Laminar: A Scalable Asynchronous RL Post-Training Framework

Guangming Sheng[1*], Yuxuan Tong[2*], Borui Wan[1], Wang Zhang[2], Chaobo Jia[2], Xibin Wu[2], Yuqi Wu[2], Xiang Li[2], Chi Zhang[2], Yanghua Peng[2], Haibin Lin[2], Xin Liu[2], Chuan Wu[1]

[1] *The University of Hong Kong* [2] *ByteDance Seed*

Abstract

Reinforcement learning (RL) post-training for Large Language Models (LLMs) is now scaling to large clusters and running for extended durations to enhance model reasoning performance. However, the scalability of existing RL frameworks is limited, as extreme long-tail skewness in RL trajectory generation causes severe GPU underutilization. Current asynchronous RL systems attempt to mitigate this, but they rely on global weight synchronization between the actor and all rollouts, which creates a rigid model update schedule. This global synchronization is ill-suited for the highly skewed and evolving distribution of trajectory generation latency in RL training, crippling training efficiency. Our key insight is that efficient scaling requires breaking this lockstep through *trajectory-level asynchrony, which generates and consumes each trajectory independently*. We propose Laminar, a scalable and robust RL post-training system built on a fully decoupled architecture. First, we replace global updates with a tier of relay workers acting as a distributed parameter service. This enables asynchronous and fine-grained weight synchronization, allowing rollouts to pull the latest weight anytime without stalling the actor's training loop. Second, a dynamic repack mechanism consolidates long-tail trajectories onto a few dedicated rollouts, maximizing generation throughput. The fully decoupled design also isolates failures, ensuring robustness for long-running jobs. Our evaluation on a 1024-GPU cluster shows that Laminar achieves up to 5.48× training throughput speedup over state-of-the-art systems, while reducing model convergence time.

1 Introduction

Reinforcement learning (RL) has emerged as a transformative paradigm for post-training large language models (LLMs), fundamentally enhancing their reasoning capabilities through iterative policy optimization [1–3]. Contemporary state-of-the-art models, including OpenAI's o1 series [4], DeepSeek-R1 [1], and xAI's Grok 4 [5], leverage RL techniques to achieve unprecedented performance on complex reasoning tasks spanning mathematics, coding, and agentic tasks.

Current RL post-training workflows operate through two main serial phases: generation and training, as shown in Figure 1(a). In the generation phase, rollout models produce trajectories by responding to prompts or interacting with environments. After LLM generates some tokens, the system may trigger an interaction with the environment, which in turn produces feedback. A recent trend is to scale this phase across many rollout replicas [1, 2, 5, 7, 8]. xAI scales an RL training job to 200k GPUs scale with massive rollout generation [5]. The goal is to produce vast and diverse trajectories. These trajectories are then evaluated using verifiers, reward models, or environmental signals [3, 9] and stored in the experience buffer. During the training phase, a subset of these trajectories is sampled from the buffer [2, 10]. The sampled batch is then used to compute policy gradients and update model parameters. This extensive generation is critical for effective policy optimization, but relies on the system's ability to manage the scaled-out rollout process efficiently.

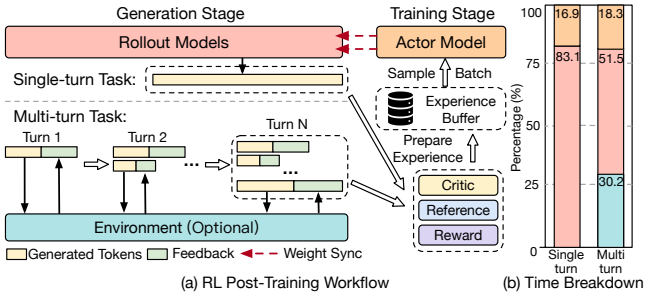


Figure 1. RL post-training workflow and time breakdown of single-turn (math) [2] and multi-turn (code) tasks [6].

Existing synchronous RL systems [11–16] face significant efficiency bottlenecks that limit their scalability. The generation stage dominates overall training time, accounting for up to 83.1% of total execution time in reasoning tasks [2] (Figure 1(b)). This bottleneck stems primarily from long-tail trajectory generation, where the distributions of output length and environment latency are highly skewed. For instance, in mathematics and coding tasks, the 99th percentile output length can exceed the 50th percentile by an order of magnitude (Figure 2). This issue is particularly acute for tasks requiring complex reasoning or multi-turn environment interaction (detailed in §2.2). As generation progresses, only the generation of a few long-tail trajectories remains active, leading to severe GPU underutilization across the cluster. Simply adding more GPUs cannot resolve this load imbalance issue.

Recent asynchronous RL frameworks attempt to address this long-tail problem by decoupling generation and training across different devices and overlapping their execution [17–22]. They typically operate under k-step bounded staleness, where rollouts use actor weights that are up to k

training iterations old. A global synchronization then distributes the new actor weights to all rollouts following a predefined schedule. However, this approach still struggles with fundamental inefficiencies on the long-tail generation problem. A simple and widely adopted one-step staleness (k=1) pipeline [17, 19, 20, 22] cannot effectively hide the generation latency of long-tail trajectories. While increasing the staleness bound ($k > 1$) can absorb this skewness, it can negatively affect model convergence [10, 18, 23, 24]. Consequently, determining an optimal staleness bound that balances overlap efficiency and training convergence remains a difficult tuning problem [25–28].

A fundamental limitation of existing RL systems is their reliance on global model weight synchronization, which is not suitable for the highly variant and dynamic nature of RL workloads. Recent findings show that trajectory lengths change dynamically as models learn [1, 2, 8], exhibiting increasing, decreasing, or fluctuating patterns. Environment interaction latency varies dramatically due to unpredictable API call time or task complexity [29–31]. Current approaches force all trajectories to conform to the same rigid update schedule, regardless of their diverse characteristics, completion status, or inherent execution variability. This inflexibility enforces rigid data and parameter dependencies across actor and all rollouts, preventing effective system scaling.

To address these limitations, we propose Laminar, a scalable and robust asynchronous RL post-training system that eliminates the long-tail trajectory generation bottleneck at production scale while ensuring stable RL training. Our key idea is enabling *trajectory-level asynchrony*, with each trajectory generated and consumed independently at its own optimal pace, to accommodate the vast variability in output length and environment latency. It removes the primary bottleneck to scaling out to thousands of GPUs and enhances trajectory diversity, which is crucial for efficient RL training.

To realize trajectory-level asynchrony, we introduce a fully decoupled architecture that breaks data and parameter dependencies between the actor model and all rollout replicas, and among rollout replicas themselves. This architectural decoupling is further fundamental to achieving robustness at scale. By isolating components, the failure of a single rollout machine does not halt the entire training job, enabling swift recovery that is critical for long-running jobs.

We implement this architecture with two key designs. *First*, we decouple the actor and each rollout replica using a tier of *relay workers*. Functioning as a distributed parameter service, the relays provide asynchronous and fine-grained weight synchronization. The actor can be trained uninterruptedly, while rollouts can retrieve new model parameters from the relays at any time. Nonetheless, individual rollouts may still face long-tail trajectory generation issues. *Further*, we propose a dynamic repack mechanism to consolidate long-tail trajectories from underutilized rollouts into a few dedicated rollout replicas, while liberating other rollouts to update using the latest weight version. This ensures high generation throughput while introducing minimal staleness across the system. Our contributions are summarized as follows:

- We identify trajectory-level asynchrony as the key to scaling-out RL and realize it through a fully decoupled architecture. Our design systematically breaks data and parameter dependencies between all system components and isolates component failures to ensure swift recovery for long-running training (§3).
- We design a tier of relay workers functioning as distributed parameter service, providing fine-grained, robust, and anytime weight synchronization without stalling training (§4).
- We propose a dynamic repack mechanism that boosts generation throughput, by actively monitoring rollout idleness via KVCache-based metrics and concentrating long-tail trajectories generation onto fewer rollout replicas (§5).
- We conduct extensive experiments comparing Laminar with state-of-the-art RL systems [12, 17, 18, 20] at scales up to 1024 GPUs. Our evaluation demonstrates up to 5.48× throughput speedup while ensuring model convergence and robustness (§8).

2 Background and Motivation

2.1 RL for LLM Post-Training

Reinforcement Learning (RL) for LLM post-training was first used to align models with human preferences, known as Reinforcement Learning from Human Feedback [32–36]. More recently, RL has been extended to enhance complex reasoning in domains like mathematics [2, 37] and to develop multi-turn agentic capabilities [6, 31, 38]. As shown in Figure 1(a), RL post-training workflow can be primarily decomposed into two stages: Generation and Training.

Generation stage: The rollout model produces trajectories by responding to prompts through auto-regressive generation or by interacting with environments. For single-turn reasoning tasks, such as solving math problems [39], a trajectory for a simple question may be a short chain-of-thought, while difficult ones can require exploring vast reasoning trees. For multi-turn agentic tasks like SWE-Bench [6], a trajectory consists of interactions with a code sandbox; fixing a simple bug requires only a few interactions, whereas diagnosing a complex issue can lead to more debugging steps.

Training stage: Actor training begins by evaluating each trajectory and assigning it a reward score. This score can come from a separate reward model [32, 33], rule-based functions that define explicit criteria [1, 2], or signals derived from the environment [6, 30]. Each scored trajectory is combined with other metrics, such as advantage estimation derived from a reference model and a critic model, to form a training experience. These experiences are then added to a buffer, where batches are sampled to update the actor model's parameters via backpropagation. While foundational RL methods like PPO [40] rely on a critic model, many recent algorithms (e.g.,

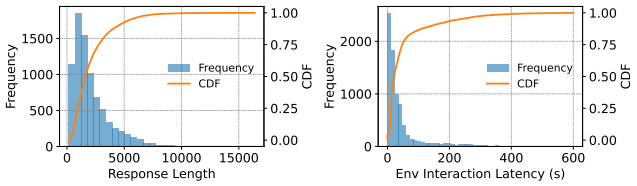


Figure 2. Trajectory length distribution on AIME dataset and execution latency distribution of code sandbox [38, 39].

GRPO [37], RLOO [41] and DAPO [2]) simplify this process by approximating advantage computation through generating multiple trajectories to the same prompt, removing the need for a critic model entirely.

2.2 Recent Trends in RL Post-Training

Exacerbated data skewness. Data skewness in modern RL post-training has become increasingly severe due to the shift toward reasoning-focused models and agentic tasks. The primary source of this skew is trajectory length. As shown in Figure 2, trajectory lengths are highly heterogeneous with the 99th-percentile being ten times the median in some tasks [2, 6]. As shorter trajectories in a batch are generated, only a few long-running trajectory generation tasks remain, leading to severe GPU underutilization. The problem is acute because the memory-bound nature of LLM decoding requires large batch sizes to maintain high throughput. Consequently, the generation stage now dominates the post-training workflow, accounting for up to 83.1% of total execution time in reasoning tasks, as shown in Figure 1(b).

This long-tail problem extends to multi-turn agentic tasks, which involve highly unpredictable environment execution time. Here, the LLM interacts with environments like code sandboxes [6, 38], computer environments [42, 43] and gaming platforms [24, 44]. These environments typically run as external programs or shared services. As shown in Figure 2, the latency of these interactions can vary significantly due to request queuing and varying computational load. This environmental unpredictability, combined with the inherent variance in trajectory length, introduces major efficiency bottlenecks that current systems cannot resolve.

