from two perspectives: resource allocation and its impact on training progress. For resource allocation, we record the number of GPUs and CPUs allocated by each scaling decision, visualize the allocation over time,

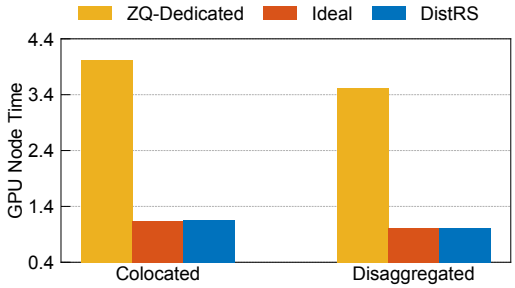


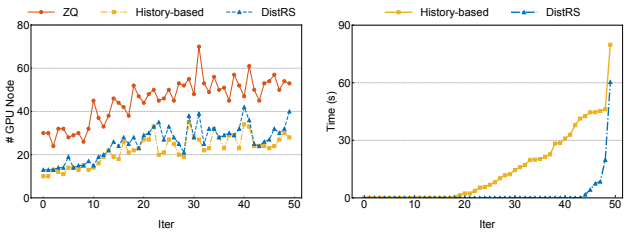
Figure 12: Normalized GPU time.

and measure the total GPU/CPU time consumed during the entire training process. To assess the impact on training, we primarily measure the extra delay of each batch (denoted as d in Table 1), which captures the additional time caused by queuing in reward computation beyond the theoretical minimum required to complete all requests in the batch.

6.2 End-to-end Performance

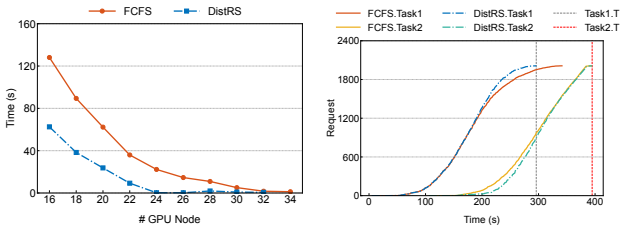
We first evaluate the end-to-end performance of DistRS with six concurrent RL training tasks under both colocated and disaggregated RL training architectures.

Colocated RL training. Figure 10 presents the performance of DistRS under colocated RL training. As shown in Figures 10(a) and 10(b), DistRS significantly reduces resource consumption compared to the ZQ-Dedicated baseline, lowering total GPU time by 3.79× and CPU time by 1.98×. This improvement stems from leveraging the *request-in, batch-out* property to flexibly queue requests, enabling all requests to be completed under the batch-level constraint using fewer resources. Additionally, DistRS supports multi-tenancy and employs EBF to prioritize requests from batches with more strict constraints, further enhancing resource utilization. In contrast, the ZQ-Dedicated baseline must allocate sufficient resources for each individual training task to avoid queuing, resulting in substantial resource waste. Moreover, DistRS's resource allocation closely tracks the ideal configuration, demonstrating its effectiveness in adapting to dynamic workloads without prior knowledge of future requests. Figure 10(c) shows that DistRS incurs negligible additional latency, averaging 0.62 seconds, which is insignificant compared to the over 600 seconds required for a full training iteration. Overall, DistRS substan-



(a) Allocated GPU nodes. (b) Extra delay.

Figure 13: Effectiveness of resource scaling.



(a) Allocated GPU nodes. (b) Extra delay.

Figure 14: Effectiveness of request scheduling.

tially reduces resource consumption while having minimal impact on training progress.

Disaggregated RL training. Figure 11 presents the performance of DistRS under disaggregated RL training with one-step staleness. With reduced idle time between batches, the workload becomes more continuous, leading to higher overall resource demand. Nevertheless, similar to the colocated training scenario, DistRS effectively reduces resource consumption compared to the ZQ-Dedicated baseline, achieving 3.49× reduction in total GPU time and 2.16× reduction in total CPU time, as shown in Figures 11(a) and 11(b). DistRS's resource allocation closely follows the ideal configuration, demonstrating its ability to adapt to different training architectures while maintaining high resource efficiency. The extra delay introduced remains low, averaging 0.85 seconds (Figure 11(c)), indicating minimal impact on training progress even under a more continuous workload.

