USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

RobustRL: Role-based Fault Tolerance System for RL Post-Training

Zhenqian Chen and Baoquan Zhong, *Zhejiang University;* Xiang Li, *unaffiliated;* Qing Dai, Xinkui Zhao, and Miao Ye, *Zhejiang University;* Ren Cheng, *unaffiliated;* Lufei Zhang, *State Key Laboratory of Mathematical Engineering and Advanced Computing, China;* Jianwei Yin, *Zhejiang University*

https://www.usenix.org/conference/osdi26/presentation/chen-zhenqian

This paper is included in the Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation.

July 13–15, 2026 • Seattle, WA, USA

ISBN 978-1-939133-55-7

Open access to the Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

RobustRL: Role-based Fault Tolerance System for RL Post-Training

Zhenqian Chen[1], Baoquan Zhong[1], Xiang Li[2], Qing Dai[1], Xinkui Zhao[1†],
Miao Ye[1], Ren Cheng[2], Lufei Zhang[3], Jianwei Yin[1]
[1]Zhejiang University [2]Unaffiliated
[3]State Key Laboratory of Mathematical Engineering and Advanced Computing, China

Abstract

RL post-training for LLMs has been widely scaled to enhance reasoning and tool-using capabilities. However, RL post-training interleaves training and inference workloads, exposing the system to faults from both sides. Existing fault tolerance frameworks for LLMs target either training or inference, leaving the optimization potential in the asynchronous execution unexplored for RL. Our key insight is role-based fault isolation so the failure in one machine does not affect the others. We treat trainer, rollout, and other management roles in RL training as distinct distributed sub-tasks. Instead of restarting the entire task in pretrain robust system ByteRobust, we recover only the failed role and reconnect it to living ones, thereby eliminating the full-restart overhead including rollout replay and initialization delay.

We present RobustRL, the first comprehensive robust system to handle GPU machine errors for RL post-training ETTR (Effective Training Time Ratio) improvement via a *Detect-Restart-Reconnect* paradigm. (1) *Detect*. We implement role-aware monitoring to distinguish actual failures from role-specific behaviors to avoid the false positive and delayed detection. (2) *Restart*. For trainers, we implement a non-disruptive recovery where rollouts persist state and continue trajectory generation, while the trainer is rapidly restored via rollout warm standbys. For rollout, we perform isolated machine replacement without interrupting the RL task. (3) *Reconnect*. We replace static collective communication with dynamic, UCX-based (Unified Communication X) point-to-point communication, enabling immediate weight synchronization between recovered roles. In an RL training task on a 256-GPU cluster with Qwen3-8B-Math workload under 10% failure injection frequency, RobustRL can achieve an ETTR of over 80% compared with the 60% in ByteRobust and achieves 8.4%-17.4% faster in end-to-end training time.

1 Introduction

Reinforcement Learning (RL) has emerged as a transformative paradigm for post-training LLMs (Large Language Models) [2, 6, 32], where the week-level [24] and 200K GPUs large-scale [80] RL training has been deployed to enhance their reasoning and tool-using capabilities through iterative policy optimization [57,59,85]. State-of-the-art models [7,24,52,63,64,80] all leverage RL post-training to achieve superior performance in complex tasks [91], such as mathematics [50], code generation [30,42], and search [31] *et al.*.

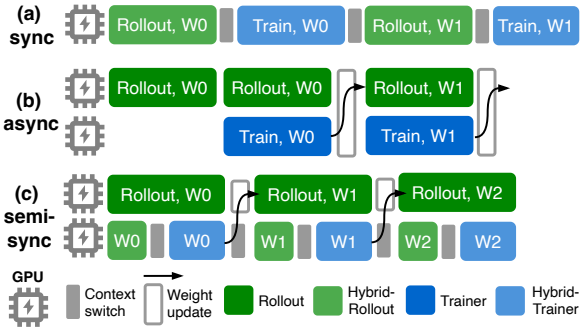

Figure 1: Different RL training architectures. The GPU role includes trainer, rollout and hybrid (colocate the both).

Unlike traditional LLM pre-training [19, 29, 73] or serving [34,62,75,81] workloads, RL post-training comprises two phases: rollout (generation and tool interaction) and training, where they contribute close to 100% and 70% post-training time in reasoning tasks and tool learning tasks [60]. To mitigate the long-tail latency inherent in the rollout phase, RL training frameworks have evolved from synchronous architectures that co-locate rollout and trainers [26, 43, 61, 84, 90] in Figure 1(a) to asynchronous designs [16, 25, 60, 77, 89, 93] in Figure 1(b) and (c). These include async mode with dedicated standalone rollouts and trainers, as well as semi-sync mode [7, 65] that combine co-located workers with additional standalone rollouts.

The efficiency of large-scale RL training is therefore critically dependent on the reliability of both the trainer and rollout because they consume the majority of training time [60]. When scaling to thousands of GPUs, machine failures become the dominant source of task interruption [11, 38, 73].

†Corresponding author.

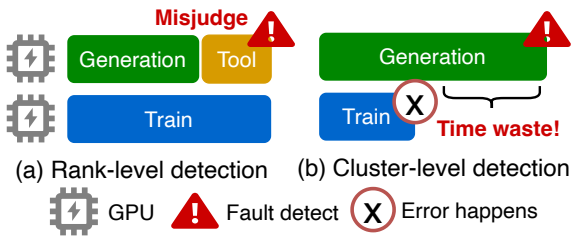

Figure 2: Fault detection of pre-train applied in RL. (a) Rank-level leads to false positive. (b) Cluster-level leads to delay.

However, because rollout and training are tightly interleaved in a single RL task, fault detection and recovery must handle failures stemming from both phases. A naive solution is applying different fault tolerance strategies to the specific roles of the failed components. Unfortunately, they lack the fine-grained mechanisms required for the unique demands of RL. Existing fault tolerance solutions are tailored to pure pre-training [4, 13, 17, 27, 36, 70, 71, 73] or pure inference workloads [39, 46, 54]. In current RL systems, the failure of a single critical worker would trigger a RL task restart, discarding training progress and wasting computation resources. Ideally, a fault in a machine hosting one distributed RL role should not disrupt the others. Only the ***detected*** failed role should ***restart*** and subsequently ***reconnect*** with others while the remaining roles continue to run. However, achieving this ideal state presents three significant challenges.

Challenge #1: Accurate and efficient fault detection. The rank-level detection for training in ByteRobust does not work for RL. It flags a process as suspect when its behavior deviates from peers or it exhibits zero GPU or network activity [11, 29, 38, 73] because each process runs identical forward-backward passes. As shown in Figure 2(a), however, rollouts can idle their GPUs while awaiting external tool responses, causing false positives. Conversely, cluster-level detects the error when all ranks have no GPU activity. It masks these idle periods and introduces significant detection delays as shown in Figure 2. For instance, a trainer network failure might remain undetected for hours until the long-tail rollout phase completes, wasting valuable cluster resources.

Challenge #2: Fine-grained and efficient restart. We need to guarantee both machine failures in trainer and rollout would not lose the trajectory due to the task restart. Furthermore, for the trainer, we need rollout to continue the generation during the recovery of the trainer. Besides, the gang-scheduling leads to the long blocking time until the new machine initializes, while the extra warm standby machine leads to the resource waste. An efficient and resource-friendly system is needed for efficient trainer restart. For rollout, we want the recovered rollout to soon get the latest weight for trajectories generation.

Challenge #3: Dynamic and efficient weight synchronization. The restart trainer or rollout needs to build reconnection for subsequent weight update as shown in Figure 1(b) and (c). Collective communication based on NCCL [1] needs to build the fixed communication group before transfer, so it cannot connect with the recovered role in the new machine and does not support fault tolerance. Point-to-point communication like UCX is available for dynamic reconnection, but needs asynchronous and parallel optimization to be efficient.

To address these challenges, we designed RobustRL, the first comprehensive role-based fault tolerance system for RL post-training in the three RL modes of Figure 1. It mitigates the impact of the machine failures for the rollout and trainer. Our contributions are as follows.

1. Role-phase-aware fault detection. It decides which roles should restart. We introduce a phase-aware and extensible detection strategy for trainer and rollout. Faults in corresponding roles are handled using appropriate robust strategies. (§4).

2. Role-based and resource-efficient restart. It decides how the roles restart. We propose a decoupled recovery strategy to avoid the RL task restart and preserve the rollout progress. In semi-sync and async mode, for trainer failures, we implement an efficient recovery that dynamically takes rollout as warm standbys, bypassing gang-scheduling delays without requiring extra idle resources. For rollout failures, the recovered rollout would pull the latest weight from other rollouts for trajectory generation immediately. (§5.1)

3. Efficient and reliable reconnection. It decides how the roles reconnect. We implement UCX-based (Unified Communication X) dynamic communication to replace NCCL for weight synchronization. It can achieve efficient weight update through asynchronous point-to-point communication and relay server design. In addition, it can handle the different failure cases during the recovery. (§5.2)

4. End-to-end benefits. In the scenario with frequent failures where a trainer fault is injected every 10% steps, RobustRL still maintains an 80% ETTR on the Qwen3-8B-Math training task with 20% higher than ByteRobust. The RL training time is 8.4%-17.4% faster than ByteRobust. (§7)

2 Background & Motivation

2.1 RL System for LLM Post-Training

Post-training using Reinforcement Learning (RL), which evolved from initial human feedback (RLHF) techniques [5, 55], is now critical for enhancing the reasoning [24, 59, 85] and tool-using capabilities [14, 30] of LLMs. This process consists of two phases, rollout and training.

Rollout. The rollout model generates trajectories based on prompts and the results of environment interaction, a process typically completed by an inference engine [35, 88]. For example, mathematical reasoning problems [50] can be solved using chain-of-thought (CoT) [76]. The rollout model can further achieve tool learning through multi-turn tool invocations. In software engineering [30] scenarios, trajectories include multi-turn interactions with a code sandbox to fix a bug or implement a specific function. In search scenarios [31, 33, 83],

trajectories involve multi-turn interactions with a search engine to retrieve target answers.

Training. The actor updates its policy by trajectory rewards evaluated in the training engine [49, 87]. The score can be obtained via various reward sources such as rule-based functions [24, 85] or learned reward models [5, 53]. Subsequently, the reference and critic models convert per-trajectory scores and advantages into training experience. Finally, the actor model consumes this experience through a specific policy update strategy [3, 57, 59, 85] to complete this iteration.