Existing synchronous RL post-training frameworks [11–14, 16, 45] commonly optimize the generation stage and the training stage independently. They adopt sequential and synchronous execution in stages as they are typically built for on-policy RL algorithms. While some systems adopt hybrid execution parallelism (e.g., HybridEngine [11, 12] and Context Switching [13, 14]) to improve throughput, they struggle to mitigate the prolonged GPU idle time caused by skewed trajectory generation, as shown in Figure 3(a). Other systems [15, 46] introduce an inter-stage fusion strategy to overlap the generation stage and experience preparation in the training stage. However, since experience preparation computation only comprises 7.3% of total RL iteration time, the benefits remain limited.

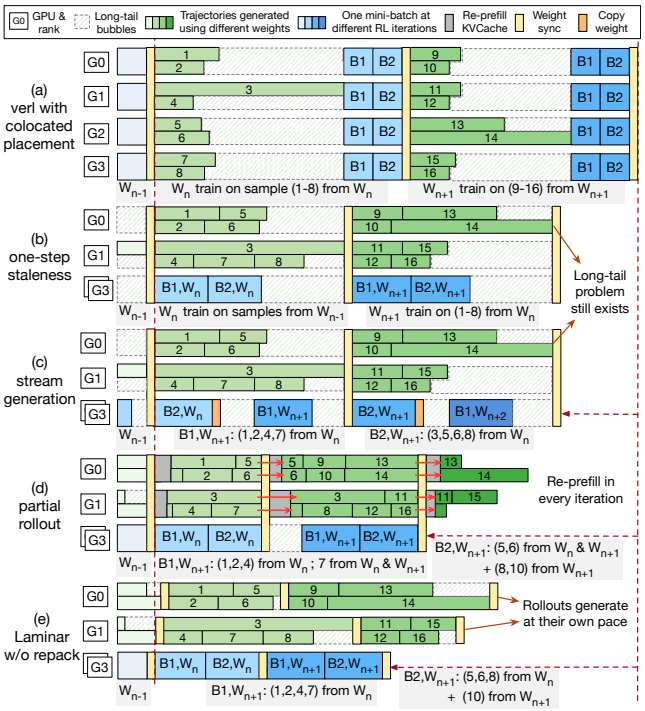


Figure 3. Comparison of RL systems. (a)-(d) adopt GPU-direct communication to perform global weight synchronization, while (c) needs to copy the previous weight version to dedicate memory buffers for later synchronization. (e) performs asynchronous weight synchronization through RDMA on CPU. W_n is the actor weight version at iteration n.

Applying asynchronous RL. Recent advances in RL post-training have increasingly explored asynchronous (i.e., off-policy) RL algorithms to enable parallel execution of generation and training stages. They place actor and rollout models on different devices, allowing trajectories used for training to be generated from previous model versions. Asynchronous RL algorithms have demonstrated remarkable success in domains such as games [24, 44] and robotics [47, 48]. While they improve resource efficiency, their benefits in recent asynchronous RL for LLM post-training systems are severely diminished. The fundamental limitation lies in their batch-oriented design, which fails to effectively mask the generation latency of long-tail trajectories.

2.3 Limitations of existing asynchronous RL systems

Limited scalability due to global weight synchronization. Recent asynchronous RL training systems for LLMs pipeline generation and training across different training iterations, ensuring *k*-step staleness bounds [17–22]. Within an RL iteration, the actor processes a full batch of trajectories by dividing it into smaller mini-batches and conducting a model update on each mini-batch [37, 40]. The actor's final model weights for that iteration are only ready after the last mini-batch is processed. In a widely used *one-step*

staleness pipeline [17, 20] shown in Figure 3(b), rollouts generate a full batch of trajectories using model weights from the past iteration, while concurrently the actor is trained on an older, completely generated batch. Some concurrent research [17, 22] introduces a *streaming generation* approach. As shown in Figure 3(c), the actor is trained first on short trajectories in early mini-batches while consuming long trajectories in later mini-batches. Both systems create a strong data dependency: trainer's progress is tied to the completion of a full batch, forcing it to wait for the slowest trajectory and coupling the execution of all rollouts.

These systems also suffer from a rigid model weight dependency, as all rollouts rely on a global synchronization point to receive new weights after k training iterations. This global synchronization becomes a major bottleneck in tasks with highly skewed generation times (e.g., math competition [39]), failing to effectively hide the long generation time of a few tail trajectories and creating substantial pipeline bubbles. It limits scalability, as adding more GPUs to increase rollouts cannot mitigate the wait for long-tail generation, and allocating additional GPUs per rollout provides only marginal latency reductions, as shown in Figure 4.

Other asynchronous RL systems [7, 8, 18, 19] improve the global synchronization design with *partial rollout*. It interrupts all ongoing trajectories generation to apply the latest actor model in rollouts, and then continues trajectory generation using the latest model weights (Figure 3(d)). A single long trajectory is then composed of several segments, each generated by a different policy version. While this reduces long-tail bubbles, it introduces two main issues. (1) The pause-and-sync cycle incurs significant overhead by forcing rollouts to rebuild the KVCache (i.e., re-prefill) for every interrupted trajectory, repeatedly in each RL iteration, wasting GPU resources without advancing generation. (2) Generating a single response with inconsistent policy versions can harm model convergence, resulting in slower convergence as we show experimentally in §8.2. A comprehensive discussion with more related works is provided in §9.

Undesirable coupling between system throughput and training stability. A core challenge in existing asynchronous RL systems is selecting an appropriate staleness bound k. A larger k can better hide generation latency and improve system throughput. However, it also increases the divergence between the data-generating policy and the policy being trained, which can harm model convergence. Conversely, a smaller k reduces this divergence but is less effective at masking generation latency, leading to poor throughput. Finding a suitable k is a difficult, task-specific tuning problem.

This problem is compounded by the dynamic nature of RL training. As an LLM learns, trajectory length often changes significantly [1, 2, 8]. A staleness bound k that is optimal early in training can become inefficient or unstable over time. Existing systems treat k as a static hyperparameter, creating a rigid parameter dependency that fails to adapt to evolving training dynamics.

2.4 Opportunity and Challenges

Opportunity: Adopting trajectory-level asynchrony to scale up RL. Existing asynchronous RL frameworks are fundamentally constrained by a global synchronization point to update all rollouts. The key opportunity lies in decoupling individual trajectories generation from this global lockstep, enabling *trajectory-level asynchrony*. It allows each trajectory generation to complete at its own pace on a consistent rollout model version and be stored in an experience buffer. Rollouts operate independently and do not stall the execution of one another (Figure 3(e)). This effectively masks generation delays of longer trajectories and allows shorter trajectories to be promptly available for sampling. The fully decoupled trainer samples a batch from the experience buffer without interrupting ongoing rollout generation. Importantly, the staleness of each trajectory (due to the model version it is generated on) emerges naturally from its generation latency, rather than due to a static staleness bound (§6). The staleness varies according to evolving trajectory lengths and diverse environmental latencies throughout training.

We identify the following challenges in achieving this trajectory-level asynchrony and scaling up RL.

Challenge 1: Support asynchronous weight synchronization between actor and rollouts. Implementing trajectory-level asynchrony requires asynchronous and fine-grained weight synchronization, where a rollout fetches the latest actor weights as soon as it completes a batch's generation. As each rollout operates independently and finishes generation at its own pace, rollout updates can occur at any moment during actor training. Existing asynchronous RL systems leverage high GPU-GPU bandwidth for parameter transfers at global synchronization points [17–19]. This approach becomes infeasible for asynchronous rollout updates.

First, direct GPU transfers for asynchronous rollout updates introduce resource contention. It requires dedicated memory buffers, yet GPU memory is a scarce and critical resource in RL training [12, 14, 15]. To maximize throughput, both actor and rollouts operate near peak memory capacity; actor uses large training batches, while rollouts expand their KVCache for larger decode batches [49]. However, asynchronous transfers require buffering. As discussed in §2.3, the final model weights from an RL iteration are not available until the last mini-batch has been trained. Consequently, to support on-demand pulls from rollouts during a training iteration, the actor must hold the previous iteration's weights in a buffer. Similarly, to handle proactive pushes from the actor, a rollout must buffer incoming weights until its batch generation completes. For large models, the buffer can exceed available GPU memory. Even if it fits, this would reduce the training batch size or KVCache capacity, degrading system throughput. Furthermore, the communication

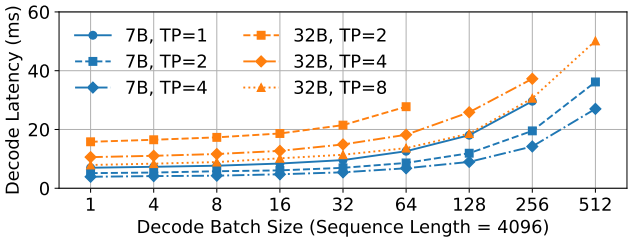


Figure 4. One-step decode latency of Qwen2.5-7B/32B on H800 GPUs, under various tensor parallel (TP) sizes with decode batch sizes up to the KVCache limit.

kernels (e.g., NCCL) may compete for the GPU's streaming multiprocessors with training and decoding kernels, stalling computation on both actor and rollouts.

Second, without these buffers, the actor must stall until rollouts fetch the new weights before it can proceed to the next mini-batch's training, in an RL iteration. This reintroduces a blocking synchronization point, defeating the purpose of trajectory-level asynchrony, especially as the number of rollouts increases.

We seek to minimize GPU idle time while introducing zero extra GPU memory consumption on both the actor and rollouts during asynchronous weight synchronization, by introducing a tier of relay workers (§4).

Challenge 2: Resolve the long-tail generation problem at each rollout. While the relay worker design allows rollouts to operate independently, individual rollouts can still get stuck in long-tail generation (Figure 3(e)). Some rollouts may eventually run only a few trajectories' generation, resulting in low utilization of their GPUs. More critically, these rollouts are prevented from fetching the latest policy (model weights) from the actor, forcing them to continue generating trajectories with increasingly stale weights. Maximizing the number of rollouts that generate near-on-policy trajectories is crucial for training stability and model performance.

We leverage that LLM decoding is a memory-bound operation [50–52]. Rollout replicas stuck on long-tail generations are left with a very small decode batch (e.g., <8) of incomplete trajectories. For a memory-bound operation, decoding such a small batch has nearly the same latency as a much larger one (e.g., 64) [53–55], as shown in Figure 4. This motivates us to consolidate incomplete trajectories from multiple rollouts using the same weight version onto a few destination rollouts, forming a larger decode batch with negligible impact on latency. The released replicas can then be updated with the latest actor weights to produce fresh on-policy trajectories.

However, the dynamic nature of RL training complicates this consolidation. The central challenge is determining when to trigger consolidation and which rollout replicas are involved, as identifying genuinely underutilized rollouts is non-trivial. Prior research has relied on task- and model-specific metrics that require extensive offline profiling to define static thresholds to identify underutilization [15], which are impractical for dynamic RL workloads due to the prohibitive overhead of repeated profiling. We design an efficient solution that uses a lightweight indicator based on KVCache utilization to dynamically make consolidation decisions (§5).

Challenge 3: Ensure robust RL training at scale. Production-level RL post-training jobs are scaling to thousands of GPUs and can run for weeks or months [5, 56]. At this scale, hardware faults are inevitable. As the rollout generation stage often dominates RL iteration time, resilience of rollouts is critical to overall training progress. While the trainer can leverage checkpointing for recovery, existing RL systems provide no fault tolerance for the rollout stage. In addition, GPU-direct communication with NCCL [57] is typically used for weight synchronization [12, 16–18], and NCCL lacks native support for fault tolerance and elasticity [58]. Consequently, a single rollout failure can force a full job to restart from a checkpoint, preventing fast recovery needed for long-running workloads.