Figure 12 compares the normalized total GPU time consumed by DistRS against the ZQ-Dedicated and ideal configurations under both colocated and disaggregated training architectures. The normalization is based on the total GPU time actually required to execute all requests, so values closer to 1 indicate lower resource waste. The results show that the values achieved by DistRS are very close to 1, indicating that it attains near-ideal resource efficiency in both training architectures, with only a minor increase in resource consumption.

6.3 Design Choices

We further evaluate the design choices of DistRS, focusing on the effectiveness of the resource scaling policy and the request scheduling policy. For brevity, we focus on GPU resource allocation, since CPU allocation follows similar trends.

Effectiveness of resource scaling. We first evaluate the effectiveness of the resource scaling policy in DistRS using a single RL training task. Since the scaling policy operates at the granularity of batches, we use the iteration ID as the x-axis in the figures. We compare DistRS with two variants: (1) *ZQ*, which allocates resources with the objective of zero queuing; and (2) *History-Based*, which allocates resources purely based on historical workload patterns. Figure 13(a) shows the number of GPU nodes allocated over time for each method. The ZQ approach consistently allocates a large number of GPU nodes, resulting in substantial resource waste. The history-based method allocates the fewest resources, as it relies solely on historical information and cannot anticipate long-running requests in new batches. In contrast, DistRS incorporates a timeout-aware adjustment strategy under the batch-level constraint, which allocates additional resources to handle potential long-running requests. Consequently, DistRS provisions slightly more resources than the history-based approach while significantly reducing the impact on training. Figure 13(b) presents the extra delay incurred by each method, where the values are sorted by extra delay for clarity. The ZQ method is not plotted, as it always incurs zero delay. The history-based method experiences a high average extra delay of 14.3 seconds due to its inability to handle long-running requests effectively. In contrast, DistRS maintains a low average extra delay of only 2.1 seconds, demonstrating the effectiveness of the timeout-aware mechanism in balancing resource efficiency and training performance.

Effectiveness of request scheduling. We then evaluate the effectiveness of DistRS's request scheduling policy using two concurrent batches, with the second batch starting 100 seconds after the first. We compare DistRS, which employs the EBF scheduling policy, with a variant using the FCFS policy. To isolate the impact of scheduling, we fix the number of CPU nodes at a large value to ensure CPU resources are not a bottleneck and vary the number of GPU nodes. Figure 14(a) shows the average extra delay incurred by the two tasks under different numbers of GPU nodes. With a small number of GPU nodes, both methods experience high extra delays due to queuing. As the number of GPU nodes increases, the extra delay for DistRS decreases rapidly, while the FCFS method remains relatively high. Using EBF, DistRS requires 1.5× fewer GPU nodes than FCFS to achieve the same low extra delay of just a few seconds. This improvement arises because EBF prioritizes requests from the first batch, which has a more strict constraint, enabling more requests from this batch to complete sooner and thereby reducing overall extra delay. In contrast, FCFS processes requests strictly in arrival order, which can delay the first batch when resources are limited. Figure 14(b) provides a detailed view of the finish times of requests from both batches when 24 GPUs are allocated. It shows that DistRS completes requests from the first batch earlier than FCFS, allowing the entire batch to finish sooner and further reducing the extra delay.

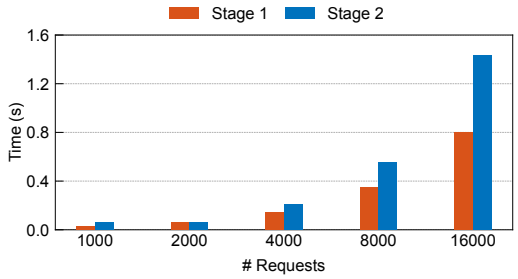


Figure 15: Algorithm running time.