RL training system. RL frameworks like verl [22] adopt a synchronous mode. After the rollout phase is completed, they switch the inference engine to a training engine by resharding the model weights. However, the multi-turn interactions and decode latency in rollout leads to a significant long-tail latency. Therefore, existing async RL frameworks [16, 25, 60, 89, 93] allow the trainer to train within a certain range of offline steps without waiting for all rollout trajectories to complete. It greatly improves training efficiency while maintaining an acceptable loss in precision. Compared to the hybrid deployment of the sync mode, the async paradigm independently deploys the trainer and standalone rollouts, with weight synchronization through network pulling. The semi-sync [58, 65] mode replaces the async trainer with a hybrid one, thereby avoiding idle waiting caused by an improper ratio of trainer to standalone rollouts as shown in Figure 1.

2.2 LLM Fault Tolerance System

Like pre-training, the RL post-training of large models can last for several weeks or even months [80]. During this period, machine failures inevitably occur, leading to a loss of training progress. In RL, a GPU machine where a failure occurs could be executing either a training or an inference task. Prior works, ByteRobust [73] and EaaS [39], have respectively discussed machine failure scenarios during training and inference. In the pre-training process, at the scale of a 100K GPU cluster, machine-induced failures (e.g., GPU, network, CPU, memory, disk) cause over 3,000 interruptions per week on average (30/k-machine/week). For inference services, at the scale of a 4K GPU cluster, 8.8 failures also occur weekly (2.2/k-machines/week), with the main faults manifesting as GPU disconnection and ECC (Error-Correcting Code) errors.

On top of training and inference workload, RL introduces additional stages such as weight synchronization, Ray [48] management, and tool calls. It further increases the possibility of machine failures. Furthermore, by analyzing issues from open-source RL frameworks with keyword matching, we have found the similar problems caused by the machine error as shown in Table 1. Unfortunately, there is no specific robust system for the RL system as we know and the ByteRobust is applied since RL post-training is a kind of training task. Restarting the task when an error happens in ByteRobust is not the best optimization in RL.

Project	verl [61]	Roll [21]	OpenRLHF [20]	Slime [93]
Total issues	1830	127	719	162
CUDA error	189	7	74	14
Timeout	152	6	60	9
OOM	117	7	66	12
Job Hang	65	2	20	4

Table 1: Issues of machine error and job hang in open source RL training framework.

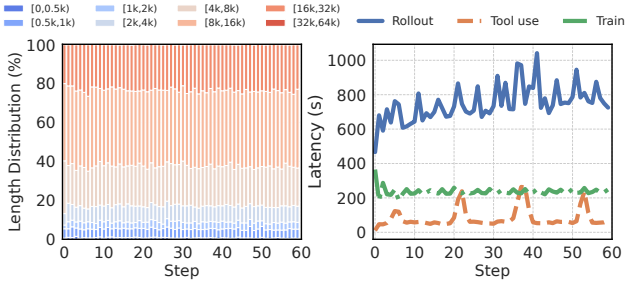

Figure 3: Trajectory length distribution (a) and time cost of each step (b) of Search R1 [31] training in HotPotQA dataset [83] with Qwen2.5-7B [67].

For instance, the pre-train robust system ByteRobust [73] achieves warm standby machines to enable fast recovery from a task restart. But it restarts the whole task whatever the error happens, leading to the large progress loss especially in RL training. A better solution is fine-gained restart by role and rollout can be the warm standby for the trainer. We can apply ByteRobust only when the role-based robust strategy in RL cannot handle the case like human-made code or configuration error. For the inference system, they mainly focus on keeping the service running [39,46] or avoid re-prefilling [18,54] when one inference machine fails. Current RL robust systems follow the inference design [60, 79] to achieve the robust rollout because it only needs to extra consider how to reconnect with the trainer in async mode. However, the trainer process has much higher failure frequency compared with inference. The robust trainer is important and the robust rollout should take the trainer failure into consideration.

2.3 Opportunity and Challenge

RL robust benefits. Role-based fault tolerance in RL has two benefits. First, it ensures the failure isolation of different roles. The task can continue to run, which guarantees that the current rollout and agent environment states are maintained. This is important under the long-tail phenomenon, where the rollout phase constitutes the main overhead. In Figure 3 of search agent training, rollout phase plays the major overhead in one step. Because the response length is long and the tool calling time contributes to the extra overhead. Second, it reduces the recovery time. We only need to restart the trainer instead of the RL task. In the case of async training, the failure of a rollout

does not affect the training task, as requests are forwarded to other rollouts. A trainer failure does not affect the rollouts from generating trajectories. To achieve so, we need to handle the following issues.

Per-step checkpoint. If weights are not saved each step, a weight inconsistency between the recovered trainer and the rollouts will occur when a failure happens. This leads to trajectory deviations and affects the training results. In this situation, it is necessary to roll back to the last checkpoint saved step to retrain and re-rollout. Per-step checkpoint can avoid the re-execution of the time consuming rollout phase.

We believe that a per-step checkpoint is available in the RL scenario based on the following insights. First, in pre-training scenarios, checkpoints are typically saved every ten or hundred steps because a single training step takes only seconds. In contrast, an RL step can last for minutes or even hours [91], which makes the checkpoint overhead ratio small. Second, in async RL training, the checkpoint process only blocks the trainer for a short time, while rollouts can continue inference during this period. Third, to further reduce the overhead brought by a per-step checkpoint, we apply the ByteCheckpoint [72]. The blocking time for saving weights is mainly the GPU to memory time within 5 seconds.

Efficient point-to-point communication. In asynchronous RL training, when trainer or rollout has faults, it needs to re-establish connections after recovery for later weight synchronization shown in Figure 1(b) and (c). The existing collective communication protocol NCCL only supports the static communication member so it cannot support robust cases. For example, the trainer would send its weight by layer to the rank-0 of the rollout machine by NCCL. Then the rollout machine boards the weight to other ranks through NVLink. When the number of communication members increases, the transfer overhead would increase linearly.

We solve this by the UCX-based point-to-point communication. It supports dynamic connection and asynchronous rank-level transmission from the trainer GPU to rollout GPU directly. It solves the traffic bottleneck of single NICs (Network Interface Cards) in NCCL. In addition, the rollout which finishes the weight update can be the relay server for the outdated or recovered rollout pulling. Finally, We further support the fault tolerance case during weight synchronization.

3 System Overview

The design goal of RobustRL is to ensure reliable RL training by using a role-based fault detection mechanism and applying different fault recovery strategies to trainer and rollout. RobustRL consists of two core components: control plane and data plane. The architecture overview is shown in Figure 4.

Control Plane. It mainly consists of the phase aware analyzer and the runtime controller. The analyzer is responsible for interacting with the RL robust runtime. It tailors fault-detection strategies to each RL role and training stage. The controller,

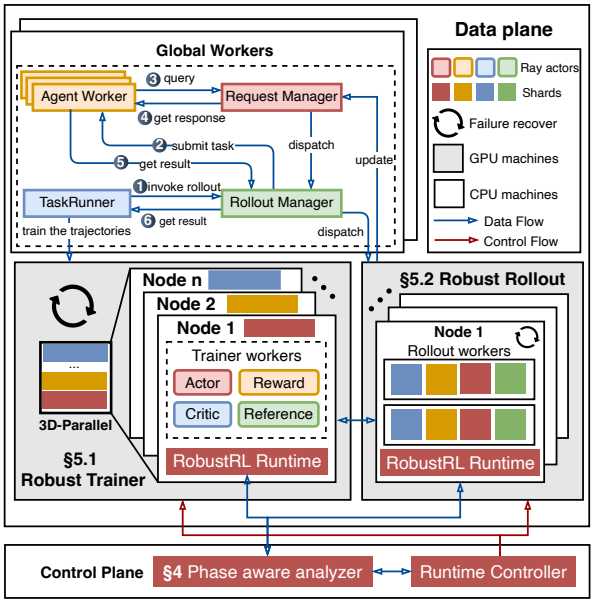

Figure 4: System overview of RobustRL.

responsible for machine scheduling, receives analysis results from the analyzer and adopts the appropriate fault tolerance and recovery strategy (§4).

Data Plane. The GPU machines are the primary failure sources and they all have fault tolerance capabilities (§5) for machine errors with RobustRL. All trainers would restart whenever any trainer machine fails, while rollouts would only restart or replace the corresponding faulty machine.

The CPU machine includes management roles, such as the `AgentWorker` for tool interaction, the `RolloutManager`, and the `RequestManager` for managing inference instances and requests. They are placed on CPU machines, and the number of machines is determined by the workload. There can be multiple `AgentWorkers`, while the other roles default to one instance each. The `TaskRunner` invokes the `RolloutManager` to get the trajectories (①). Then `RolloutManager` invokes the `AgentWorker` (②) to call the inference engine and tools to store the result in the `RequestManager` (③). This process can be a multi-turn conversation and the `AgentWorker` get the trajectories (④ and ⑤) to the trainer (⑥).

A failure in these roles triggers a RL task restart [73]. We use affinity scheduling to ensure that management roles are not scheduled on trainer or rollout machines. This eliminates RL task restarts that would otherwise be triggered by the termination of a management role when a trainer or rollout machine is replaced after a failure.

4 Role-based Fault Detection

We illustrate the semi-sync RL workflow in Figure 5 since both sync and async training are special cases of it. Our goal

is to detect which trainer or rollout machine has failed fast and apply the corresponding robust mechanism in §5.

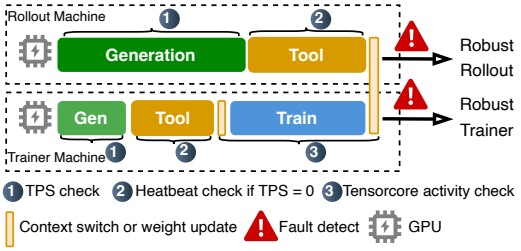

Figure 5: Role-phase-aware fault detection in semi-sync RL training. Row 2 is the hybrid. TPS: Throughput Per Second.

Trainer fault detection. As introduced in §2.1, the training phase is divided into the forward-only computation of the reference/critic/reward models and the forward-backward computation of the actor/critic policy update. The detection strategy during this process is consistent with that of pre-training. Failures manifest as zero GPU TensorCore activity for a five-minute window in RobustRL. When this occurs, we switch to the robust trainer workflow (§5.1.2).

Trainer detection is only applied during the training phase on the trainer machines (Figure 5 ③). Although phases such as context switching, weight updates, or advantage computation exhibit no GPU activity, their duration is short enough relative to the training phase. Thus, monitoring TensorCore activity is sufficient to detect failures without false positives from short GPU-idle phases. Users can also extend their own detection strategy for this case. For example in large scale RL training, the advantage computation can be over 5 minutes and we can set a large GPU idle threshold. The trainer fault detection method is orthogonal to both pre-training [10,11] and machine diagnostic methods [73].