In existing systems, each rollout independently manages its active trajectories being generated. A machine failure results in the complete loss of all in-progress work on that node. This loss is particularly costly for complex tasks [6, 31], where a single trajectory can take hours to generate. Regenerating trajectories during failure recovery wastes valuable GPU cycles. Our fully decoupled architecture enables robust long-running RL training by isolating each system component, ensuring rapid and independent recovery (§3.3, §4.3).

3 Fully Decoupled Architecture Design

We introduce Laminar, a scalable and robust asynchronous RL post-training framework designed to scale out RL training jobs, while ensuring robust training. Laminar achieves full decoupling of both data dependencies (i.e., decoupling trainer consumption from trajectory batch generation) and parameter dependencies (i.e., no lock-step weight synchronization) between the actor and all rollout replicas, as well as among rollouts themselves. Such decoupling enables trajectory generation, actor model training, and weight synchronization between actor and rollouts to proceed independently. It eliminates any global synchronization bottlenecks that limit scalability, and effectively isolates faults to ensure rapid, non-disruptive recovery and long-running training. Enabling trajectory-level asynchrony, our framework allows each trajectory's generation to complete at its own optimal pace, while maintaining training stability with diverse trajectories under minimal staleness.

3.1 System Components

Figure 5 gives the architecture of Laminar, consisting of four core modules:

•*Rollout Module* consists of a rollout manager and numerous rollouts. The manager runs on a CPU machine, isolating

it from any failures of GPU machines. It coordinates the rollouts by monitoring their workload and applying trajectory repacking accordingly. It also ensures system stability by monitoring rollout health, managing the weight update topology, and handling fault recovery. Each rollout performs auto-regressive generation to produce trajectories independently and may interact with external environments during generation. After generating its own batch sampled from the prompt pool, each rollout fetches the latest actor model weights from its colocated relay worker.

•*Data Module* manages the lifecycle of trajectories through three distinct storage components, each running as a separate process and storing data in a CPU machine: a prompt pool to supply initial states for generation, such as a math or coding question; a partial response pool that centrally stores in-progress trajectories to ensure fault tolerance; and an experience buffer that holds completely generated trajectories. Interaction with this buffer is managed by a writer and a sampler. Flexible APIs are provided for both writer and sampler, allowing users to customize the sampling strategy for training and the eviction strategy for removing old experiences when buffer capacity is not enough.

•*Relay Workers* function as a hierarchical parameter service. The rollout manager designates one relay as the master, which receives updated weights from the actor and broadcasts them to the other relays. Each relay is a separate CPU process running on each rollout GPU machine, hosting the latest actor model weights in local CPU memory. This allows any rollout replica to pull a new version on demand without blocking GPU computation on trainer and other rollouts.

•*Trainer* samples batches of experiences from the experience buffer to perform model updates according to the selected RL algorithm, and may involve multiple models, such as actor, critic, and reference models [37, 40]. The models are colocated on the same set of GPUs, and are executed sequentially in a time-sharing manner [11, 12].

3.2 Training Workflow

The continuous, asynchronous training workflow of Laminar is designed to maintain high training throughput when scaling up. It begins as rollouts pull prompts from the prompt pool to generate trajectories on their GPUs (step ①). For fault tolerance, in-progress trajectories are streamed to the partial response pool (step ②). Upon generation completion, they are moved to the experience buffer (step ③). In parallel with rollout generation, the trainer samples completed trajectories from the experience buffer to perform model training (step ④). This fundamental decoupling of data production from consumption is key to the system's scalability.

After a model update, the trainer pushes the new actor weights to the master relay and immediately resumes its next training iteration without waiting for weights to be fully distributed to other relays or rollouts (step ⑤). The master relay then broadcasts the new weights directly to all

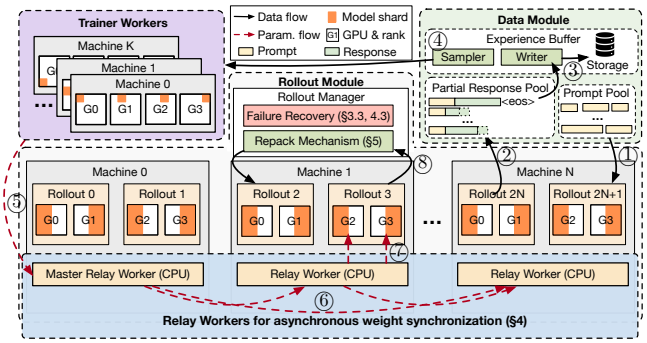


Figure 5. Laminar architecture and training workflow.

other relays using RDMA, which occurs in the background on CPU memory without affecting ongoing GPU-based generation on the same machine (step ⑥). A rollout can fetch the latest weights from its colocated relay at any time, over high-speed PCIe with minimal latency (step ⑦). This decouples parameter dependencies across the actor and all rollouts in the system.

Meanwhile, Laminar actively manages workload imbalance on rollouts to maintain high generation throughput. Rollout manager monitors rollouts stuck on long-tail trajectories generation and triggers a repack mechanism (step ⑧). The repack consolidates trajectories from underutilized rollouts onto fewer rollout, freeing the former which can pull the latest model weights and reducing system-wide staleness.

3.3 Fault Tolerance and Recovery

The decoupled architecture in Laminar allows dynamic removal and addition of rollouts without halting training and losing data, attaining elasticity. Individual faults in rollouts, relays, and the trainer are isolated, enabling rapid recovery without a costly global restart. This resilience is critical for large-scale training and is handled in key stages of the training workflow.

When a fault occurs during rollout generation, our heartbeat-based failover mechanism quickly detects a faulty rollout and notifies the rollout manager. We first attempt to recover by re-initializing the faulty replica on the same GPUs. If the fault persists after re-initialization, we then evict the entire affected machine. During this, in-progress trajectory states remain safe in the partial response pool. The rollout manager redirects interrupted trajectories to healthy rollouts with the same weight version, or waits for replacement machines if none are available. Recovery is swift as new machines initialize rollouts and corresponding relays by synchronizing with the master relay for the latest weights or loading specific weight versions from actor checkpointing files.

Relay faults during weight synchronization are managed by our fault-tolerant relay broadcast mechanism (§4.3). Faulty relay workers are detected instantly without a timeout, and the communication scheduler rebuilds the broadcast chain using healthy nodes. This recovery can be completed within seconds without disrupting ongoing rollout generation.

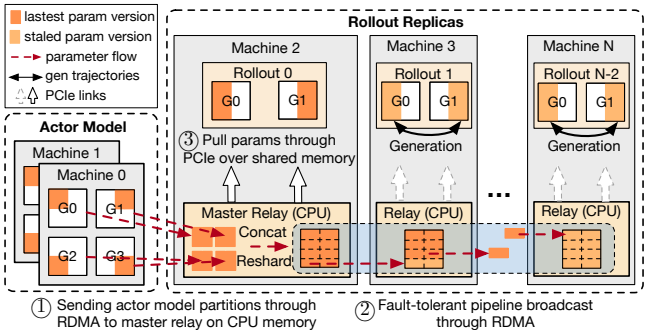


Figure 6. Asynchronous weight synchronization workflow.

Trainer faults are handled by standard checkpoint recovery methods [59–61], with actor model weights checkpointed periodically. When a trainer worker fails, it is evicted [62, 63] and then recovered from the latest checkpoint. During this period, rollouts continue generation with the latest available weights. Once recovered, the actor resumes sampling from the experience buffer and resumes training.

4 Asynchronous Weight Synchronization Using Relay Workers

4.1 Design Considerations

Inefficiency of using a storage system. Traditional RL systems often involve a storage system for weights synchronization between the actor and rollout models. SRL [64] relies on a file system (NFS [65]) and OpenAI-Five [66] utilizes key-value stores (Redis [67]) to transfer and store the published weights. When model sizes were measured in megabytes, weight synchronization through these storage systems incurs reasonable delay; however, they are ill-suited for LLMs with gigabytes scale of model sizes. *First*, substantial serialization and I/O overhead is incurred. Our profiling of a 32B LLM shows that serializing a single 4GB model shard takes approximately 8 seconds. Transferring this volume of data over a standard TCP network to/from the storage system adds another 10 to 20 seconds of latency into the critical path of every rollout. *Next*, the storage system renders a contention bottleneck when numerous rollouts simultaneously pull the latest weights, incurring long and unpredictable latencies for weight updates. Instead, we exploit local host memory and high-bandwidth networks like RDMA for efficient weight synchronization, avoiding serialization and a single point of contention.

Hosting relay workers in local host memory. Direct GPU-to-GPU transfers for asynchronous weight synchronization incur prohibitive memory overhead and computational stalls (§2.4). Allocating separate GPUs dedicated as relays would be computationally wasteful and cost-prohibitive. We introduce a tier of relay workers that run in CPU memory of rollout machines, exploiting that such host memory remains largely underutilized. The CPU-based intermediaries decouple the actor model weights from asynchronous rollout demands and leverage RDMA for efficient and robust weight synchronization among machines.

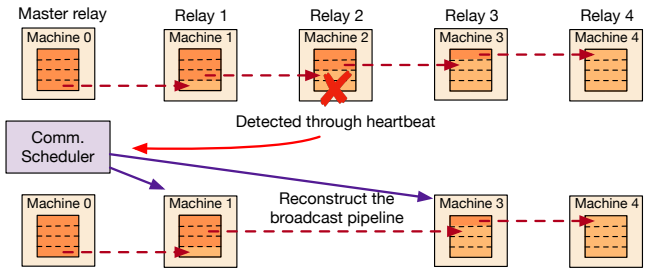


Figure 7. Swift recovery during relay broadcast.

4.2 Hierarchical Relays and Their Workflow

Our relay worker hierarchy is designed to attain two goals: **1)** to minimize actor stall time during weight publication, and **2)** to ensure low-latency access for any rollout requesting a new version. For the first goal, we designate a single master relay: the actor transfers its updated weights to this master relay and can immediately resume training, effectively hiding the broadcast latency (step ① in Figure 6). After receiving the latest weights, the master relay reshards them according to the rollout sharding strategy (e.g., tensor parallelism among GPUs a rollout runs on) [11–14]. A single master relay may become a bottleneck when serving weight requests from multiple rollouts. We further adopt one relay worker per rollout machine to distribute the workload, and enable rollout's low-latency access to updated weights anytime.

The updated weights must be efficiently propagated from the master relay to all other relays. We implement this with a chain-based pipelined broadcast using RDMA (step ②). The broadcast pipelines model chunk transfers along a chain of relays, overlapping communication among different hops. This makes the broadcast time nearly constant regardless of the length of the chain scale [68], which we formally analyze in Appendix D. Broadcast time of less than 1.6 seconds is incurred for a 72B model from the master to 127 other relays (Figure 18 in Appendix D). This broadcast time is negligible compared to the lengthy trajectory generation time at hundreds to thousands of seconds [2, 6, 38]. Rollouts fetch the latest weights through PCIe links from their colocated relay at any time, without waiting for the resharding and broadcast to complete (step ③).

4.3 Fault-Tolerant Relay Broadcast

As RL training scales to thousands of GPUs, hardware and software faults become inevitable. We design a fault-tolerant relay broadcast scheme. The communication scheduler in the rollout manager establishes the broadcast pipeline connecting the relay workers on rollout machines. The master relay receives weights from the actor, chunks them, and broadcasts them down the chain in a pipelined manner. If a rollout machine fails, the scheduler detects the failure via heartbeat

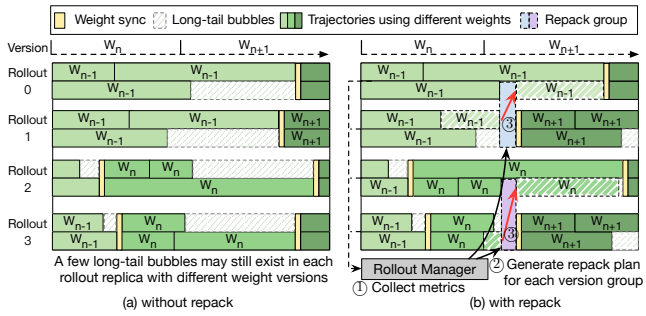


Figure 8. Workflow of the Repack Mechanism. Dashed timeline shows the weight version in the trainer.