6.4 System Overhead

Figure 15 shows the running time of the scaling algorithm in DistRS under varying numbers of estimated requests. Algorithm 1 determines the resource allocation for each stage sequentially. The reward computation for CUDA code generation consists of two stages: compilation (Stage 1) and execution (Stage 2), so the algorithm's running time can be divided accordingly. As shown in the figure, both stages require more time as the number of requests increases. However, even with 16,000 requests, the total running time of the algorithm remains around 2 seconds. Since the algorithm executes only at the start or completion of each batch, and batch durations are typically on the order of hundreds of seconds or more, this overhead is negligible. Furthermore, the scaling planner runs asynchronously in a separate process, ensuring it does not interfere with the normal operation of the reward service.

7 Related Work

System optimization for LLM RL. As research increasingly focuses on improving LLM performance through reinforcement learning, a growing body of work [9, 10, 14–16, 33–39] has explored system-level optimizations for LLM RL frameworks. The disaggregated architecture is first introduced in OpenRLHF [10] and NeMo-Aligner [38], which decouples the rollout and training phases to run on separate resources. In contrast, systems such as verl [9], ReaL [34], and RLHFuse [14] propose colocated architectures, colocating rollout and training on the same resources to mitigate idle periods in disaggregated designs. More recently, variants of disaggregated architectures with pipeline, streaming, or asynchronous execution have been proposed [15, 16, 35, 39]. However, these efforts focus primarily on optimizing the rollout and training phases, while the reward computation phase has received little attention. Our work is orthogonal to these RL framework optimizations and focuses on system optimizations for the reward computation phase in LLM RL training. We characterize the unique workload properties of reward computation and design a disaggregated reward service with elasticity and multi-tenancy, thereby improving the resource efficiency while minimizing its impact on RL training progress.

Resource autoscaling. Autoscaling is a widely adopted technique in cloud computing, enabling dynamic resource allocation in response to workload variations to achieve both efficient resource utilization and performance guarantees [40–45] Approaches can be broadly categorized into reactive, proactive [42, 46], and hybrid approaches. Recent work has also explored autoscaling in LLM serving systems, such as BlitzScale [47] and LambdaScale [48], to accommodate dynamic workloads. Compared to general cloud applications, reward computation workloads in RL training exhibit stronger periodicity, making workload prediction less of a challenge for autoscaling. Instead, the key difficulty lies in defining appropriate autoscaling objectives tailored to the request-in, batch-out nature of reward computation.

SLO-targeted system. Service-level objectives (SLOs) are widely employed in cloud systems to define performance targets. Prior work [17, 18, 49, 50] has focused on satisfying SLOs while improving resource efficiency or reducing cost in microservice and serverless environments. In the context of model serving, systems such as ClockWork [51], SHEPHERD [52], and DistServe [53] optimize scheduling and resource allocation to meet latency SLOs and maximize goodput. Our analysis in §3 reveals that reward computation in RLVR training exhibits a unique request-in, batch-out characteristic, meaning that the effective SLO of each individual request is determined by the batch to which it belongs. Guided by this insight, we design a reward service that improves resource efficiency while ensuring seamless integration of the reward computation phase with the RLVR training pipeline.

8 Conclusion

We present DistRS, a disaggregated reward service framework designed to provide efficient reward computation for RLVR training tasks. By leveraging the *request-in, batch-out* characteristic of reward computation, DistRS relaxes the latency requirements at the request level to a batch-level constraint. Based on this, we design resource scaling and request scheduling policies that minimize resource consumption while minimizing the impact on training progress, thereby achieving a reduction of up to 3.79× in resource consumption. We analyze the workload and explore system optimization for the reward computation phase in LLM RL training, and we believe this work will prompt a revisit of the importance of cloud infrastructure beyond training frameworks in advancing the frontier of AI capabilities.