Rollout fault detection. Rollout workloads comprise inference, weight synchronization, and awaiting requests or tool returns. TensorCore activity occurs solely during inference. In the other two stages, the worker exhibits no GPU activity yet remains healthy. We first periodically collect the throughput of each rollout from the `RolloutManager` (①). If the rollout exhibits zero throughput for a specified time interval (60 seconds in our system), we mark it as suspect and trigger heartbeat detection to further verify its status (②). No response within the timeout confirms a rollout machine failure and we will handle it with a robust rollout strategy (§5.2). The Rollout detection adopts throughput detection prior to heartbeat, avoiding misjudgment caused by unprocessed heartbeat requests and excessively long TTFT (time-to-first-token) in high-load Rollout scenarios. For the hybrid-rollout of sync and semi-sync mode, this detection method can also be applied. Since the error happens at the trainer machine, we switch to the robust trainer workflow (§5.1.2).

Extensibility to complex faults. It is worth noting that the role separation architecture proposed in RobustRL naturally facilitates the integration of advanced fault detection methods. By decoupling the execution contexts, our system becomes inherently compatible with specialized detection strategies designed for complex issues like silent data corruption (SDC) [28,40,41] and stragglers [38,78] to further harden the system.

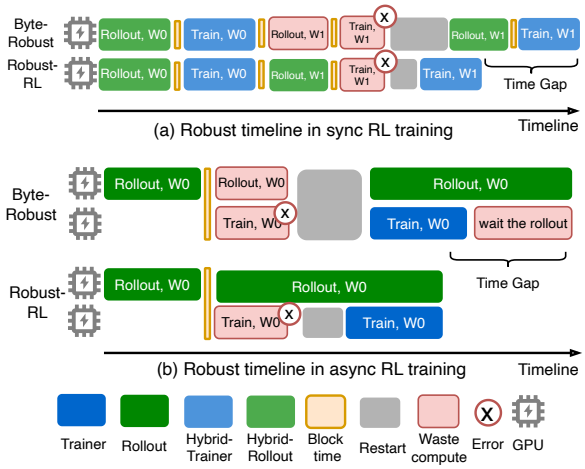

Figure 6: Timeline comparison between ByteRobust and RobustRL in (a) sync and (b) async mode when the trainer machine error happens.

5 Role-based Fault Tolerance

5.1 Robust Trainer

This section illustrates the benefit of robust trainers in different RL architectures (§5.1.1), appropriate restart strategy (§5.1.2) and reduction of restart overhead by rollout warm standbys (§5.1.3).

5.1.1 Robust Trainer Timeline

Figure 6 illustrates the difference between the ByteRobust and the RobustRL in both sync and async training mode. The semi-sync mode is the special case of sync and async mode because it contains both hybrid and standalone rollout. The yellow part denotes the time that blocks the RL training, which includes weight synchronization, per-step checkpoint and weight reshard between training and inference engine.

Robust trainer benefit. In sync training mode of Figure 6(a), ByteRobust restarts the entire task, with the rollout progress of the current iteration lost. RobustRL only restarts the affected hybrid role and preserves the rollout progress. After recovery, it simply resumes rollout and thus completes the iteration sooner than ByteRobust. In async training mode of Figure 6(b), the extra benefit is that rollout can continue running during the trainer recovery. Under the same failure case, the rollout in RobustRL ends earlier, it avoids the trainer to wait for the rollout to end in ByteRobust. In addition, compared

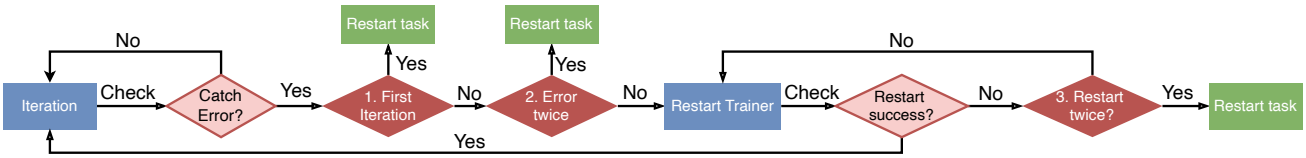

Figure 7: Robust trainer workflow. The blue part is the operation in RL training. The red part is the judge for the robust case. The green part is the policy to handle the robust case.

with restarting the RL task, we eliminate the overhead for container initialization, ray cluster initialization *et al.*.

Low blocking overhead. Due to the long-tail phenomenon of rollout, one step in RL is long shown in Figure 3(b), making the blocking time relatively short. The efficiency of rollout weight update has been solved in §5.2.1. In addition, we introduce ByteCheckpoint [72] to improve the checkpoint saving efficiency. The blocking time is only the GPU-to-memory, while the transfer from memory to disk or other persistent storage is executed asynchronously.

5.1.2 Robust Trainer Workflow

Robust trainer targets restarts from machine failures and we must avoid an infinite loop of trainer restarts by the human-induced fault. When a human-induced fault occurs, we restart the task and check the code or roll back the code version instead of trainer role restart as soon as possible. This operation "Restart task" is similar to ByteRobust's machine diagnostics and code rollback procedures. We present the robust trainer workflow in Figure 7.

Iteration. One iteration includes the step ① to ⑥ in Figure 4. The RL training is completed through the iteration of multiple steps. Robust trainer wraps the training step function with a `try-catch` block to capture exceptions during training. The sources of exceptions include both explicit and implicit failures, which are detected by the method in §4. When the exception happens it turns to the robust phase.

Trainer restart. First, `TaskRunner` terminates all trainer-related processes. Then, it re-executes the trainer's initialization phase and finally reloads the weights from the last saved model checkpoint. Since the model weights are saved each step, the reloaded weights are consistent with the rollout. Information that binds rollouts to the trainer, such as communication addresses for pulling weights, is updated concurrently with the trainer loading the weights.

When we restart to iterate, we skip to rollout a new batch. Instead, when the current batch finishes rollout, `TaskRunner` fetches the trajectories from the `RequestManager` to the trainer. Otherwise, if we use the sync or semi-sync mode for RL training, we switch the context to the inference engine to rollout the trajectories.

In the following situations, we restart the RL task instead of only the trainers shown in Figure 7.

1. First-iteration exception detected. If the model fails on the first step of training or on the first step after resuming training, this indicates an explicit machine or code error. [2]

2. Repeated exception detected. If it is the second time the fault has been detected in the current step, we indicate a reproducible fault exists in this step. This situation is also similar to a code error and we restart the task. For example, developers introduce a code error in the validation process, which is executed each 5 steps.

3. Repeated restart failure detected. The restart process itself may fail. For example, if resources are not fully released during the trainer restart or if connection establishment fails due to the restart. In such cases, one restart failure is permitted. If restart failure persists, we also consider it to be a reproducible machine or code error.

5.1.3 Trainer Warmup by Rollout

The trainer schedule follows the gang scheduling, meaning the trainer's initialization cannot begin until a new machine is scheduled and the environment is initialized, which is time-consuming as shown in Figure 8(a). ByteRobust prepares the extra warm standby machines to replace faulty ones, thereby saving scheduling and container initialization time. However, using redundant machines leads to potential resource waste, making this strategy limited to large-scale training.

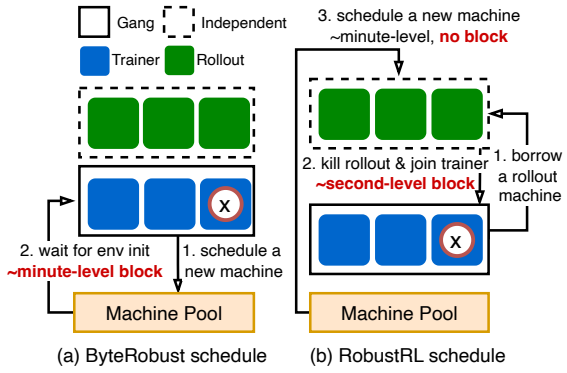

Figure 8: Schedule without extra warmup machines when trainer fails. (a) ByteRobust scheduled a new one from the machine pool. (b) RobustRL borrows a machine from rollout and rollout schedules a new one from the pool.

This design does not work for sync-mode training. In semi-

[2]Code errors in rollout trigger this process as well.

sync and async mode, we can use rollouts as warm standbys for the trainer as shown in Figure 8(b). When a failure occurs, one rollout is killed, and it will join the trainer's initialization. Since rollouts use independent scheduling, requests being processed by the rollout on the replaced machine are handled through the robust rollout strategy (§5.2).

The prerequisite for warmup rollout is that rollouts and trainers are in the same data center and rollout and trainer machines are homogeneous. In deployment, similar to ByteRobust, one can reserve a quantity of max(DP, TP, PP, EP, CP) rollouts as homogeneous machines in the same data center, enabling hot-swapping when a failure or suspected fault occurs in any parallel group. Since warmup rollouts have the same environment as the trainer, they can be quickly substituted when a trainer machine fails.

Our scheme is compatible with existing Trainer-Rollout heterogeneous deployment [89]. For example, we can add an H800 warm Rollout in cluster A for heterogeneous training with cluster A (10×H800 Trainer) and cluster B (10×H20 Rollout)). In addition, Rollout warmup requires no weight resharding because one Rollout machine is corresponding to one DP Group, which means recrimination only reduces inference concurrency. Trainer parallelism remains unchanged after borrowing a machine.

5.2 Robust Rollout

The design goals for robust rollout are to ensure that after a rollout fails, the recovered rollout can **1)** re-establish a connection with the trainer. **2)** Achieve high pulling efficiency for weight from the trainer or other relay rollouts. **3)** Update weights and resume inference as quickly as possible after recovery. To achieve these goals, we have introduced UCX-based weight synchronization (§5.2.1) and discuss the situation of the failure handling during rollout recovery (§5.2.2).