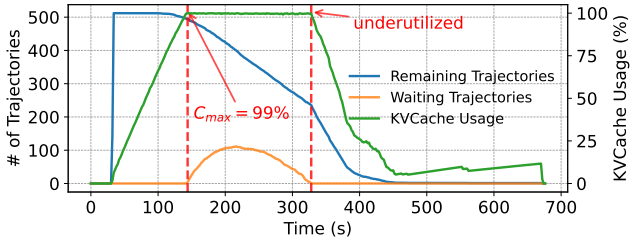


Figure 9. KVCache utilization lifecycle during rollout generation. Usage ramps up to a threshold, then remains steady while remaining trajectories decrease, and finally falls, marking the idle phase that enables trajectory repacking.

monitoring. The failed machine is evicted and the scheduler immediately reconstructs the broadcast chain among the remaining rollout machines, as depicted in Figure 7. This repair is a constant-time ($O(1)$) operation that can be completed in less than one second. If the master relay fails, the rollout manager selects a new master from the available relays and rebuilds the broadcast chain. The trainer is then notified of the new master relay's IP address, for resuming weight distribution. During this fast repair, RL training continues seamlessly, and rollout generation on healthy rollout machines is not affected, as rollout generation and relay communication are isolated on different processes in each machine (§7).

5 Bubble Elimination in Long-tail Trajectory Generation

Each rollout fetches the last model weight upon its completion of a trajectory batch generation. At any time, different versions of the model weights can be used by different rollouts in trajectory generation, while groups of rollouts are performing generation using the same weight versions. To remove GPU idle time on rollouts generating long-tail trajectories, we introduce a repack mechanism that consolidates in-progress trajectory generation from these straggler rollouts onto a few designated rollouts within the same weight version group. Our design involves three key aspects: establishing the workflow of repacking, monitoring rollouts trapped in long-tail generation, and determining suitable repack destination rollouts.

5.1 Workflow of the Repack Mechanism

Our repack mechanism is primarily triggered by a periodic check (e.g., 5 seconds), which detects if any rollout replicas are trapped in long-tail trajectories generation and thus being underutilized. Additionally, a repack is also initiated immediately after the trainer completes weight updating on a global batch, aiming to more rapidly free up rollouts for on-policy trajectories generation with the latest model version.

The repack workflow, given in Figure 8, begins with the rollout manager collecting progress metrics from all rollouts and grouping them by their model weight versions (step ①). Within each group, the rollout manager identifies rollouts undergoing long-tail generation using an idleness metric (§5.2). Then a packing algorithm is performed to decide the repacking plan that consolidates in-progress trajectories from multiple rollouts into fewer destination rollouts (step ②). The rollout manager then transfers unfinished trajectories of the long-tail generation rollouts to destination rollouts accordingly (step ③).

5.2 Online Trajecgtory Repacking

Upon each triggering, the packing algorithm dynamically partitions a group of rollouts into sources (conducting long-tail generation to be released) and destinations (to consolidate long-tail trajectories generation onto).

Idleness metric. We design the idleness metric to identify GPU-underutilized rollouts. Prior approaches rely on a predefined threshold of remaining generation requests to detect the long-tail effects [15]. Determining an optimal threshold requires extensive, per-job offline profiling, which is impractical for RL deployments where workloads can vary significantly. Instead, we leverage *KVCache utilization* as a more direct indicator of resource pressure in memory-bound LLM generation. Our observation is that KVCache capacity is the primary barrier to scaling the parallel-decode batch size in memory-bound rollout generation [50–52]. We have observed that the latency per decode step remains stable even as the batch size increases (Sec. 2.4).

Figure 9, whose results are from a 32B model generating a batch of 512 trajectories on H800 GPUs (TP=4), shows that a rollout's KVCache usage follows a distinct lifecycle according to a natural threshold, $C_{\max}$ (e.g., above 99% of total KVCache), which indicates full KVCache utilization. Initially, usage ramps up to this threshold as the rollout is filled with active token generation of trajectories. KVCache utilization then remains at this peak even as trajectories finish. This is because waiting trajectories in the batch immediately fill the newly available KVCache space. The KVCache usage begins to fall from this peak only when no waiting trajectories are left and the remaining running trajectories complete. We consider the rollout idle from this time on and its remaining

long trajectory generations become candidates for repacking. This consistent threshold behavior across different RL workloads eliminates the need for workload-specific threshold tuning, facilitating our identification of source rollouts.

Repacking algorithm. We consider trajectory repacking as a bin packing problem [69, 70], where each underutilized rollout represents an item and each destination rollout acts as a bin. The destinations are selected from the pool of underutilized rollouts to achieve the most densely packed KVCache after the repack. Our primary goal is to maximize the number of released source rollouts (aka minimize the number of destination rollouts) with minimal overhead and negligible impact on generation latency (verified in §8.4). We design an efficient heuristic algorithm (Algorithm 1), inspired by the Best-Fit approach [71].

We define a rollout's capacity using two key indicators: its KVCache utilization, C_{used}, and its current trajectory count N_{reqs}. The KVCache threshold, $C_{\max}$, serves as a memory capacity limit that bounds the decode batch size, representing the maximum number of trajectories a single decode step can hold. We also enforce a batch size upperbound, B, representing the maximum number of trajectories that can be decoded in parallel with only a negligible increase in latency. This limit is determined using the roofline model [72, 73], which identifies the point where the decode operation transitions from being memory-bound to compute-bound. Exceeding this limit would unacceptably increase generation latency. A rollout becomes one of the candidates S for repacking if it is in its ramp-down phase (its KVCache utilization is non-increasing and below the $C_{\max}$ threshold) and its remaining trajectory count is smaller than the upperbound B (Line 3). Then we sort the candidate rollouts S and prioritize releasing those with the smallest KVCache footprint first (Line 4), as smaller workloads are easier to be replaced.

The core of the algorithm is an iterative matching process to match source rollouts to destinations (Lines 6-13). For each source rollout, a set of valid destinations, D_s, are identified. A valid destination rollout is a candidate rollout which is not already slated for release and has sufficient headroom (Line 9), as verified by the `CanFit` procedure. In `CanFit`, the destination's current KVCache usage and request counts are summed with those from the source, plus any other workloads already assigned to this destination in the current plan P; the projected total KVCache usage and request count must not exceed $C_{\max}$ and B, respectively (Lines 14-17). From this set of valid destinations, we select the one that will become most densely packed in terms of KVCache utilization after the trajectory transfers (Line 11). This approach preserves capacity in other destinations for larger workloads. If a valid destination is found, the trajectory move is recorded, and the source is marked for release.

The algorithm runs in $O(|S|^2)$ complexity, where $|S|$ is the number of candidate rollouts, efficient in practice due to the typically small size of $|S|$. By creating larger batches on destination rollouts and maximizing the release of source rollouts, our algorithm simultaneously increases generation throughput and promotes the generation of on-policy data.

Algorithm 1 Best-Fit Trajectory Consolidation

1: **Input:** Set of rollout replicas R, KVCache threshold $C_{\max}$, Roofline batch size B
2: **Output:** A consolidation plan P of (source, destination) pairs
3: $S \leftarrow \{r \in R \mid r.C_{\text{used}} < \min(C_{\max}, r.C_{\text{prev}}) \wedge r.N_{\text{reqs}} < B\}$
4: $S \leftarrow \text{sort}(S, \text{key} \leftarrow r.C_{\text{used}})$
5: $P \leftarrow \emptyset$; $\mathcal{E} \leftarrow \emptyset$ *// Plan and set of emptied replicas*
6: **for all** $s \in S$ **do**
7: **if** $s \in \mathcal{E}$ **then**
8: **continue**
9: $D_s \leftarrow \{d \in S \mid d \notin \mathcal{E} \wedge d \neq s \wedge \text{CanFit}(d, s, P)\}$
10: **if** $D_s \neq \emptyset$ **then**
11: $d^* \leftarrow \text{argmax}_{d \in D_s} \left(d.C_{\text{used}} + \sum_{(s',d') \in P, d'=d} s'.C_{\text{used}}\right)$
12: $P \leftarrow P \cup \{(s, d^*)\}$; $\mathcal{E} \leftarrow \mathcal{E} \cup \{s\}$
13: **return** P
14: **Procedure** CanFit(d, s, P):
15: $C_{\text{load}} \leftarrow d.C_{\text{used}} + \sum_{(s',d') \in P, d'=d} s'.C_{\text{used}}$
16: $N_{\text{load}} \leftarrow d.N_{\text{reqs}} + \sum_{(s',d') \in P, d'=d} s'.N_{\text{reqs}}$
17: **return** $(C_{\text{load}} + s.C_{\text{used}} \leq C_{\max}) \wedge (N_{\text{load}} + s.N_{\text{reqs}} \leq B)$

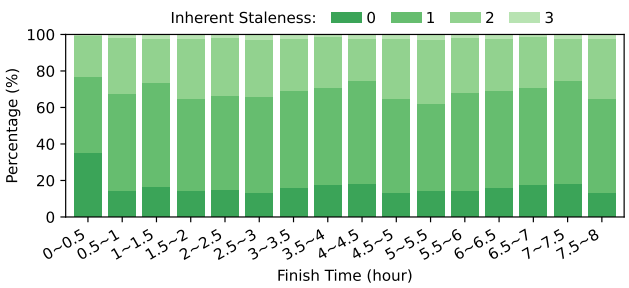


Figure 10. Inherent staleness distribution over finish time ranges of trajectory generation during RL training of a 7B model on 64 H800 GPUs using Laminar.

6 Trajectory-level Asynchrony Analysis

Our asynchronous weight synchronization and repacking mechanism enable true trajectory-level asynchrony. Trajectories are generated independently, each proceeding at its own pace with minimal bubbles in each rollout. This asynchrony gives rise to a key property of each trajectory's generation: *inherent staleness*. If a trajectory is generated using model version K whose generation is finished when the actor model version becomes M, we define its inherent staleness as $M-K$. This staleness is determined purely by the trajectory's generation latency and the trainer's model update speed.

Under our fully decoupled/asynchronous design, Laminar does not require an explicitly configured staleness bound. Instead, the staleness of trajectories is naturally decided according to system dynamics (resource configuration and generation/training progress), eliminating any manual tuning of staleness bound. Especially, minimal inherent staleness is pursued with our rollout update design, as we eliminate

idle time and update model version on each rollout as soon as its batch generation is completed or released. Figure 10 shows that the inherent staleness of trajectories in Laminar remains consistently low (typically under 3). Laminar fluidly caters to the shifting minimal staleness rendered by dynamic RL workloads, accommodating evolving trajectory lengths and variable generation latencies.

Upon completion, trajectories are moved to the experience buffer for trainer sampling. Various sampling strategies can be adopted by the trainer (e.g., priority-based sampling [10, 74] on metrics like temporal-difference errors, importance ratio). Experience sampling is a classic topic in RL; developing effective sampling strategies utilizing massive experiences efficiently generated by Laminar is a promising direction for future exploration, and is orthogonal to the current paper.

7 Implementation

Laminar is implemented in ~11k lines of Python code (LoC).