Acknowledgements. We thank our shepherd, Rachee Singh, and the anonymous reviewers for their valuable feedback. We thank Weiqiang Lou for implementing the basic CUDA sandbox. This work was supported in part by the National Natural Science Foundation of China under Grant 62325201 and the Scientific Research Innovation Capability Support Project for Young Faculty under Grant ZYGXQNJSKYCXNLZCXM-I1. Xin Jin and Xuanzhe Liu are the corresponding authors. Ruidong Zhu, Yinmin Zhong, Xuanzhe Liu, and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, 2022.

[2] Y. Bai, A. Jones, K. Ndousse, A. Askell, A. Chen, N. DasSarma, D. Drain, S. Fort, D. Ganguli, T. Henighan, *et al.*, "Training a helpful and harmless assistant with reinforcement learning from human feedback," *arXiv preprint arXiv:2204.05862*, 2022.

[3] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, *et al.*, "Deepseek-r1: Incentivizing reasoning capability in LLMs via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[4] J. Pan, J. Zhang, X. Wang, L. Yuan, H. Peng, and A. Suhr, "TinyZero." https://github.com/Jiayi-Pan/TinyZero, 2025.

[5] B. Jin, H. Zeng, Z. Yue, J. Yoon, S. Arik, D. Wang, H. Zamani, and J. Han, "Search-r1: Training LLMs to reason and leverage search engines with reinforcement learning," *arXiv preprint arXiv:2503.09516*, 2025.

[6] Q. Yu, Z. Zhang, R. Zhu, Y. Yuan, X. Zuo, Y. Yue, W. Dai, T. Fan, G. Liu, L. Liu, *et al.*, "Dapo: An open-source LLM reinforcement learning system at scale," *arXiv preprint arXiv:2503.14476*, 2025.

[7] Y. Zheng, J. Lu, S. Wang, Z. Feng, D. Kuang, and Y. Xiong, "EasyR1: An efficient, scalable, multi-modality RL training framework." https://github.com/hiyouga/EasyR1, 2025.

[8] X. Li, X. Sun, A. Wang, J. Li, and C. Shum, "Cuda-l1: Improving cuda optimization via contrastive reinforcement learning," *arXiv preprint arXiv:2507.14111*, 2025.

[9] G. Sheng, C. Zhang, Z. Ye, X. Wu, W. Zhang, R. Zhang, Y. Peng, H. Lin, and C. Wu, "HybridFlow: A flexible and efficient RLHF framework," in *EuroSys*, 2025.

[10] J. Hu, X. Wu, W. Shen, J. K. Liu, Z. Zhu, W. Wang, S. Jiang, H. Wang, H. Chen, B. Chen, *et al.*, "OpenRLHF: An easy-to-use, scalable and high-performance RLHF framework," *arXiv preprint arXiv:2405.11143*, 2024.

[11] T. Griggs, S. Hegde, E. Tang, S. Liu, S. Cao, D. Li, C. Ruan, P. Moritz, K. Hakhamaneshi, R. Liaw, A. Malik, M. Zaharia, J. E. Gonzalez, and I. Stoica, "Evolving SkyRL into a highly-modular RL framework," 2025. Notion Blog.

[12] P. Intellect, "PRIME-RL." https://github.com/PrimeIntellect-ai/prime-rl, 2025.

[13] Y. Cheng, J. Chen, J. Chen, L. Chen, L. Chen, W. Chen, Z. Chen, S. Geng, A. Li, B. Li, *et al.*, "Fullstack bench: Evaluating LLMs as full stack coders," *arXiv preprint arXiv:2412.00535*, 2024.

[14] Y. Zhong, Z. Zhang, B. Wu, S. Liu, Y. Chen, C. Wan, H. Hu, L. Xia, R. Ming, Y. Zhu, *et al.*, "Optimizing RLHF training for large language models with stage fusion," in *USENIX NSDI*, 2025.

[15] Y. Zhong, Z. Zhang, X. Song, H. Hu, C. Jin, B. Wu, N. Chen, Y. Chen, Y. Zhou, C. Wan, *et al.*, "StreamRL: Scalable, heterogeneous, and elastic RL for LLMs with disaggregated stream generation," *arXiv preprint arXiv:2504.15930*, 2025.