5.2.1 UCX-based Weight Update

The naive-UCX baseline transfers weights from GPU memory on machine 1 to host memory via PCIe, then across machines through a single network interface to machine 2's host memory, and finally from machine 2's memory to individual ranks via PCIe. The path is: PCIe -> NIC -> PCIe.The NCCL baseline uses RDMA GPU zero-copy (IB/RoCE v2) across nodes and intra-node NVLink for rank-to-rank communication. The path is: NIC -> NVLink. Compared to naive UCX, NCCL has a shorter data path and benefits from higher NVLink bandwidth than PCIe. Therefore, we need to shorten the communication path and improve the concurrency of UCX so as to combine the advantages both on the efficiency and elasticity. Previous work like Owl [15] discusses the data relay transfer mechanism to achieve the data-ready object sending to another object. But in RL weight synchronization, we need to further support GPU-to-GPU zero-copy transmission

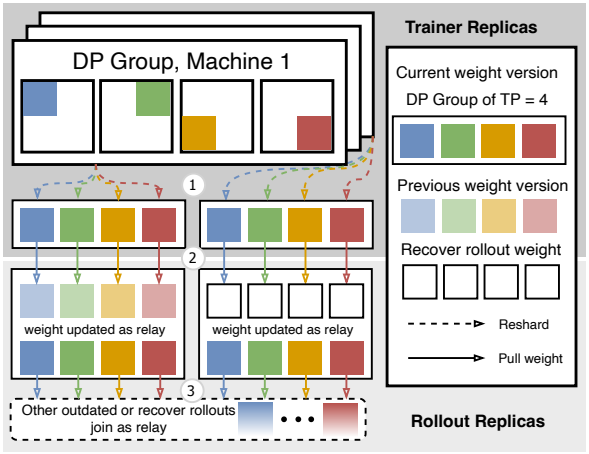

Figure 9: UCX-based weight update.

and weight shard management, and handle fault scenarios throughout the RL training lifecycle. The weight synchronization includes three parts.

① **Reshard.** As shown in Figure 9 step ①, we reshard the model to align with the sharding of the rollout. The implementation is based on verl [22], and this process takes under ten seconds.

② **UCX weight synchronization.** To ensure efficient weight synchronization, we implement point-to-point communication from the trainer's multiple data-parallel (DP) groups to the rollout replicas based on their corresponding ranks shown in Figure 9 ②. Each transfer initiates by placing the model layer by layer into a buffer for transmission. We ensure the buffer size is large enough to saturate the RDMA bandwidth. Besides, we also prevent OOM error on the GPU by preserving a buffer for transfer. The object pulling weights include both outdated and recovered rollouts awaiting weight updates. We ensure that at any given time, each weight is pulled by only one rollout to prevent network bandwidth contention.

Compared with current point-to-point communication design [60], the entire weight synchronization process is executed on the GPU. However, since torch tensors do not directly support UCX transfer, we convert them to cupy arrays [51], as cupy supports byte-level transfers. We use DLPack [12] to support a unified tensor memory layout, achieving zero-copy overhead for the conversion from torch tensors to cupy arrays.

③ **Relay transfer.** A weight updated rollout will act as a relay server, allowing other outdated and recovered rollouts to pull weights from it. Once weight pulling is complete, they join the relay-server set. The pulling process in this stage is the same as the one described in Figure 9 step ③. If a rollout recovers outside of a weight update stage, it directly pulls weights from a relay server, as shown in step ③.

The transfer efficiency between trainers and rollouts from different DP groups can vary due to network fluctuations. We make the entire weight synchronization asynchronous. It

allows the outdated and recovered rollouts to immediately pull from any rollout having the latest weights. A more extreme asynchronous strategy would be to allow asynchronous transfers at the tensor parallelism (TP) granularity, taking into account that stragglers can exist among different NICs [60]. However, the async DP transfer for a 235B model on 4×200Gbps NICs is under 10s. This overhead is sufficiently small compared to a single RL step in minutes or even hours level. Therefore, to maintain the simplicity of the weight synchronization design, we did not pursue further optimization.

In semi-sync RL training, the rollout phase does not begin until all rollouts have finished updating. In this case, the hybrid-rollout in the trainer machine also acts as a relay. In async RL training, the trainer resumes its shard once the first batch of relay rollouts finish their weight update, minimizing training stalls.

5.2.2 Rollout Failure Cases

Preserve the trajectories. We save the trajectories of the prompt of each tool iteration in the `RequestManager` to avoid the rollout progress lost when the rollout machine fails. After each tool iteration, the `RolloutManager` checks the liveness status of the corresponding rollouts. When a failure occurs on a rollout, the previous results are reassigned to other living rollouts. If it is a rollout machine failure, such as an ECC error, the machine is replaced with a new one. Since the `AgentWorker` and the sandbox reside outside the rollout machine, the rollout progress can be preserved.

Relay server failure during pulling. When resuming the weight pull, the rollout will first update its set of target addresses to the living rollouts. If a weight pull fails, it will find another relay server to pull from. The outdated or recovered rollout records their successful pulling progress and pulls the remaining one from other living relay servers.

Trainer failure during pulling. Failure may happen when the trainer prepares to be pulled or being pulled by rollout. In the first case, rollouts close their established connections with the trainer and wait for its recovery before re-establishing connections. In the case where a pull is in progress, the process is interrupted, and the partially updated weights are cleared. Finally, rollouts wait for the trainer to recover before re-initiating the weight pull.

6 Implementation

The implementation of the RL fault tolerance system at the framework level is 8k LoCs of Python for the semi-sync support, fault detection, and robust roles. RobustRL is built on verl [22] and its asynchronous RL extension [44]. We extend this foundation with a semi-sync training mode, which retains the trainer's resharding operation to the inference engine. The implementation of the training scheduler that supports the training architectures in Figure 1.

```
1.  class ElasticRayWorkerGroup
2.      # init function
3.      def __init__(self, cls, args):
4.      # support scale up/down for worker group
5.      def create_worker(...): # and destroy worker
6.      # create RayClass with init_args
7.      def scale_up(num_worker): # and scale down
8.      # invoke the create_worker
9.      def liveness_probe(...):
10.     # detect the liveness of worker groups
11. class ElasticPolicy:
12.     # define when to scale up/down
13.     def should_scale_up(...): # and scale down
14.     # define the scale up condition (recover phase)
15.     def _scaling_loop(...):
16.     # scale up if reinit, scale down if error
17.
18. rollout_cls = RayClassWithInitArgs(RolloutClass, args)
19. - rollout = RayWorkerGroup(rollout_cls, ...)
19. + elastic_rollout = ElasticRayWorkerGroup(rollout_cls, ...)
20. + rollout_policy = ElasticPolicy(elastic_rollout, ...)
```

Figure 10: The replicated ray worker group and scaling policy for the robust and elastic ray worker in RobustRL.

Robust API. In verl, the initialization of workers for the trainer or rollout roles uses the `RayWorkerGroup` (RWG) in line 19 of Figure 10. We further encapsulate this abstraction to handle fault tolerance and recovery scenarios (line 19-20). The `ElasticRayWorkerGroup` (ERWG) supports expansion and destruction, while a `ElasticPolicy` determines the timing for these actions. The ERWG supports monitoring the liveness status of the RWG and defines the initialization and destruction functions, `create_worker` and `destroy_worker`, along with their corresponding pre- and post-processing hooks. The `scale_up` is used to determine the number of workers to expand. For example, multiple rollouts fail or are taken to warm up the trainer (§5.1.3) or scales [79]. The policy defines the conditions for elastic scaling with a polling thread to check the liveness of the RWG. For instance, when the platform detects a machine failure during training, it is captured by this policy and scales up the failure worker. This abstraction avoids the need to repeatedly implement creation and destruction logic for different RWGs.

7 Evaluation

Our evaluation tries to answer the following questions.

Q1. What is the end-to-end training time and ETTR of RobustRL in scenarios with training failures? (§7.2)

Q2. What are the enhancements in RobustRL's fault recovery in terms of restart efficiency and state preservation? (§7.3)

Q3. What is the extra overhead introduced by RobustRL to implement role-specific fault tolerance? (§7.4)

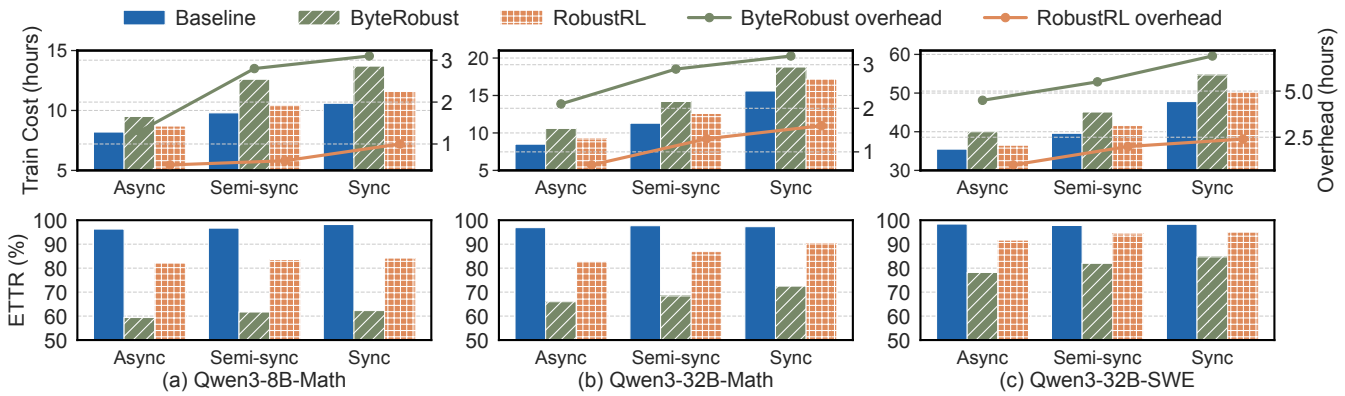

Figure 11: End to end evaluation for RL post-training of Qwen3 models in math [85] and SWE-bench [30]. The first row is the job completion time and the second row is the ETTR. The overhead of the first row is the time gap with Baseline.

7.1 Setup

Testbed. We deploy RobustRL on 32 GPU machines with 256 GPUs. Each machine is equipped with eight NVIDIA H20 96GB GPUs. The intra-machine and inter-machine bandwidths are configured with 900 GB/s NVLink and 4×200Gbps NICs, respectively. Each machine has 2TB of memory. Our software versions are CUDA 12.4, PyTorch 2.4.1, and NCCL 2.21.5. We build our system on verl 0.5.x [22], using Pytorch FSDP2 [87] and vLLM [35] as the training and inference backend. Our checkpointing uses ByteCheckpoint [72] with HDFS as the storage backend. We selected Qwen3-8B, 32B, and 235B-A22B [82] to evaluate the performance because they are the popular models for RL post-training research in academia and industry. We use the 8B and 32B models to test training performance and provide results from the 235B model to test weight saving and weight synchronization costs. The TP size of rollout is 8 to guarantee each rollout occupies a machine.