Decoupled RL training framework. The trainer module and rollout module are implemented with 3.8k LoC on top of verl [12], placed on different sets of GPUs. The partial response pool stores the tokens and statistics of each ongoing trajectory from each rollout for fast recovery and online analysis. Inter-module data transmission is implemented with Ray [75] through Remote Process Calls.

Relay workers are implemented in 1k LoC. Each relay worker runs as a separate process, colocated with rollouts on the same GPU machine. For efficient intra-machine weight transfer, a relay worker uses pinned CPU shared memory. This allows rollout workers to directly load model shards onto their GPUs via fast PCIe links. We built a resilient communication layer for chain-based pipelined broadcast using Unified Communication X (UCX) [76]. To ensure robust operation, the layer also integrates fault tolerance through heartbeat monitoring and dynamic chain rebuilding.

Repack mechanism is supported by the rollout manager, which is implemented with 1.6k LoC.

8 Evaluation

Testbed. We deploy Laminar on a cluster of 128 machines (1024 GPUs in total). Each machine is equipped with 8 NVIDIA H800-80GB GPUs inter-connected with 400GB/s NVLink. The inter-machine bandwidth is 8×400Gbps. Our experiments use the following software versions: CUDA 12.6, PyTorch 2.7.1, NCCL 2.26.2, and vLLM 0.9.0.

Models. We choose Qwen2.5 models of sizes 7B, 32B, and 72B [77], which are popular models for RL post-training research in academia and industry.

Baselines. We compare Laminar with four categories of RL post-training frameworks: *synchronous*, *one-step staleness*, *stream generation*, and stream generation with *partial rollout*: **(1)** For the synchronous baseline, we use verl [12] v0.5.0, a state-of-the-art RL training framework, configured with its optimal colocate placement. **(2)** We implemented the asynchronous *one-step staleness* and *stream generation* pipelines ourselves on top of verl for fair comparison, alleviating implementation bias. This is necessary as many prominent and concurrent asynchronous RL training systems are not open-sourced [17, 19], only runnable on Ascend NPUs [22], or with limited system optimization [20, 23] in training, generation, and weight syncing stages. **(3)** For partial rollout, we use AReaL [18] v0.3.0, a concurrent research that implements partial rollout with stream generation (truncating ongoing trajectory generation of rollouts and adopting updated weights to continue generation of these trajectories) and uses an algorithm to mitigate the impact of different policy versions within each trajectory.

To ensure rigorous comparison, all baselines except AReaL use vLLM [50] for generation and PyTorch FSDP [78] for training. AReaL utilizes its modified SGLang [52] for generation and only supports Megatron-LM [79] for training.

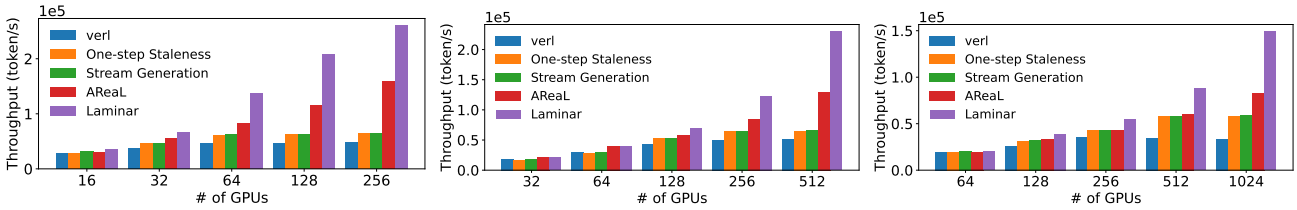
Datasets. We perform RL post-training on "DAPO-Math-17k" dataset [2], which is widely used for math solving and multi-turn tool-calling tasks [38, 80]. Maximum input and output lengths are set to 2K and 16K, respectively. Distributions of model output length are given in Appendix A.1.

Settings. We evaluate Laminar on both math reasoning and multi-turn tool-calling tasks. For both tasks, we utilize an open-source implementation of GRPO algorithm [37] with a higher clipping range in the RL loss (i.e., Clip-Higher) [2], which is widely adopted for RL post-training. The effectiveness of Laminar does not rely on any specific RL algorithm and can generalize to others such as PPO [40]. We utilize a rule-based reward function to score trajectories on both tasks following [1, 2, 38]. For tool-calling task, the rollout interacts with a code sandbox [30] to generate reasoning steps with multi-turn code execution for solving math problems. The global training batch size is set to 8192 with 512 prompts each having 16 responses, and the number of mini-batch update steps per training iteration is 16; the maximum number of tool calls is set to 8, following previous research [2, 38].

Metrics. We use throughput (tokens/sec) as the main performance metric [12], computed by dividing the total number of tokens in prompts and responses in a global training batch by one RL iteration time (the duration between consecutive actor update completions). All reported performance numbers are averaged over 5 RL iterations after a 10-iteration warm-up.

8.1 End-to-End Training Throughput

Figures 11, 12 show the training throughput of various RL frameworks under different model sizes. We tune the optimal placement under each baseline by matching the generation and training throughput to reduce GPU idleness between the two stages. The placement configurations are listed in Appendix A.2, denoting a key performance factor when actor and rollouts are disaggregated [17]. We set the staleness



(a) 7B (1.12×~5.48×) (b) 32B (1.01×~4.52×) (c) 72B (1.03×~4.51×)

Figure 11. Training throughput on single-turn task (mathematical reasoning).

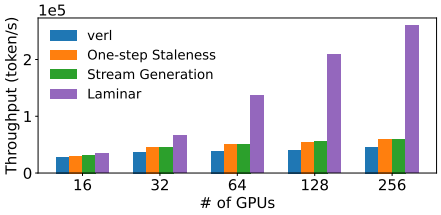
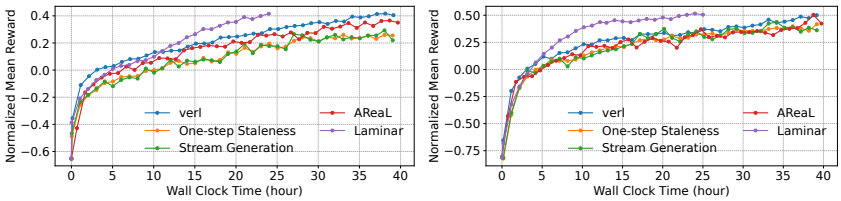


Figure 12. Training throughput on multi-turn task (tool calling). (7B, 1.21×~5.42×)



(a) 7B (32 GPUs) (b) 32B (128 GPUs)

Figure 13. Reward with respect to wall clock time.

bound of stream generation to 1 following [17, 22] and set AReaL's to ∞ to achieve the largest training throughput. The maximum staleness in Laminar, as we observed in all experiment settings, is 4. The global training batch size remains fixed when the cluster scales, a strong scaling setting which is standard practice in RL research [12, 14, 18].

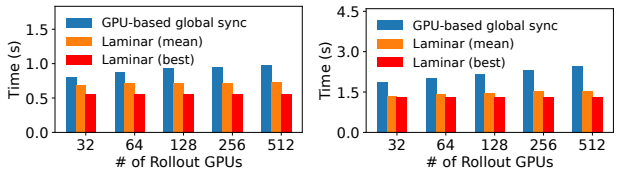
Overall performance. We observe that Laminar consistently outperforms the baselines across all model and cluster scales. In Figure 11, Laminar achieves an average speedup of 2.56× (up to 5.49×) over verl, 1.98× (up to 4.09×) over one-step staleness pipeline, 1.93× (up to 4.06×) over stream generation, and 1.39× (up to 1.81×) over AReaL. Similar gains are achieved on the tool calling task (Figure 12), where Laminar delivers an average speedup of 2.62× across all baselines. The performance advantage of Laminar increases as we scale to more GPUs, stemming from two primary factors.

First, Laminar adopts trajectory-level asynchrony, allowing each rollout to proceed at its own pace and enhancing generation throughput. In contrast, baselines are constrained by a global synchronization barrier, which forces faster rollouts to wait for the slowest ones, creating significant pipeline bubbles. While AReaL attempts to mitigate this with partial rollouts, it introduces overhead due to trajectory generation aborting/resumption and KVCache recomputation, which adversely affects generation throughput. *Second*, by minimizing rollout bubbles, Laminar sustains a higher generation throughput. This efficiency allows allocating more GPUs to the trainer within a given cluster as compared to other asynchronous baselines, while still balancing generation and training throughput.

Scalability. Figures 11, 12 show that Laminar achieves better scalability than all baselines, a strong scaling efficiency of 53.7% (up to 68.2% on 32B model) on math tasks and 46.5% on tool-calling tasks. The scaling efficciency is computed by dividing $\frac{\text{throughput in largest scale}}{\text{throughput in smallest scale}}$ by $\frac{\text{max. \# of GPUs}}{\text{min. \# of GPUs}}$ [81]. The best baselines only reach 33.6% (up to 39.2%) (AReaL on math) and 12.9% (stream generation on tool-calling). The enhanced scalability of Laminar stems from its efficient resource utilization across all scales. As the cluster scales up, Laminar effectively utilizes additional GPUs to boost training throughput with little bubbles, as shown in Figure 3(e). By avoiding straggler effects from global synchronization, Laminar exhibits increasing speedups as training scales. This results in an average speedup of 3.34× at the largest cluster scales across all baselines and model sizes. A more detailed analysis of this scalability is in Appendix B.

8.2 Training Convergence

Model convergence speed is crucial in RL post-training, measured by the training reward improvement over time. We compare Laminar with baselines using their respective tuned or open-sourced hyperparameters for the GRPO algorithm, with staleness bound under 1 for the stream generation baseline. AReaL utilizes its proposed Decoupled PPO [82] algorithm to mitigate errors introduced by partial rollout with a suggested optimal staleness of 4. Laminar's maximum staleness is 4 for a direct comparison (detailed settings in Appendix A.2). As shown in Figure 13, Laminar converges about 1.77× faster for 7B and 1.59× faster for 32B than the best baseline (on-policy verl) on the math reasoning task. Laminar achieves this by significantly increasing the training throughput while minimizing staleness without introducing additional biases; however, in other asynchronous systems, the throughput improvement is outweighed by staleness or additional biases, such as mixing multiple policy versions within a single trajectory in AReaL.



(a) 32B (b) 72B

Figure 14. Rollout waiting time during weight syncing. Trainer has the same number of GPUs as all rollouts' GPUs.

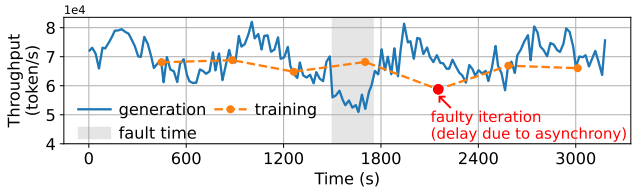


Figure 15. Training with a rollout machine failure.

8.3 Weight Synchronization Overhead

We evaluate the weight synchronization overhead of our relay worker design by measuring rollout waiting time and actor stalling time. We compare Laminar against the GPU-based global synchronization approach adopted by recent RL systems [12, 16, 17, 19, 22], that each actor model partition is broadcast to the corresponding model shards on each rollout. It follows the mapping between actor and rollout parallel groups and uses NCCL to achieve zero-copy transfer [16, 19].

Figure 14 compares the rollout waiting time, measured from when each rollout begins updating to the latest weights until the update is complete. From 64 to 1024 GPUs, Laminar consistently outperforms the GPU-based global synchronization. It reduces the average and best-case waiting times by up to 37% and 47%, respectively. Laminar's best-case occurs when the latest weights are already cached on the relay worker's CPU memory, allowing for loading model shards in parallel over the fast PCIe bus. Notably, Laminar's average rollout waiting time remains close to its best-case, due to our RDMA-based relay broadcast (Figure 18 in Appendix D) and trajectory-level asynchrony. By eliminating global synchronization points, few rollouts request the latest weights exactly when the actor finishes its update. Consequently, most rollouts can fetch weights locally over PCIe without waiting for a network broadcast.