[16] W. Fu, J. Gao, X. Shen, C. Zhu, Z. Mei, C. He, S. Xu, G. Wei, J. Mei, J. Wang, *et al.*, "AReaL: A large-scale asynchronous reinforcement learning system for language reasoning," *arXiv preprint arXiv:2505.24298*, 2025.

[17] Z. Wang, P. Li, C.-J. M. Liang, F. Wu, and F. Y. Yan, "Autothrottle: A practical bi-level approach to resource management for SLO-targeted microservices," in *USENIX NSDI*, 2024.

[18] Z. Wu, W.-L. Chiang, Z. Mao, Z. Yang, E. Friedman, S. Shenker, and I. Stoica, "Can't be late: optimizing spot instance savings under deadlines," in *USENIX NSDI*, 2024.

[19] L. Weng, "Reward hacking in reinforcement learning," *lilianweng.github.io*, Nov 2024.

[20] Y. Miao, S. Zhang, L. Ding, R. Bao, L. Zhang, and D. Tao, "Inform: Mitigating reward hacking in RLHF via information-theoretic reward modeling," *Advances in Neural Information Processing Systems*, 2024.

[21] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[22] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu, *et al.*, "Deepseekmath: Pushing the limits of mathematical reasoning in open language models," *arXiv preprint arXiv:2402.03300*, 2024.

[23] C. Zheng, S. Liu, M. Li, X.-H. Chen, B. Yu, C. Gao, K. Dang, Y. Liu, R. Men, A. Yang, *et al.*, "Group sequence policy optimization," *arXiv preprint arXiv:2507.18071*, 2025.

[24] M. Luo, S. Tan, R. Huang, A. Patel, A. Ariyak, Q. Wu, X. Shi, R. Xin, C. Cai, M. Weber, *et al.*, "Deepcoder: A fully open-source 14b coder at o3-mini level," 2025. Notion Blog.

[25] C. Baronio, P. Marsella, B. Pan, S. Guo, and S. Alberti, "Kevin: Multi-turn RL for generating CUDA kernels," *arXiv preprint arXiv:2507.11948*, 2025.

[26] S. Liu, S. Hegde, S. Cao, A. Zhu, D. Li, T. Griggs, E. Tang, A. Malik, K. Hakhamaneshi, R. Liaw, P. Moritz, M. Zaharia, J. E. Gonzalez, and I. Stoica, "SkyRL-SQL: Matching GPT-4o and o4-mini on text2SQL with multi-turn RL," 2025. Notion Blog.

[27] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal, "A survey on online judge systems and their applications," *ACM Computing Surveys (CSUR)*, 2018.

[28] H. Wu, Y. Liu, L. Qiu, and Y. Liu, "Online judge system and its applications in c language teaching," in *International Symposium on Educational Technology (ISET)*, 2016.

[29] S. Yao, "The second half." https://ysymyth.github.io/The-Second-Half, 2025.

[30] Y. Yue, Y. Yuan, Q. Yu, X. Zuo, R. Zhu, W. Xu, J. Chen, C. Wang, T. Fan, Z. Du, *et al.*, "Vapo: Efficient and reliable reinforcement learning for advanced reasoning tasks," *arXiv preprint arXiv:2504.05118*, 2025.

[31] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: NIMBLE task scheduling for serverless analytics," in *USENIX NSDI*, 2021.

[32] C. Jin, Z. Zhang, X. Xiang, S. Zou, G. Huang, X. Liu, and X. Jin, "Ditto: Efficient serverless analytics with elastic parallelism," in *ACM SIGCOMM*, 2023.

[33] Z. Yao, R. Y. Aminabadi, O. Ruwase, S. Rajbhandari, X. Wu, A. A. Awan, J. Rasley, M. Zhang, C. Li, C. Holmes, *et al.*, "DeepSpeed-Chat: Easy, fast and affordable RLHF training of chatgpt-like models at all scales," *arXiv preprint arXiv:2308.01320*, 2023.

[34] Z. Mei, W. Fu, K. Li, G. Wang, H. Zhang, and Y. Wu, "Real: Efficient RLHF training of large language models with parameter reallocation," *arXiv preprint arXiv:2406.14088*, 2024.