Baselines. The methods we compare are ByteRobust and our RobustRL. The baseline system is RL training without any injected faults. In ByteRobust and RobustRL, we inject a fault for the trainer at a random time within every 10% interval steps. Since the trainer follows the gang-scheduling, it would trigger the `TaskRunner` to automatically restart all the trainers. This failure frequency setting is higher than in practical production environments if the machine error rate is *i.i.d.* For example, the 100K GPU training in Meta [56] the error frequency is 18min, corresponding to 117 hours in a 256 GPU cluster. Our experiments are designed as stress tests to quantify the benefits of our fault-tolerant design under the extreme, large-scale training scenarios that will emerge in future workloads.

For ByteRobust, we retain only the in-place restart part of a failure, without machine rescheduling. The RL training includes the three architectures in Figure 1. Semi-sync switches from rollout mode to train mode when 50% of the prompts in a batch are completed and the sync mode is 100%. Async is 0% with a one-step warmup. For semi-sync and async mode, the number of GPUs for the trainer and rollouts is equal. We do not compare with the current robust rollout RL system [37, 60] because existing RL systems do not support trainer fault tolerance. Under trainer failures, their behavior is equivalent to ByteRobust. For rollout fault tolerance, the failure or recovery of rollout would not lead to the training bottleneck with further evaluation. For the weight synchronization efficiency, we directly compare with the time of ideal network bandwidth.

Dataset. For reasoning tasks on math with "DAPO-Math-17K" [85], we use a rule-based reward function to score the trajectories. For the multi-turn tool-learning task, we generate the trajectories by obtaining answers through multi-turn interactions with a sandbox in SWE-bench [30]. We use GRPO [59] for both tasks. RbustRL is independent of the specific RL algorithm and the effectiveness of RobustRL can generalize to the others. Our global batch size is set to 512, with 64 prompts per batch, and 8 responses generated for each prompt following the previous research [60]. We set the maximum text length to 65536 and the maximum number of tool interaction turns to 50. We train for 100 steps by default.

Metrics. Our primary evaluation metrics for fault tolerance are the end-to-end training time and the ETTR. We further evaluate the restart time overhead, additional fault tolerance overhead, and the time saved due to fault tolerance for different methods.

7.2 End to End Evaluation

We compare the end-to-end training time and ETTR for different training architectures under various GPU workloads on math and SWE post-training tasks on 128 GPUs.

End-to-End Training Time. As shown in the first row of Figure 11, the Qwen3-8B-Math takes the shortest time because the model is small and the post-training task is relatively simple. The SWE task takes the longest time because the introduction of tool interaction significantly lengthens the rollout time. Taking the Qwen3-8B-Math task as an example, the training time varies for different training architectures,

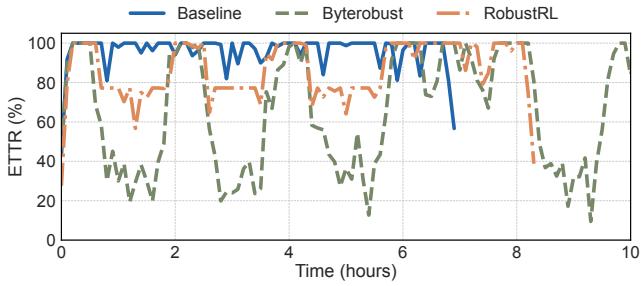

Figure 12: Sliding ETTR of the training process Qwen3-8B-Math. The sampling interval is 5 minutes.

where the async mode is faster than the sync one by mitigating long-tail overhead. Therefore, the end-to-end training time continuously increases from the async to the sync mode.

Since we inject a fault every 10 training steps, the end-to-end training times for ByteRobust and RobustRL are longer than the baseline. The overhead introduced by a fault restart is mainly caused by the restart initialization and the loss of training progress, especially the loss of rollout progress (§5.1.1). In contrast, RobustRL can restart only the trainers, allowing the rollouts to continue executing inference and environment interaction processes. Therefore, the overhead of RobustRL is smaller than that of ByteRobust. As we shift from async to sync training mode, the impact of the long-tail phenomenon becomes progressively more severe, thus the overhead of ByteRobust gradually increases. In sync mode, the rollout and training phases are executed sequentially as shown in Figure 1. By leveraging the role-based fault tolerance strategy, RobustRL explicitly preserves the rollout results. Consequently, if a failure occurs during the training phase, the system can resume directly from the training step, skipping the redundant rollout phase as shown in Figure 6(a). This stands in contrast to approaches like ByteRobust, which necessitate restarting the entire RL step. Lastly, the fault tolerance benefits of using RobustRL on complex tasks like SWE are greater than on math reasoning, because the rollout phase is more time-consuming. The benefit of preserving the rollout progress is more obvious.

In summary, under the extreme condition of a 10% failure rate, RobustRL can save 0.8-2.1h, 1.4-1.6h, and 3.5-4.5h in end-to-end training time compared to ByteRobust on the 8B-Math, 32B-Math and 32B-SWE tasks respectively. The restart overhead of RobustRL for fault tolerance accounts for less than 5% of the total time, while the ByteRobust introduces 20% overhead.

ETTR. As shown in the second row of Figure 11, computation stages such as actor policy updates and rollout are all counted towards the effective training time in RL post-training. In RobustRL, there is a phase where the trainer restarts for fault tolerance while the rollouts continue running. During this process, the recovery ETTR is calculated with the ratio $ETTR_{ratio} = \frac{\#Rollout}{\#Rollout+\#Trainer}$. Since RobustRL significantly reduces restart overhead compared to ByteRobust, its ETTR is better than ByteRobust's, and the advantage is more pronounced in simpler tasks like Qwen3-8B-Math. The reason is that the longer training time is, the more negligible the restart time. In addition, although ByteRobust would lose the progress during the restart, the re-execution of rollout is also counted towards ETTR. So the gap of average ETTR between ByteRobust and RobustRL is not large.

We further show the sliding ETTR for Qwen3-8B-Math in Figure 12 with semi-sync training mode. We sample the training process every 5 minutes. We take the process of 0.8-2h as an example. Since ByteRobust restarts both the trainer and rollouts when a fault occurs, its ETTR is significantly lower at 20%. During the same period, RobustRL's rollouts continue to execute, so the influence of failure is less. Compared to ByteRobust, RobustRL improves ETTR by 18-24% on the Qwen3-8B-Math task.

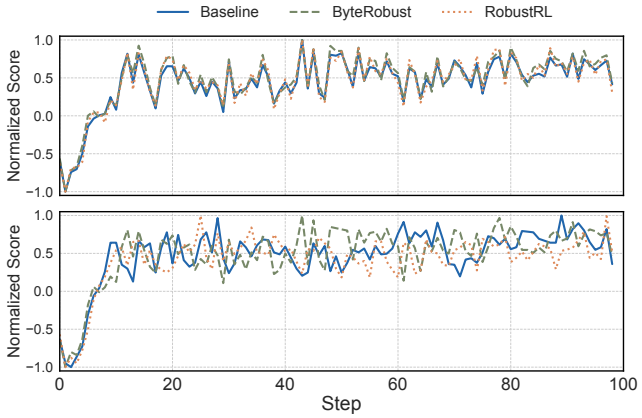

Figure 13: Training consistency in sync (row 1) and semi-sync (row 2) mode. The reward of the answer is normalized.

Training trend. We use the training trend of Qwen3-8B-Math with sync and semi-sync to verify the correctness of training in the presence of failures. We execute 100 training steps and obtain curves with similar training trends and the result is shown in Figure 13. In sync training, the three methods has the similar score tendency because we can make sure the training order of the prompt by batch mode instead of the streaming pipeline mode [89][3]. When the error happens, we can get the prompt and response from `RequestManager` to continue the training in RobustRL. However, the result in sync does not get fully alignment because we do not enable the deterministic inference [69] with FlashAttention [9] and CUDA atomic operations in our evaluation. In semi-sync situation, the results are still not deterministic and the trend difference is larger because the async schedule mode must use the streaming schedule, which cannot guarantee the order of the prompt. But generally, the three methods in semi-sync training shows the similar tendency.

[3]Batch mode is waiting for all the prompts in this batch to finish. Streaming mode is adding a new prompt to rollout if one prompt in this batch finishes.

Comparison under different error injection frequency. We further evaluate the performance of RobustRL and ByteRobust under diverse fault frequencies shown in Figure 14. All experiments are conducted on a 64-GPU in semi-sync mode with the same number of trainers and rollout for Qwen3-8B-Math workload and each experiment runs for 50 training steps. Notably, ByteRobust adopts a fixed checkpoint interval of one snapshot per 5 steps, and faults are uniformly injected throughout the entire training timeline.

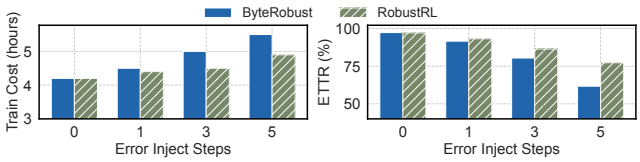

Figure 14: End-to-end training time and ETTR comparison of ByteRobust under different error injection frequency.

In the fault-free scenario, RobustRL and ByteRobust achieve comparable end-to-end training time and ETTR. Although RobustRL adopts a one-step checkpoint, the overall time for a small proportion of the full RL training §7.4. We provide an in-depth breakdown of checkpoint overhead in the following overhead analysis section. As the injected fault frequency increases from 1 to 5 (2%-10% in 50 steps), RobustRL outperforms ByteRobust by 2.2%–12.2% in end-to-end completion time reduction and 2.7%–15.7% in ETTR improvement because the time gain of RobustRL compared to ByteRobust is highly depends on the fault frequency. Considering that the performance advantage of RobustRL approaches saturation at a 2% fault rate, we omit additional evaluations for even lower fault frequencies.