Laminar also minimizes actor stalling time thanks to our hierarchical relay worker design. The actor only transfers weights to a single master relay, and its communication overhead remains constant regardless of the number of rollout replicas. Therefore, the actor stalls only 0.64 and 1.40 seconds for 32B and 72B models, respectively.

8.4 Repack Efficiency

We validate the benefit of our repack mechanism in eliminating remaining pipeline bubbles during generation. In this experiment, we use the same placement setting as the end-to-end experiment on the 32B model with 128 GPUs (§8.1): 64 GPUs for the trainer and 64 GPUs for rollouts, with each rollout occupying 4 GPUs (totaling 16 rollouts).

Table 1. Statistics of the rollouts with and without repack.

Laminar	Average/max latency of generating trajectories (s)	Repack overhead (s)	Average KVCache util.
w/ repack	290/828	0.69	82.2%
w/o repack	296/826	/	71.6%

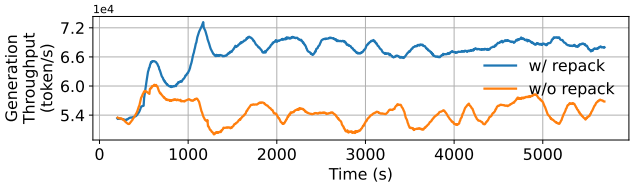


Figure 16. Repack efficiency.

As shown in Figure 16, enabling the repack mechanism boosts generation throughput by 26% than without. Laminar detects underutilized rollouts by monitoring their GPU KVCache utilization. As shown in Table 1, the repack mechanism increases average KVCache utilization by 14.8%. This improvement is achieved with a negligible repacking overhead of 0.69s and, crucially, does not increase the latency of trajectories generation.

8.5 Fault Tolerance Analysis

We evaluate the robustness of Laminar when encountering failures during RL training. We manually kill a rollout machine containing two replicas during a training job that has the same setting as in §8.4. As shown in Figure 15, this event causes an immediate drop in generation throughput due to the loss of rollouts. Training throughput remains unaffected or experiences a slight drop while waiting for adequate trajectories from remaining healthy rollouts. The system recovers in approximately 252 seconds, including allocating a new machine and initializing the rollouts on it. After recovery, both generation throughput and training throughput recover to original levels. This demonstrates that Laminar handles machine failures without disrupting training progress.

9 Related Work

RL frameworks for LLMs. Early reinforcement learning (RL) frameworks for LLMs primarily focused on synchronous algorithms [1, 8, 11, 12, 14–16, 45, 46, 83–87]. More recent work has begun to explore asynchronous RL [10, 17, 19–22]. However, these emerging asynchronous systems still depend on a global weight synchronization point. This design forces faster rollouts to idle while waiting for stragglers, creating significant long-tail performance bubbles. Some systems attempt to mitigate this issue with partial rollouts [7, 8, 18, 19]. But this technique introduces severe penalties: the high overhead of recomputing KVCache after each update and the undesirable mixing of multiple policy versions within

a single trajectory. These issues are notably exacerbated at scale. In contrast, Laminar's fully decoupled architecture and trajectory-level asynchrony eliminates these long-tail bubbles while guaranteeing that each trajectory is generated with a single, consistent policy version.

Asynchronous deep RL systems. Extensive research exists on asynchronous deep RL systems [24, 44, 88–94]. However, they target small-scale DNNs, not designed for the efficient inference, generation, and training patterns of modern LLMs. Some systems employ fully asynchronous parameter updates where gradients are computed from stale parameters [95–97]. This approach can introduce training instability and is thus rarely used for LLMs. The success of state-of-the-art deep RL systems like AlphaStar [44] and OpenAI Five [66] has been attributed to massive scale-out, with OpenAI Five, for instance, leveraging 57,200 rollout workers on 51,200 CPUs for 10 months [24]. This emphasis on scale has also spurred optimizations in related components, such as distributed experience buffers [98–100].

Fault-tolerant training. Previous studies [101–106] have introduced robust AI infrastructures to automatically detect and localize various training failures through real-time monitoring and stop-time diagnosis. The sophisticated techniques they rely on, such as RDMA-level metric collection and `dmesg` inspection, can be paired with our heartbeat-based failover mechanism to improve observability of rollout replicas and trainer workers in the RL system further. Many diagnostic tools have been developed to pinpoint network faults, both inter-host [107–109] and intra-host [110]. Rather than delving deep into the network stack to expose the exact root cause, our broadcast chain rebuilding mechanism works around communication failures by routing around faulty relay workers during weight synchronization. Checkpointing optimizations [60, 111–113] further mitigate stalls and support flexible transfer [59, 60]. By incorporating these methods, our trainer can recover from failures more quickly.

10 Conclusion

Laminar is an RL framework that addresses the limited scalability and long-tail trajectory generation problem in LLM post-training. We enable trajectory-level asynchrony, with each trajectory generated and consumed independently at its own optimal pace, eliminating rigid global synchronization. This design is achieved through a fully decoupled architecture with relay workers enabling asynchronous weight synchronization, allowing rollouts to pull new model parameters anytime without stalling computation. This architecture accommodates evolving trajectory lengths in RL training, while isolating component failures to ensure robust fault tolerance for long-running jobs. Our dynamic repack mechanism further consolidates long-tail trajectories to maximize generation throughput while minimizing staleness. We conducted extensive evaluation at scales up to 1024 GPUs. Our results demonstrate that Laminar achieves up to 5.48× speedup over state-of-the-art systems and reduces model convergence time.

References

[1] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[2] 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. 2025. DAPO: An Open-Source LLM Reinforcement Learning System at Scale. arXiv:2503.14476 [cs.LG] https://arxiv.org/abs/2503.14476

[3] 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. 2025. Llm post-training: A deep dive into reasoning large language models. *arXiv preprint arXiv:2502.21321* (2025).

[4] OpenAI. 2025. Learning to reason with llms. https://openai.com/index/learning-to-reason-with-llms/.

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

[6] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv preprint arXiv: 2310.06770* (2023).

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

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

[9] Zhong-Zhi Li, Duzhen Zhang, Ming-Liang Zhang, Jiaxin Zhang, Zengyan Liu, Yuxuan Yao, Haotian Xu, Junhao Zheng, Pei-Jie Wang, Xiuyi Chen, Yingying Zhang, Fei Yin, Jiahua Dong, Zhiwei Li, Bao-Long Bi, Ling-Rui Mei, Junfeng Fang, Xiao Liang, Zhijiang Guo, Le Song, and Cheng-Lin Liu. 2025. From System 1 to System 2: A Survey of Reasoning Large Language Models. *arXiv preprint arXiv: 2502.17419* (2025).

[10] Taiyi Wang, Zhihao Wu, Jianheng Liu, Jianye Hao, Jun Wang, and Kun Shao. 2025. DistRL: An Asynchronous Distributed Reinforcement Learning Framework for On-Device Control Agents. arXiv:2410.14803 [cs.LG] https://arxiv.org/abs/2410.14803

[11] 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. 2023. DeepSpeed-Chat: Easy, Fast and Affordable RLHF Training of ChatGPT-like Models at All Scales. arXiv:2308.01320 [cs.LG] https://arxiv.org/abs/2308.01320

[12] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. HybridFlow: A Flexible and Efficient RLHF Framework. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 1279–1297. doi:10.1145/3689031.3696075

[13] Kinman Lei, Yuyang Jin, Mingshu Zhai, Kezhao Huang, Haoxing Ye, and Jidong Zhai. 2024. PUZZLE: Efficiently Aligning Large Language Models through Light-Weight Context Switch. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 127–140. https://www.usenix.org/conference/atc24/presentation/lei

[14] Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. 2024. ReaL: Efficient RLHF Training of Large Language Models with Parameter Reallocation. *arXiv preprint arXiv: 2406.14088* (2024).

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

[16] Jian Hu, Xibin Wu, Zilin Zhu, Xianyu, Weixun Wang, Dehao Zhang, and Yu Cao. 2024. OpenRLHF: An Easy-to-use, Scalable and High-performance RLHF Framework. arXiv:2405.11143 [cs.AI] https://arxiv.org/abs/2405.11143

[17] 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. 2025. StreamRL: Scalable, Heterogeneous, and Elastic RL for LLMs with Disaggregated Stream Generation. arXiv:2504.15930 [cs] doi:10.48550/arXiv.2504.15930

[18] 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. 2025. AReaL: A Large-Scale Asynchronous Reinforcement Learning System for Language Reasoning. arXiv:2505.24298 [cs] doi:10.48550/arXiv.2505.24298

[19] 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. 2025. LlamaRL: A Distributed Asynchronous Reinforcement Learning Framework for Efficient Large-scale LLM Training. arXiv:2505.24034 [cs] doi:10.48550/arXiv.2505.24034

[20] Sijun Tan, Michael Luo, Colin Cai, Tarun Venkat, Kyle Montgomery, Aaron Hao, Tianhao Wu, Arnav Balyan, Manan Roongta, Chenguang Wang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. 2025. rLLM: A Framework for Post-Training Language Agents. https://pretty-radio-b75.notion.site/rLLM-A-Framework-for-Post-Training-Language-Agents-21b81902c146819db63cd98a54ba5f31. Notion Blog.

[21] Prime Intellect Team, Sami Jaghouar, Justus Mattern, Jack Min Ong, Jannik Straube, Manveer Basra, Aaron Pazdera, Kushal Thaman, Matthew Di Ferrante, Felix Gabriel, Fares Obeid, Kemal Erdem, Michael Keiblinger, and Johannes Hagemann. 2025. INTELLECT-2: A Reasoning Model Trained Through Globally Decentralized Reinforcement Learning. arXiv:2505.07291 [cs.LG] https://arxiv.org/abs/2505.07291

[22] 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. 2025. AsyncFlow: An Asynchronous Streaming RL Framework for Efficient LLM Post-Training. arXiv:2507.01663 [cs.LG] https://arxiv.org/abs/2507.01663

[23] Michael Noukhovitch, Shengyi Huang, Rishabh Agarwal, Aaron Courville, Sophie Xhonneux, and Arian Hosseini. 2025. ASYNCHRONOUS RLHF: FASTER AND MORE EFFICIENT OFF-POLICY RL FOR LANGUAGE MODELS. (2025).

[24] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. 2019. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680* (2019).

[25] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. 2018. Pipe-SGD: A decentralized pipelined SGD framework for distributed deep net training. *Advances in Neural Information Processing Systems* 31 (2018).

[26] Saar Barkai, Ido Hakimi, and Assaf Schuster. 2019. Gap aware mitigation of gradient staleness. *arXiv preprint arXiv:1909.10802* (2019).

[27] Yangrui Chen, Cong Xie, Meng Ma, Juncheng Gu, Yanghua Peng, Haibin Lin, Chuan Wu, and Yibo Zhu. 2022. SAPipe: Staleness-Aware Pipeline for Data Parallel DNN Training. In *Advances in Neural Information Processing Systems*.

[28] Guangming Sheng, Junwei Su, Chao Huang, and Chuan Wu. 2024. Mspipe: Efficient temporal gnn training via staleness-aware pipeline. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2651–2662.

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

[30] Yao Cheng, Jianfeng Chen, Jie Chen, Li Chen, Liyu Chen, and et al. 2025. FullStack Bench: Evaluating LLMs as Full Stack Coders. arXiv:2412.00535 [cs.AI] https://arxiv.org/abs/2412.00535

[31] OpenAI. 2025. Introducing deep research. https://openai.com/index/introducing-deep-research/.