[35] J. He, T. Li, E. Feng, D. Du, Q. Liu, T. Liu, Y. Xia, and H. Chen, "History Rhymes: Accelerating LLM reinforcement learning with RhymeRL," *arXiv preprint arXiv:2508.18588*, 2025.

[36] Z. Wang, T. Zhou, L. Liu, A. Li, J. Hu, D. Yang, J. Hou, S. Feng, Y. Cheng, and Y. Qi, "DistFlow: A fully distributed RL framework for scalable and efficient LLM post-training," *arXiv preprint arXiv:2507.13833*, 2025.

[37] K. Lei, Y. Jin, M. Zhai, K. Huang, H. Ye, and J. Zhai, "PUZZLE: Efficiently aligning large language models through Light-Weight context switch," in *USENIX ATC*, 2024.

[38] G. Shen, Z. Wang, O. Delalleau, J. Zeng, Y. Dong, D. Egert, S. Sun, J. Zhang, S. Jain, A. Taghibakhshi, *et al.*, "Nemo-aligner: Scalable toolkit for efficient model alignment," *arXiv preprint arXiv:2405.01481*, 2024.

[39] Z. Han, A. You, H. Wang, K. Luo, G. Yang, W. Shi, M. Chen, S. Zhang, Z. Lan, C. Deng, *et al.*, "AsyncFlow: An asynchronous streaming RL framework for efficient LLM post-training," *arXiv preprint arXiv:2507.01663*, 2025.

[40] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, *et al.*, "Autopilot: workload autoscaling at google," in *EuroSys*, 2020.

[41] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *IEEE 4th international conference on cloud computing*, 2011.

[42] Z. Pan, Y. Wang, Y. Zhang, S. B. Yang, Y. Cheng, P. Chen, C. Guo, Q. Wen, X. Tian, Y. Dou, *et al.*, "Magicscaler: Uncertainty-aware, predictive autoscaling," *VLDB*, 2023.

[43] H. Qian, Q. Wen, L. Sun, J. Gu, Q. Niu, and Z. Tang, "Robustscaler: Qos-aware autoscaling for complex workloads," in *IEEE ICDE*, 2022.

[44] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *IEEE ICDCS*, 2019.

[45] D. Zou, W. Lu, Z. Zhu, X. Lu, J. Zhou, X. Wang, K. Liu, K. Wang, R. Sun, and H. Wang, "Optscaler: A collaborative framework for robust autoscaling in the cloud," *VLDB*, 2024.

[46] Z. Zhou, C. Zhang, L. Ma, J. Gu, H. Qian, Q. Wen, L. Sun, P. Li, and Z. Tang, "AHPA: adaptive horizontal pod autoscaling systems on alibaba cloud container service for kubernetes," in *Proceedings of the AAAI conference on artificial intelligence*, 2023.

[47] D. Zhang, H. Wang, Y. Liu, X. Wei, Y. Shan, R. Chen, and H. Chen, "BlitzScale: Fast and live large model autoscaling with O (1) host caching," in *USENIX OSDI*, 2025.

[48] M. Yu, R. Yang, C. Jia, Z. Su, S. Yao, T. Lan, Y. Yang, Y. Cheng, W. Wang, A. Wang, *et al.*, "λ Scale: Enabling fast scaling for serverless large language model inference," *arXiv preprint arXiv:2502.09922*, 2025.

[49] Z. Zhang, C. Jin, and X. Jin, "Jolteon: Unleashing the promise of serverless for serverless workflows," in *USENIX NSDI*, 2024.

[50] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs," in *USENIX OSDI*, 2022.

[51] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving DNNs like clockwork: Performance predictability from the bottom up," in *USENIX OSDI*, 2020.

[52] H. Zhang, Y. Tang, A. Khandelwal, and I. Stoica, "SHEPHERD: Serving DNNs in the wild," in *USENIX NSDI*, 2023.

[53] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving," in *USENIX OSDI*, 2024.