7.3 Robust Benefits Analysis

Trainer restart benefit. Figure 15 shows a comparison of restart time breakdown between ByteRobust and RobustRL for different model sizes and numbers of GPUs. We use the semi-sync case as an example because its startup time is the longest, as the trainer machines need both the training and inference engines, in addition to the rollout roles. The main bottlenecks in starting an RL training task consist of four stages: instance restart, rollout initialization, checkpoint loading, and worker initialization. The "restart instance" stage includes container startup, installation of third-party libraries after startup, Kubernetes scheduling *et al.*. The "worker init" stage is primarily the initialization of the training engine. The "rollout init" stage is the initialization process for the standalone rollout inference engine, not including weight synchronization. The "load checkpoint" is the stage to load from memory to the GPU. When we detect a model anomaly, we asynchronously load the corresponding shards from HDFS to the local memory. The overhead from "rollout init" and "restart instance" phases does not exist in RobustRL, making it faster than ByteRobust. Benefited by the warmup by rollout

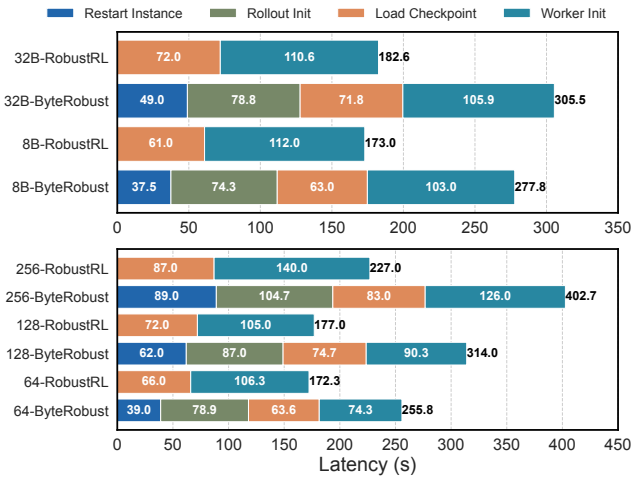

Figure 15: Restart cost comparison of ByteRobust and RobustRL. Row 1 is the result with different model size and row 2 is the result of 8B with different number of GPUs.

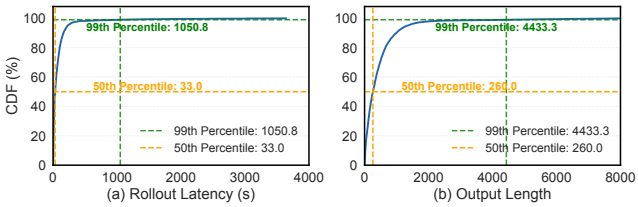

Figure 16: Benefit from preserving the rollout progress (a) Rollout cost CDF and (b) Rollout length CDF of all prompts in Qwen3-32B-SWE.

mechanism in §5.1.3, we avoid the trainer's gang scheduling process when the trainer has machine error because we can schedule the rollout machines to replace.

The second row of Figure 15 shows the results of restart latency of ByteRobust and RobustRL as the number of GPUs increases, where the duration of all initialization stages increases accordingly. The "worker init" overhead in RobustRL is larger than in ByteRobust. Compared to ByteRobust's startup, the trainer's restart includes a destruction phase, which introduces additional overhead for destroying network connections. Overall, RobustRL improves restart efficiency by a factor of 1.5-1.7× compared to ByteRobust.

Preserving rollout progress benefit. In addition to the restart benefit, RobustRL can preserve rollout progress compared to ByteRobust. We recorded the output time distribution and output length distribution for 50k prompts (batch size × steps = 512 × 100) on the Qwen3-32B-SWE task shown in Figure 16. The tail latency reaches 1050s, with an output length of over 4k tokens. RobustRL can avoid the overhead of re-executing this inference. When we extend to larger RL training scenarios involving more conversation turns and more complex tool invocation logic, the benefit can be more significant.

Rollout fault tolerance benefit. A machine failure on a rollout or a machine being borrowed by the trainer does not

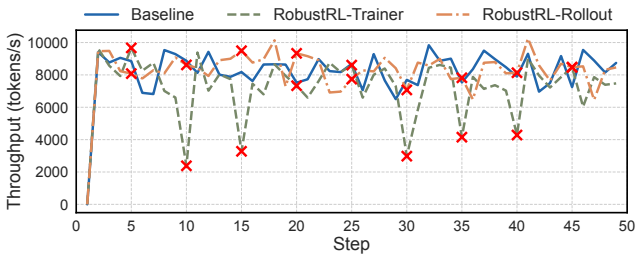

Figure 17: Rollout token throughput of Qwen3-8B-Math with 50 steps. We inject the error each five steps.

influence the RL task. There are multiple rollout instances, and in semi-sync mode, the trainer can also provide rollout capabilities. Compared to the over 300s required for a RL task restart shown in Figure 15, the impact of a rollout restart would not influence the throughput of token decode as shown in Figure 17. In the event of a rollout machine failure in 32B model, the time for a single rollout to start up and begin providing service includes machine scheduling (30s), container startup (under 30s), inference engine startup (49s), and weight synchronization (10s), for a total of 119s.

Throughput under the trainer and rollout failure. We visualize the rollout throughput of semi-sync training in Figure 17. The failure in trainer machines can lead to the throughput decrease because the error can happen when rollout phase finishes. In this phase, the trajectories wait for the consumption by the trainer and continue the next step rollout. The token throughput of rollout failure is not affected because we have multiple rollout replicas.

Detection benefit. ByteRobust [73] applies the 30 seconds interval for network-related fault detection and 10 seconds for the GPU ones (setup in its Table 1). As discussed in §1, it would lead to endless restart of the task since the RL task can have idle time waiting for the response from the tools. The RobustRL can continue the running process. Compared with the cluster-level detection in Figure 2(b), it can save at most about 1000 seconds as shown in Figure 16(a) since we set the hang detection interval for rollout as 60 seconds.

7.4 Robust Overhead Analysis

7.4.1 Weight Synchronization of Trainer and Rollout

Weight synchronization efficiency. Figure 18 shows the weight synchronization efficiency for an equal number of trainers and rollouts. The 235B model requires more than 64 GPUs for the trainer to start as shown in Figure 18(b). The NCCL communication method first gathers all weight to rank 0 and then broadcasts the weight to all the rollouts. The UCX uses point-to-point communication to the corresponding ranks in rollouts. Since UCX can fully utilize all the bandwidth of the NICs, it can have close performance to the NCCL. In addition, UCX allows for dynamic connections, providing fault tolerance capabilities. For a 235B model with FP16 precision, the model size is 470GB. With the 4×200Gbps NICs, the theoretical transfer cost is $\frac{235\times2}{4\times200/8} = 4.7$s. Our UCX-based weight synchronization cost is about 6s. The extra overhead can be introduced by the network bandwidth fluctuations and the async bubbles between NICs in one machine.

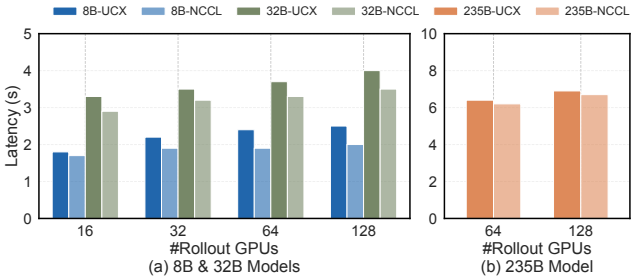

Figure 18: Weight synchronization latency with different number of GPUs and model size. The number of GPUs of trainer and rollout is equal. (a) 8B and 32B and (b) 235B.

We further evaluate the pulling efficiency of our weight synchronization strategy in Figure 19. Our UCX-based transfer can achieve a linear increase in cost even when the number of rollouts grows exponentially. Benefit by the joining as relay design (§5.2.1), the outdated or recovered rollout can pull the weight from the relay server. On the contrary NCCL-based weight synchronization strategy does not support dynamic connection with relay server and all the rollout must pull the weight from the trainer. When the number of rollouts is larger than the trainer, its efficiency decreases.

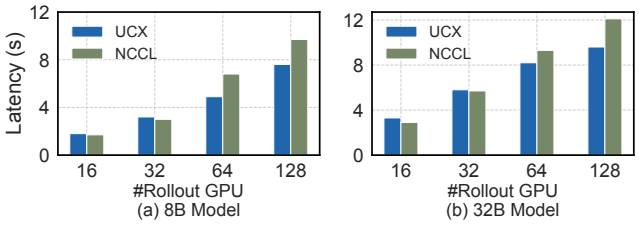

Figure 19: Weight synchronization latency with 16 trainer GPUs and different number of rollout GPUs. (a) 8B and (b) 32B models.

7.4.2 Per-step Checkpoint

Checkpoint efficiency. We evaluate the checkpoint overhead of both FSDP2 and Megatron in Figure 20. It shows the GPU to memory and memory to disk latency with different model size and number of GPUs for checkpoint. The GPU to memory and memory to disk are independent of the model size and the number of GPUs because the saving process only needs to save the shard corresponding to each rank. The 3s blocking time is 1% of the minutes or hours required for one step in the RL task. It guarantees that a per-step GPU-to-memory checkpoint would not lead to the OOM during the process of checkpoint from GPU to memory because each step takes

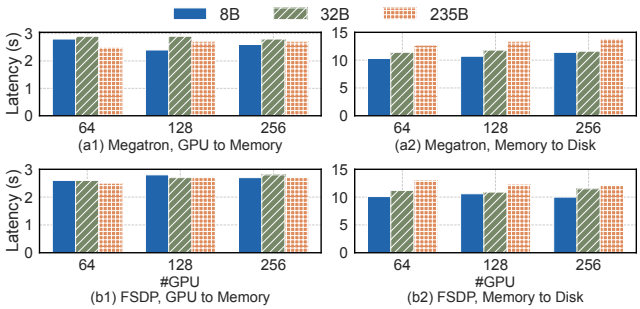

Figure 20: Checkpoint latency of ByteCheckpoint. (a) GPU to memory and (b) memory to disk with different model size and number of GPUs.

over minutes. We have enough time to offload the checkpoint from memory to the disk for about 10s. The disk writing stage is non-blocking, which further minimizes the overhead of checkpoint for the RL training.

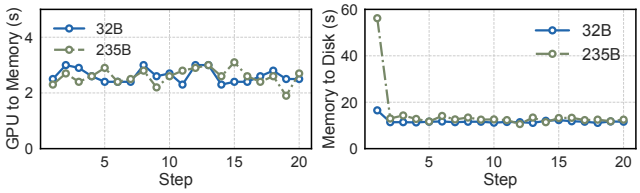

Figure 21: Checkpoint overhead of each step in GPU to memory (left) and memory to HDFS (right).

Continuous checkpoint. In Figure 21, we analyze the latency of GPU-to-memory and memory-to-disk across the first 20 training steps for the 32B model. For the 235B model, checkpoint saving is executed at a consistent 10-minute interval. Consistent with the observations, the GPU-to-memory latency is independent of model scale and training steps. This is because the GPU-to-memory address mapping in ByteCheckpoint [72] is pinned during the initialization phase.