[32] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

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

[34] Josef Dai, Xuehai Pan, Ruiyang Sun, Jiaming Ji, Xinbo Xu, Mickel Liu, Yizhou Wang, and Yaodong Yang. 2024. Safe RLHF: Safe Reinforcement Learning from Human Feedback. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=TyFrPOKYXw

[35] Rui Zheng, Wei Shen, Yuan Hua, Wenbin Lai, Shihan Dou, Yuhao Zhou, Zhiheng Xi, Xiao Wang, Haoran Huang, Tao Gui, et al. 2023. Improving generalization of alignment with human preferences through group invariant learning. *arXiv preprint arXiv:2310.11971* (2023).

[36] Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Lu, Thomas Mesnard, Colton Bishop, Victor Carbune, and Abhinav Rastogi. 2023. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267* (2023).

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

[38] Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. 2025. ReTool: Reinforcement Learning for Strategic Tool Use in LLMs. *arXiv preprint arXiv: 2504.11536* (2025).

[39] Mathematical Association of America. 2024. AIME 2024.

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

[41] Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. 2024. Back to Basics: Revisiting REINFORCE Style Optimization for Learning from Human Feedback in LLMs. arXiv:2402.14740 [cs.LG] https://arxiv.org/abs/2402.14740

[42] Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2023. GAIA: a benchmark for General AI Assistants. arXiv:2311.12983 [cs.CL] https://arxiv.org/abs/2311.12983

[43] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. arXiv:2404.07972 [cs.AI] https://arxiv.org/abs/2404.07972

[44] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *nature* 575, 7782 (2019), 350–354.

[45] Gerald Shen, Zhilin Wang, Olivier Delalleau, Jiaqi Zeng, Yi Dong, Daniel Egert, Shengyang Sun, Jimmy Zhang, Sahil Jain, Ali Taghibakhshi, Markel Sanz Ausin, Ashwath Aithal, and Oleksii Kuchaiev. 2024. NeMo-Aligner: Scalable Toolkit for Efficient Model Alignment. arXiv:2405.01481 [cs.CL] https://arxiv.org/abs/2405.01481

[46] Youshao Xiao, Zhenglei Zhou, Fagui Mao, Weichang Wu, Shangchun Zhao, Lin Ju, Lei Liang, Xiaolu Zhang, and Jun Zhou. 2023. An Adaptive Placement and Parallelism Framework for Accelerating RLHF Training. *arXiv preprint arXiv: 2312.11819* (2023).

[47] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. 2017. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 3389–3396.

[48] Gregory Kahn, Adam Villaflor, Pieter Abbeel, and Sergey Levine. 2018. Composable action-conditioned predictors: Flexible off-policy learning for robot navigation. In *Conference on robot learning*. PMLR, 806–816.

[49] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. 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)*. 155–170.

[50] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[51] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, et al. 2025. {NanoFlow}: Towards Optimal Large Language Model Serving Throughput. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 749–765.

[52] 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. 2023. Efficiently Programming Large Language Models using SGLang. *arXiv preprint arXiv: 2312.07104* (2023).

[53] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 117–134.

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

[55] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, et al. 2025. MegaScale-Infer: Serving Mixture-of-Experts at Scale with Disaggregated Expert Parallelism. *arXiv preprint arXiv:2504.02263* (2025).

[56] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. 2025. Kimi K2: Open Agentic Intelligence. *arXiv preprint arXiv:2507.20534* (2025).

[57] 2023. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl

[58] Zhiyi Hu, Siyuan Shen, Tommaso Bonato, Sylvain Jeaugey, Cedell Alexander, Eric Spada, James Dinan, Jeff Hammond, and Torsten Hoefler. 2025. Demystifying NCCL: An In-depth Analysis of GPU Communication Protocols and Algorithms. arXiv:2507.04786 [cs.DC] https://arxiv.org/abs/2507.04786

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

[60] 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. 2025. ByteCheckpoint: A Unified Checkpointing System for Large Foundation Model Development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 559–578. https://www.usenix.org/conference/nsdi25/presentation/wan-borui

[61] Tianle Sun Yingyi Hao Rong Chen Mingcong Han Jinyu Gu Xingda Wei, Zhuobin Huang and Haibo Chen. 2025. PhoenixOS: Concurrent OS-level GPU Checkpoint and Restore with Validated Speculation. In *Proceedings of the ACM SIGOPS 31th Symposium on Operating Systems Principles*.

[62] 2023. SageMaker. https://docs.aws.amazon.com/sagemaker/index.html.

[63] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. 2024. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 709–729.

[64] Zhiyu Mei, Wei Fu, Guang Wang, Huanchen Zhang, and Yi Wu. 2023. SRL: Scaling Distributed Reinforcement Learning to Over Ten Thousand Cores. *International Conference on Learning Representations* (2023). doi:10.48550/arXiv.2306.16688

[65] Alexandros Batsakis, Randal Burns, Arkady Kanevsky, James Lentini, and Thomas Talpey. 2009. Ca-nfs: A congestion-aware network file system. *ACM Transactions on Storage (TOS)* 5, 4 (2009), 1–24.

[66] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv preprint arXiv: 1912.06680* (2019).

[67] Sanfilippo S. 2009. "Redis In-memory Data Structure Server". https://redis.io/

[68] Michael Ikkert, Tim Kieritz, and Peter Sanders. 2009. *Parallele Algorithmen*. Technical Report. Karlsruher Institut für Technologie (KIT). https://ae.iti.kit.edu/documents/teaching/parallele-algorithmen/ws18/skript.pdf Stand: 16. Oktober 2009.

[69] David S Johnson. 1974. Fast algorithms for bin packing. *J. Comput. System Sci.* 8, 3 (1974), 272–314.

[70] Andrew Chi-Chih Yao. 1980. New algorithms for bin packing. *Journal of the ACM (JACM)* 27, 2 (1980), 207–227.

[71] David S. Johnson, Alan Demers, Jeffrey D. Ullman, Michael R Garey, and Ronald L. Graham. 1974. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on computing*

3, 4 (1974), 299–325.

[72] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[73] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, Yan Yan, Beidi Chen, Guangyu Sun, and Kurt Keutzer. 2024. LLM Inference Unveiled: Survey and Roofline Model Insights. arXiv:2402.16363 [cs.CL] https://arxiv.org/abs/2402.16363

[74] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized Experience Replay. arXiv:1511.05952

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

[76] 2019. Unified Communication X. https://github.com/openucx/ucx

[77] 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. 2024. Qwen2.5 Technical Report. *arXiv preprint arXiv: 2412.15115* (2024).

[78] Yanli Zhao, A. Gu, R. Varma, Liangchen Luo, Chien chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Bernard Nguyen, Geeta Chauhan, Y. Hao, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proceedings of the VLDB Endowment* (2023). doi:10.14778/3611540.3611569

[79] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. 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*. 1–15.

[80] Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Xuanhe Zhou, Yufei Huang, Chaojun Xiao, et al. 2024. Tool learning with foundation models. *Comput. Surveys* 57, 4 (2024), 1–40.

[81] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*. 483–485.

[82] Jacob Hilton, Karl Cobbe, and John Schulman. 2022. Batch Size-Invariance for Policy Optimization. arXiv:2110.00641 doi:10.48550/arXiv.2110.00641

[83] Collosal-AI Corporation. 2023. *Collosal-Chat.* https://github.com/binmakeswell/ColossalChat

[84] Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. 2020. TRL: Transformer Reinforcement Learning. https://github.com/huggingface/trl.

[85] 2025. NeMo RL: A Scalable and Efficient Post-Training Library. https://github.com/NVIDIA-NeMo/RL. GitHub repository.

[86] Tyler Griggs, Sumanth Hegde, Eric Tang, Shu Liu, Shiyi Cao, Dacheng Li, Charlie Ruan, Philipp Moritz, Kourosh Hakhamaneshi, Richard Liaw, Akshay Malik, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Evolving SkyRL into a Highly-Modular RL Framework. Notion Blog.

[87] Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. 2025. slime: An LLM post-training framework for RL Scaling. https://github.com/THUDM/slime. GitHub repository. Corresponding author: Xin Lv.

[88] C. Hesse, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. 2017. *OpenAI baselines*. https://github.com/openai/baselines

[89] Danijar Hafner, James Davidson, and Vincent Vanhoucke. 2017. Tensorflow agents: Efficient batched reinforcement learning in tensorflow. *arXiv preprint arXiv:1709.02878* (2017).

[90] Albert Bou, Matteo Bettini, Sebastian Dittert, Vikash Kumar, Shagun Sodhani, Xiaomeng Yang, Gianni De Fabritiis, and Vincent Moens. 2023. TorchRL: A data-driven decision-making library for PyTorch. arXiv:2306.00577 [cs.LG] https://arxiv.org/abs/2306.00577

[91] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. 2018. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. arXiv:1802.01561 [cs] doi:10.48550/arXiv.1802.01561

[92] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2018. RLlib: Abstractions for Distributed Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 3053–3062. https://proceedings.mlr.press/v80/liang18b.html

[93] Eric Liang, Zhanghao Wu, Michael Luo, Sven Mika, Joseph E Gonzalez, and Ion Stoica. 2021. RLlib Flow: Distributed Reinforcement Learning is a Dataflow Problem. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 5506–5517. https://proceedings.neurips.cc/paper_files/paper/2021/file/2bce32ed409f5ebcee2a7b417ad9beed-Paper.pdf

[94] Jiayi Weng, Huayu Chen, Dong Yan, Kaichao You, Alexis Duburcq, Minghao Zhang, Yi Su, Hang Su, and Jun Zhu. 2022. Tianshou: A Highly Modularized Deep Reinforcement Learning Library. *Journal of Machine Learning Research* 23, 267 (2022), 1–6. http://jmlr.org/papers/v23/21-1127.html

[95] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems*, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger (Eds.), Vol. 24. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2011/file/218a0aefd1d1a4be65601cc6ddc1520e-Paper.pdf

[96] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-Level Control through Deep Reinforcement Learning. *Nature* 518, 7540 (Feb. 2015), 529–533. doi:10.1038/nature14236

[97] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. (June 2016). arXiv:1602.01783 [cs.LG]

[98] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. 2018. Distributed Prioritized Experience Replay. arXiv:1803.00933 [cs]

[99] Albin Cassirer, Gabriel Barth-Maron, Eugene Brevdo, Sabela Ramos, Toby Boyd, Thibault Sottiaux, and Manuel Kroiss. 2021. Reverb: A Framework For Experience Replay. arXiv:2102.04736 [cs.LG]

[100] Hanjing Wang, Man-Kit Sit, Congjie He, Ying Wen, Weinan Zhang, Jun Wang, Yaodong Yang, and Luo Mai. 2023. GEAR: A GPU-Centric Experience Replay System for Large Reinforcement Learning Models. In *Proceedings of the 40th International Conference on Machine Learning*. PMLR, 36380–36390.