In contrast, for the memory-to-disk persistence phase in Bytecheckpoint, the first weight saving needs extra full traversal of all model tensors, global shard metadata generation, distributed execution planning. Larger models introduce more tensors, larger metadata footprints, and greater volumes of data to be written, leading to a strong correlation between persistence overhead and model scale. Starting from the second checkpoint, however, distributed planning and metadata are fully cached. Redundant operations including tensor traversal, metadata parsing, cross-node communication, and structural construction are eliminated, leaving only raw data copying and disk I/O. This process is no longer affected by model size, but bounded by storage bandwidth for each shard.

Checkpoint frequency > 1. We further evaluate the reward trends under different checkpoint frequencies with a failure injected every 10 steps shown in Figure 22. All experiments are conducted in semi-synchronous mode on the math task using

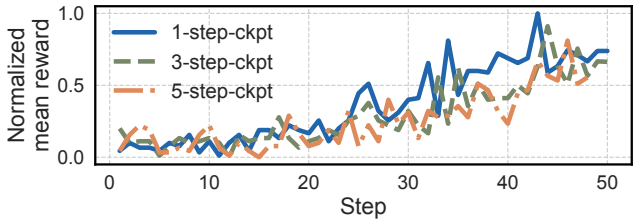

Figure 22: Reward under different checkpoint frequency with 10-step-frequency failures.

the Qwen3-8B-Base model. For a 3-step checkpoint interval, if a failure occurs at the 10th step, the rollback checkpoint is restored from the 9th step, resulting in 1 offline step. Similarly, a 5-step checkpoint interval introduces 5 offline steps when a failure happens at step 10. The experimental results demonstrate that the number of offline steps directly affects the upward trend of reward growth. Such degradation can be more severe for large-scale, highly complex reinforcement learning training workloads like search [31] and coding [30]. Given that the overhead of per-step checkpoint is negligible, we prioritize preserving algorithmic fidelity and training effectiveness in our design.

8 Discussion and Limitation

Limitation of Test Scaling. Frequent failures only emerge in large-scale training scenarios. Under small-scale settings with infrequent failures like 2%, RobustRL exhibits no substantial performance gains over ByteRobust because the benefit is highly dependent on the failure frequency and end-to-end training time. Even so, our experiments simulate large-scale frequent failure conditions via small-scale cluster testbeds, which adequately demonstrates the inherent advantages of RobustRL for future large-scale workloads.

Limitations of Diagnosis Tools. Current fault diagnosis tools are primarily designed for training [11, 38, 66, 73, 74], lacking specific capabilities for RL post-training scenarios. In RL, the presence of multiple roles and complex data and control dependencies makes root cause localization challenging. For instance, an OOM error in one role might be caused by a memory leak in another co-located role in the same machine. Similarly, a role hang event could be due to abnormal metrics from other roles. More precise RL training system diagnostic tools would help developers pinpoint root causes faster during failures, thereby further improving RL training efficiency.

Role-based Hot Updates. Adjusting parameters during training is a common requirement. For example, increasing the batch size when GPU memory utilization is found to be low. ByteRobust [73] proposes a hot update mechanism that allows instances to restart and update in-place, avoiding machine rescheduling. Building upon role-based fault tolerance for hot updates, it helps developers achieve faster role-specific parameter updates. RobustRL reduces restart overhead and

prevents the loss of rollout.

Elastic RL Training. Existing work supports elasticity for the rollout [79,89]. Since the rollout performs inference tasks and uses independent scheduling, elastic scaling does not affect task execution. With the support of RobustRL, the trainer's elasticity in the data-parallel dimension can adopt a similar approach to previous work [8, 23, 36, 86], requiring only a trainer restart, thus reducing the overhead of elastic scaling.

9 Related Work

Fault tolerance systems for LLM training. For training systems, their main goal is to locate system failures and recover as soon as possible through fault detection [10,11,29,38,73,78], checkpoint [47, 72], and elastic recovery [8, 36] to ensure the correctness of model training and reduce task restart overhead. Some works have further extended to elastic training scenarios, focusing on reducing the overhead of scaling due to resource changes. Examples include pipeline redundancy [27] and parallelism adjustment prediction [70] or in spot instance [4]. However, the fault detection strategies for pre-train are not incompatible in RL. For the trainer robust work like Torch-FT [45, 56], it is orthogonal to us since it can accelerate the trainer restart efficiency. But it does not consider the recovery of the trainer from partial failures in conjunction with rollouts, nor the design opportunities presented by the asynchronous execution of rollout.

Fault tolerance systems for LLM inference Existing inference fault tolerance systems primarily operate at the token, and rank levels. The token level refers to utilizing the KV Cache [18, 34, 54] to avoid re-prefilling requests. The rank level refers to scenarios with AF (attention-FFN) disaggregation [39,68,92], where the failure of a sub-ranks is prevented from causing a full service crash. For example, EaaS considers fault tolerance and recovery after an expert machine fails. However, async RL training scenarios also need to consider the interaction between rollouts and the trainer. Additionally, as an offline training task, we just need to guarantee the prompt can be generated.

RL Systems and Fault Tolerance. RL involves both training and inference stages. Considering the long-tail phenomenon of rollout, RL training paradigms have shifted from sync [43,61,90] to async mode [16,60,89,93] to improve training efficiency. Some RL training works have considered the fault tolerance and elasticity of rollouts [60, 79]. However, the probability of failure in the rollout phase is much lower than in the training ones because the communication management of training is more complex. RobustRL further considers the case of trainer fault tolerance, utilizing rollouts as warm standbys and allowing them to continue inference during a trainer failure.

10 Conclusion

We have implemented RobustRL, the first fault tolerance system for RL training that supports all GPU roles against machine failures. Through techniques in *detect-restart-reconnect*, RobustRL can detect the fault by roles quickly, isolate the failure and minimize the restart overhead with rollout progress preserving. RobustRL can achieve 80% ETTR on the 8B-Math training task with 20% higher than ByteRobust on the Qwen3-8B-Math task in an extremely robust case.

Acknowledgements

We thank OSDI anonymous shepherd and reviewers for their insightful and constructive feedback. This work was supported in part by the National Science Foundation of China under Grants (62472375, 62125206), and in part by Zhejiang Provincial Natural Science Foundation of China under Grant No. LD24F020014 and No. LD25F020002, and in part by the Zhejiang Pioneer (Jianbing) Project (2024C01032). The corresponding author is Xinkui Zhao.

References

[1] Nvidia collective communications library (nccl), 2023.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[3] Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to basics: Revisiting reinforce style optimization for learning from human feedback in llms, 2024.

[4] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: Scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 472–487, New York, NY, USA, 2022. Association for Computing Machinery.

[5] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[7] ByteDance Seed. Technical Introduction to the Seed1.6 Model Series. https://seed.bytedance.com/en/seed1_6, 2025.

[8] Zhenqian Chen, Xinkui Zhao, Chen Zhi, and Jianwei Yin. Deepboot: Dynamic scheduling system for training and inference deep learning tasks in GPU cluster. *IEEE Trans. Parallel Distributed Syst.*, 34(9):2553–2567, 2023.

[9] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle

Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[10] Yangtao Deng, Xiang Shi, Zhuo Jiang, Xingjian Zhang, Lei Zhang, Zhang Zhang, Bo Li, Zuquan Song, Hang Zhu, Gaohong Liu, Fuliang Li, Shuguang Wang, Haibin Lin, Jianxi Ye, and Minlan Yu. Minder: Faulty machine detection for large-scale distributed model training. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 505–521, Philadelphia, PA, April 2025. USENIX Association.

[11] Yangtao Deng, Lei Zhang, Qinlong Wang, Xiaoyun Zhi, Xinlei Zhang, Zhuo Jiang, Haohan Xu, Lei Wang, Zuquan Song, Gaohong Liu, et al. Mycroft: Tracing dependencies in collective communication towards reliable llm training. *arXiv preprint arXiv:2509.03018*, 2025.

[12] DLPack. DLPack: Open In Memory Tensor Structure. https://github.com/dmlc/dlpack, 2024.

[13] 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.

[14] Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv: 2504.11536*, 2025.

[15] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. Owl: Scale and flexibility in distribution of hot content. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 1–15, Carlsbad, CA, July 2022. USENIX Association.

[16] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. AReaL: A Large-Scale Asynchronous Reinforcement Learning System for Language Reasoning, June 2025.

[17] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. Recycle: Resilient training of large dnns using pipeline adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 211–228, 2024.

[18] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-efficient large language model serving for multi-turn conversations with cachedattention. In Saurabh Bagchi and Yiying Zhang, editors, *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 111–126. USENIX Association, 2024.

[19] Hao Ge, Junda Feng, Qi Huang, Fangcheng Fu, Xiaonan Nie, Lei Zuo, Haibin Lin, Bin Cui, and Xin Liu. Bytescale: Efficient scaling of llm training with a 2048k context length on more than 12,000 gpus. *arXiv preprint arXiv:2502.21231*, 2025.

[20] Github. OpenRLHF: An Easy-to-use, Scalable and High-performance RLHF Framework. https://github.com/OpenRLHF/OpenRLHF, 2022.

[21] Github. ROLL: Reinforcement Learning Optimization for Large-Scale Learning. https://github.com/alibaba/ROLL, 2022.

[22] Github. verl: Volcano Engine Reinforcement Learning for LLMs. https://github.com/volcengine/verl, 2022.

[23] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. Elasticflow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 266–280, 2023.

[24] 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.

[25] Zhenyu Han, Ansheng You, Haibo Wang, Kui Luo, Guang Yang, Wenqi Shi, Menglong Chen, Sicheng Zhang, Zeshun Lan, Chunshi Deng, Huazhong Ji, Wenjie Liu, Yu Huang, Yixiang Zhang, Chenyi Pan, Jing Wang, Xin Huang, Chunsheng Li, and Jianping Wu. Asyncflow: An asynchronous streaming rl framework for efficient llm post-training, 2025.

[26] Jian Hu, Xibin Wu, Zilin Zhu, Xianyu, Weixun Wang, Dehao Zhang, and Yu Cao. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework, 2024.

[27] 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.

[28] Yuxuan Jiang, Ziming Zhou, Boyu Xu, Beijie Liu, Runhui Xu, and Peng Huang. Training with confidence: Catching silent errors in deep learning training with automated proactive checks. In Lidong Zhou and Yuanyuan Zhou, editors, *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, pages 313–329. USENIX Association, 2025.

[29] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, Santa Clara, CA, April 2024. USENIX Association.

[30] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv: 2310.06770*, 2023.

[31] Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

[32] Komal Kumar, Tajamul Ashraf, Omkar Thawakar, Rao Muhammad Anwer, Hisham Cholakkal, Mubarak Shah, Ming-Hsuan Yang, Phillip HS Torr, Fahad Shahbaz Khan, and Salman Khan. Llm post-training: A deep dive into reasoning large language models. *arXiv preprint arXiv:2502.21321*, 2025.

[33] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019.

[34] 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 Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 611–626. ACM, 2023.

[35] 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.

[36] Mingzhen Li, Wencong Xiao, Hailong Yang, Biao Sun, Hanyu Zhao, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, et al. Easyscale: Elastic training with consistent accuracy and improved utilization on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2023.

[37] Nándor Licker, Kevin Hu, Vladimir Zaytsev, and Lequn Chen. RDMA point-to-point communication for LLM systems. *CoRR*, abs/2510.27656, 2025.

[38] Jinkun Lin, Ziheng Jiang, Zuquan Song, Sida Zhao, Menghan Yu, Zhanghan Wang, Chenyuan Wang, Zuocheng Shi, Xiang Shi, Wei Jia, Zherui Liu, Shuguang Wang, Haibin Lin, Xin Liu, Aurojit Panda, and Jinyang Li. Understanding stragglers in large model training using what-if analysis. In Lidong Zhou and Yuanyuan Zhou, editors, *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, pages 483–498. USENIX Association, 2025.

[39] Ziming Liu, Boyu Tian, Guoteng Wang, Zhen Jiang, Peng Sun, Zhenhua Han, Tian Tang, Xiaohe Hu, Yanmin Jia, Yan Zhang, He Liu, Mingjun Zhang, Yiqi Zhang, Qiaoling Chen, Shenggan Cheng, Mingyu Gao, Yang You, and Siyuan Feng. Expert-as-a-service: Towards efficient, scalable, and robust large-scale moe serving. *CoRR*, abs/2509.17863, 2025.

[40] Chang Lou, Dimas Shidqi Parikesit, Yujin Huang, Zhewen Yang, Senapati Diwangkara, Yuzhuo Jing, Achmad Imam Kistijantoro, Ding Yuan, Suman Nath, and Peng Huang. Deriving semantic checkers from tests to detect silent failures in production distributed systems. In Lidong Zhou and Yuanyuan Zhou, editors, *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, pages 19–38. USENIX Association, 2025.

[41] Yunchi Lu, Youshan Miao, Cheng Tan, Peng Huang, Yi Zhu, Xian Zhang, and Fan Yang. Trainverify: Equivalence-based verification for distributed LLM training. In Youjip Won, Youngjin Kwon, Ding Yuan, and Rebecca Isaacs, editors, *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025*, pages 237–253. ACM, 2025.

[42] Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. Deepcoder: A fully open-source 14b coder at o3-mini level. https://www.together.ai/blog/deepcoder, 2025. Notion Blog.

[43] Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. Real: Efficient rlhf training of large language models with parameter reallocation. *arXiv preprint arXiv: 2406.14088*, 2024.

[44] Meituan search. Recipe: Fully Async Policy Trainer. https://verl.readthedocs.io/en/latest/advance/fully_async.html, 2024.

[45] Meta. Easy per step fault tolerance for pytorch. https://meta-pytorch.org/torchft, 2026.

[46] 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 Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 1112–1127. ACM, 2024.

[47] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, Fine-Grained DNN checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216. USENIX Association, February 2021.

[48] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.

[49] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[50] Mathematical Association of America. Aime 2024, 2024.

[51] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[52] OpenAI. Introducing OpenAI o1. https://openai.com/o1/, 2024.

[53] 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.

[54] 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.

[55] Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, Tom Eccles, Jake Bruce, Ali Razavi, Ashley Edwards, Nicolas Heess, Yutian Chen, Raia Hadsell, Oriol Vinyals, Mahyar Bordbar, and Nando de Freitas. A Generalist Agent, November 2022.

[56] Omkar Salpekar, Rohan Varma, Kenny Yu, Vladimir Ivanov, Yang Wang, Ahmed Sharif, Min Si, Shawn Xu, Feng Tian, Shengbao Zheng, Tristan Rice, Ankush Garg, Shangfu Peng, Shreyas Siravara, Wenyin Fu, Rodrigo de Castro, Adithya Gangidi, Andrey Obraztsov, Sharan Narang, Sergey Edunov, Maxim Naumov, Chunqiang Tang, and Mathew Oldham. Training llms with fault tolerant HSDP on 100,000 gpus. *CoRR*, abs/2602.00277, 2026.

[57] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[58] ByteDance Seed, Jiaze Chen, Tiantian Fan, Xin Liu, Lingjun Liu, Zhiqi Lin, Mingxuan Wang, Chengyi Wang, Xiangpeng Wei, Wenyuan Xu, et al. Seed1. 5-thinking: Advancing superb reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.13914*, 2025.

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

[60] Guangming Sheng, Yuxuan Tong, Borui Wan, Wang Zhang, Chaobo Jia, Xibin Wu, Yuqi Wu, Xiang Li, Chi Zhang, Yanghua Peng, Haibin Lin, Xin Liu, and Chuan Wu. Laminar: A scalable asynchronous RL post-training framework. *CoRR*, abs/2510.12633, 2025.

[61] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 1279–1297, New York, NY, USA, 2025. Association for Computing Machinery.

[62] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, pages 173–191, 2024.

[63] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

[64] Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. Kimi k1. 5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*, 2025.

[65] Meituan LongCat Team. Longcat-flash-thinking technical report. *CoRR*, abs/2509.18883, 2025.

[66] NVIDIA Team. NVIDIA DCGM. https://developer.nvidia.com/dcgm, 2021.

[67] Qwen Team, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv: 2412.15115*, 2024.

[68] Stepfun Team. Step-3 is large yet affordable: Model-system co-design for cost-effective decoding. *CoRR*, abs/2507.19427, 2025.

[69] Thinking Machines. Defeating Nondeterminism in LLM Inference. https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/, 2025.

[70] 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.

[71] Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. Tenplex: Dynamic parallelism for deep learning using parallelizable tensor collections. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 195–210, 2024.

[72] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, Xin Liu, and Chuan Wu. ByteCheckpoint: A unified checkpointing system for large foundation model development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 559–578, Philadelphia, PA, April 2025. USENIX Association.

[73] Borui Wan, Gaohong Liu, Zuquan Song, Jun Wang, Yun Zhang, Guangming Sheng, Shuguang Wang, Houmin Wei, Chenyuan Wang, Weiqiang Lou, Xi Yang, Mofan Zhang, Kaihua Jiang, Cheng Ren, Xiaoyun Zhi, Menghan Yu, Zhe Nan, Zhuolin Zheng, Baoquan Zhong, Qinlong Wang, Huan Yu, Jinxin Chi, Wang Zhang, Yuhan Li, Zixian Du, Sida Zhao, Yongqiang Zhang, Jingzhe Tang, Zherui Liu, Chuan Wu, Yanghua Peng, Haibin Lin, Wencong Xiao, Xin Liu, and Liang Xiang. Robust llm training infrastructure at bytedance, 2025.

[74] Wandb Team. AI is Easy to Productionize. https://wandb.ai/site/, 2025.

[75] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. Kvcache cache in the wild: Characterizing and optimizing kvcache cache at a large cloud provider. *arXiv preprint arXiv:2506.02634*, 2025.

[76] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[77] Bo Wu, Sid Wang, Yunhao Tang, Jia Ding, Eryk Helenowski, Liang Tan, Tengyu Xu, Tushar Gowda, Zhengxing Chen, Chen Zhu, Xiaocheng Tang, Yundi Qian, Beibei Zhu, and Rui Hou. LlamaRL: A Distributed Asynchronous Reinforcement Learning Framework for Efficient Large-scale LLM Training, June 2025.

[78] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. GREYHOUND: hunting fail-slows in hybrid-parallel training at scale. In Deniz Altinbüken and Ryan Stutsman, editors, *Proceedings of the 2025 USENIX Annual Technical Conference, USENIX ATC 2025, Boston, MA, USA, July 7-9, 2025*, pages 731–747. USENIX Association, 2025.

[79] Yongji Wu, Xueshen Liu, Haizhong Zheng, Juncheng Gu, Beidi Chen, Z. Morley Mao, Arvind Krishnamurthy, and Ion Stoica. Rlboost: Harvesting preemptible resources for cost-efficient reinforcement learning on llms. *CoRR*, abs/2510.19225, 2025.

[80] xAI. Grok 4. https://x.ai/news/grok-4, 2025.

[81] Yuxing Xiang, Xue Li, Kun Qian, Wenyuan Yu, Ennan Zhai, and Xin Jin. Servegen: Workload characterization and generation of large language model serving in production. *arXiv preprint arXiv:2505.09999*, 2025.

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

[83] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *CoRR*, abs/1809.09600, 2018.

[84] Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, Zhongzhu Zhou, Michael Wyatt, Molly Smith, Lev Kurilenko, Heyang Qin, Masahiro Tanaka, Shuai Che, Shuaiwen Leon Song, and Yuxiong He. Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales, 2023.

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

[86] Xinyi Zhang, Hanyu Zhao, Wencong Xiao, Xianyan Jia, Fei Xu, Yong Li, Wei Lin, and Fangming Liu. Rubick: Exploiting job reconfigurability for deep learning cluster scheduling. *Proceedings of Machine Learning and Systems*, 7, 2025.

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

[88] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Efficiently programming large language models using sglang. *arXiv preprint arXiv: 2312.07104*, 2023.

[89] Yinmin Zhong, Zili Zhang, Xiaoniu Song, Hanpeng Hu, Chao Jin, Bingyang Wu, Nuo Chen, Yukun Chen, Yu Zhou, Changyi Wan, Hongyu Zhou, Yimin Jiang, Yibo Zhu, and Daxin Jiang. StreamRL: Scalable, Heterogeneous, and Elastic RL for LLMs with Disaggregated Stream Generation, April 2025.

[90] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, et al. Rlhfuse: Efficient rlhf training for large language models with inter-and intra-stage fusion. *arXiv preprint arXiv:2409.13221*, 2024.

[91] Jiecheng Zhou, Qinghao Hu, Yuyang Jin, Zerui Wang, Peng Sun, Yuzhe Gu, Wenwei Zhang, Mingshu Zhai, Xingcheng Zhang, and Weiming Zhang. RL in the wild: Characterizing RLVR training in LLM deployment. *CoRR*, abs/2509.25279, 2025.

[92] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, et al. Megascale-infer: Serving mixture-of-experts at scale with disaggregated expert parallelism. *arXiv preprint arXiv:2504.02263*, 2025.

[93] Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An llm post-training framework for rl scaling. https://github.com/THUDM/slime, 2025. GitHub repository. Corresponding author: Xin Lv.