[101] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, Steve Lacy, Hang Wang, Aaron Wisner, Chris Lewis, and Henri Bahini. 2024. Resiliency at Scale: Managing Google's TPUv4 Machine Learning Supercomputer. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 761–774. https://www.usenix.org/conference/nsdi24/presentation/zu

[102] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. 2024. Characterization of Large Language Model Development in the Datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 709–729. https://www.usenix.org/conference/nsdi24/presentation/hu

[103] 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. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 745–760. https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng

[104] Jianbo Dong, Kun Qian, Pengcheng Zhang, Zhilong Zheng, Liang Chen, Fei Feng, Yichi Xu, Yikai Zhu, Gang Lu, Xue Li, Zhihui Ren, Zhicheng Wang, Bin Luo, Peng Zhang, Yang Liu, Yanqing Chen, Yu Guan, Weicheng Wang, Chaojie Yang, Yang Zhang, Man Yuan, Hanyu Zhao, Yong Li, Zihan Zhao, Shan Li, Xianlong Zeng, Zhiping Yao, Binzhang Fu, Ennan Zhai, Wei Lin, Chao Wang, and Dennis Cai. 2025. Evolution of Aegis: Fault Diagnosis for AI Model Training Service in Production. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 865–881. https://www.usenix.org/conference/nsdi25/presentation/dong

[105] Yangtao Deng, Lei Zhang, Qinlong Wang, Xiaoyun Zhi, Xinlei Zhang, Zhuo Jiang, Haohan Xu, Lei Wang, Zuquan Song, Gaohong Liu, et al. 2025. Mycroft: Tracing Dependencies in Collective Communication Towards Reliable LLM Training. *arXiv preprint arXiv:2509.03018* (2025).

[106] 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. 2025. Robust LLM Training Infrastructure at ByteDance. arXiv:2509.16293 [cs.LG] https://arxiv.org/abs/2509.16293

[107] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. 2015. Packet-Level Telemetry in Large Datacenter Networks. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 479–491. doi:10.1145/2829988.2787483

[108] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies* (Irvine, California, USA) *(CoNEXT '16)*. Association for Computing Machinery, New York, NY, USA, 481–495. doi:10.1145/2999572.2999609

[109] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. 2019. NetBouncer: Active Device and Link Failure Localization in Data Center Networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 599–614. https://www.usenix.org/conference/nsdi19/presentation/tan

[110] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. 2023. Hostping: Diagnosing Intra-host Network Bottlenecks in RDMA Servers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 15–29. https://www.usenix.org/conference/nsdi23/presentation/liu-kefei

[111] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 203–216. https://www.usenix.org/conference/fast21/presentation/mohan

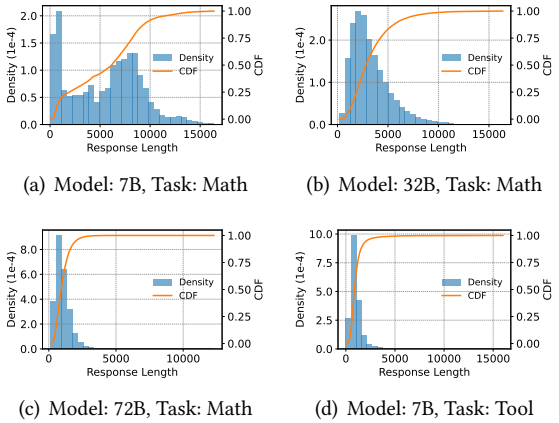
[112] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 929–943. https://www.usenix.org/conference/nsdi22/presentation/eisenman

[113] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 364–381. doi:10.1145/3600006.3613145

A Experiment Details

A.1 Response Length Distribution of Each Model

The response length distributions of each model used in the throughput experiments are listed in Figure 17.



(a) Model: 7B, Task: Math (b) Model: 32B, Task: Math

(c) Model: 72B, Task: Math (d) Model: 7B, Task: Tool

Figure 17. The response length distributions on the DAPO-Math-17k dataset of the model checkpoints used in the throughput experiments.

A.2 Hyperparameter Details

Throughput experiments. We evaluate system throughput on two tasks: math and tool-calling. For the math task, we use three intermediate checkpoints of different scales during the RL training from `Qwen2.5-Math-7B`, `Qwen2.5-32B`, and `Qwen2.5-Math-72B`, respectively. For the tool-calling task, we use the 7B checkpoint for the RL training in the open-source implementation of ReTool. Table 2 details the GPU allocation for each experiment, which we tuned to maximize throughput in our environment.

During rollout, we use temperature sampling with $t = 1$. We do not use top-P or top-k sampling. For verl, One-step Staleness, and Stream Generation, the global rollout batch size is 8192, matching the training batch size. For AReaL and Laminar, we set the maximum concurrency to 1024 trajectories per rollout replica. The rollout tensor parallelism (TP) sizes are set to 4 and 8 for the 32B and 72B models, respectively. For the 7B model, the size varies according to the system's throughput characteristics. In AReaL and Laminar, we set the TP size to 1 to maximize throughput. In verl, One-step Staleness, and Stream Generation, we set the TP size to 2, which balances generation throughput against latency, mitigating the long-tail bottlenecks.

For AReaL, we use Megatron-LM's hybrid of data (DP), tensor (TP), and pipeline parallelism (PP). The TP and PP sizes are set to 2 and 1 for the 7B model, 4 and 2 for the 32B model, and 4 and 4 for the 72B model. The DP size is adaptively set to $\frac{\#\text{ of Train GPUs}}{\text{TP Size}\times\text{PP Size}}$. For other systems, we combine Torch DDP, FSDP, and Ulysses Sequence Parallelism (SP). The FSDP and SP sizes are set to 8 and 4 for the 7B model, 16 and 8 for the 32B model, and 32 and 8 for the 72B model. The DDP size is set to $\frac{\#\text{ of Train GPUs}}{\text{FSDP Size}}$.

Convergence experiments. We conduct convergence experiments using two base models: `Qwen2.5-Math-7B` and `Qwen2.5-32B`. The GPU allocation is identical to that of the throughput experiments. Table 3 lists the other hyperparameters. For AReaL and Laminar, we reduce the maximum per-rollout concurrency to 256. We also adopt the default FIFO sampling strategy for these two systems.

B Detailed Experiment Analysis

Performance in small cluster scales. In relatively small cluster scales, Laminar provides a moderate but consistent 1.41× speedup on average across all the baselines and model sizes on at most 64 GPUs, as shown in Figure 11. The performance in these configurations is constrained by the fact that both training and rollout of LLMs require a number of GPUs to accommodate the model via parallelism. This severely limits the possible resource allocation choices, forcing different systems with similar placements, as shown in Table 2. Furthermore, these placements are usually suboptimal since the training and rollout throughput can not be effectively balanced. Within such limited configurations, Laminar improves the throughput mainly by eliminating long-tail bubbles.

Performance in large-scale clusters. The performance advantage of Laminar becomes much more pronounced in large-scale clusters, achieving an average speedup of 3.34× at the largest cluster scales across all the baselines and model sizes, as is shown in Figure 11. Systems like verl, one-step staleness, and stream generation are heavily bottlenecked by the global weight synchronization. With the end-to-end latency of the generation stage bound by the long-tail trajectories, adding more rollout resources provides marginal returns on throughput. AReaL mitigates this bottleneck by applying partial rollout, but its scalability is then limited by the KVCache recomputation overhead. As the cluster size scales up, this overhead worsens as the number of trajectories and the model update frequency increase, capping the system's performance. In contrast, Laminar effectively resolves the global weight synchronization constraint and eliminates the long-tail bubbles while introducing minimal overhead through the relay design and repack mechanism. Furthermore, as the cluster size scales up, the repacking becomes increasingly effective with more rollout replicas to manage.

C Discussion

Partial rollout. Laminar's trajectory-level asynchrony effectively mitigates long-tail latency bubbles by decoupling trajectory generation from global synchronization. Recently,

Table 2. GPU allocation of the systems in the throughput experiments. For verl, we adopt the colocated allocation which uses all the GPUs for train and rollout alternately.

System	7B			32B			72B		
	Total	**Train**	**Rollout**	**Total**	**Train**	**Rollout**	**Total**	**Train**	**Rollout**
verl	16	Colocated		32	Colocated		64	Colocated	
	32			64			128		
	64			128			256		
	128			256			512		
	256			512			1024		
One-step Staleness	16	8	8	32	16	16	64	32	32
	32	8	24	64	32	32	128	64	64
	64	16	48	128	48	80	256	96	160
	128	32	96	256	64	192	512	192	320
	256	40	216	512	80	432	1024	256	768
Stream Generation	16	8	8	32	16	16	64	32	32
	32	8	24	64	32	32	128	64	64
	64	16	48	128	48	80	256	96	160
	128	32	96	256	64	192	512	192	320
	256	40	216	512	80	432	1024	256	768
AReaL	16	8	8	32	16	16	64	32	32
	32	16	16	64	32	32	128	64	64
	64	32	32	128	64	64	256	128	128
	128	64	64	256	128	128	512	320	192
	256	128	128	512	256	256	1024	640	384
Laminar	16	8	8	32	16	16	64	32	32
	32	24	8	64	32	32	128	64	64
	64	40	24	128	64	64	256	128	128
	128	80	48	256	128	128	512	320	192
	256	192	64	512	256	256	1024	768	256

an orthogonal approach has emerged that updates weights mid-generation to reduce these bubbles known as partial rollout systems [7, 8, 18, 19]. These systems interrupt all ongoing generation and update rollout replicas once the latest actor weights become available. The aborted trajectories then continue generating with the new weights. This technique can be integrated with synchronous, k-step staleness, and Laminar to further reduce idle time. However, it introduces a challenge to training stability. Updating weights amid generation creates mixed-version trajectories, where a single long trajectory is produced using several successive weight versions. The expected number of versions grows linearly with sequence length and the actor-update frequency. These mixed-version trajectories distort the natural output-length distribution. They can also hinder convergence in valueless RL algorithms, which rely on forming trajectory groups from a consistent weight version. As update frequency rises, the contamination worsens, creating an undesirable coupling between system throughput and training stability. As shown in Figure 13, applying partial rollout can converge at a slower speed, requiring more algorithmic research to stabilize such training.

Effective experiences sampling. In Laminar, as scaling out RL tasks, the scaling of trajectory generation allows rollout throughput to outpace the trainer's consumption speed (analyzed in §8.1). This creates an abundance of diverse experiences, but presents a key challenge: *how to utilize these experiences effectively*. Experiences sampling is a classic problem in traditional RL. For instance, OpenAI's work on Dota 2 highlighted the negative impact of data staleness on training speed [24]. Recent research advocates for priority-based sampling [10, 74, 91], which considers metrics like TD errors, importance sampling, and entropy to prioritize the most informative transitions.

However, experience sampling remains an underexplored area in large-scale LLM post-training. The ability of Laminar to efficiently generate massive experiences is a key strength, but this potential can only be fully realized with an effective

Table 3. Hyperparameters for convergence experiments. Configurations are based on the original AReaL paper and DAPO. For asynchronous systems (One-step Staleness, Stream Generation, and Laminar), we increase the mini-batch size to 2048 to stabilize training with off-policy data, aligning with AReaL's configuration.

System	verl	One-step Staleness	Stream Generation	AReaL	Laminar
Algorithm	GRPO	GRPO	GRPO	Decoupled PPO	GRPO
Learning Rate	1e-6	1e-6	1e-6	2e-5	1e-6
Weight Decay	0.1	0.1	0.1	0.05	0.1
Clip $\varepsilon_{\text{high}}$	0.28	0.28	0.28	0.2	0.28
Clip ε_{low}	0.2	0.2	0.2	0.2	0.2
Discount γ	1.0	1.0	1.0	1.0	1.0
GAE λ	1.0	1.0	1.0	1.0	1.0
Group Size	16	16	16	16	16
Training Global Batch Size	8192	8192	8192	8192	8192
Training Mini-Batch Size	512	2048	2048	2048	2048
Per-rollout Max Concurrency	N/A	N/A	N/A	256	256
Sampling Strategy	N/A	N/A	N/A	FIFO	FIFO
Max Staleness Bound	0	1	1	4	4 (observed)

sampling mechanism, which is crucial for translating diverse experiences and avoid learning on low-utility experiences. Thus, developing effective sampling strategies for utilizing generated experiences represents a promising area for future work in large-scale RL systems.

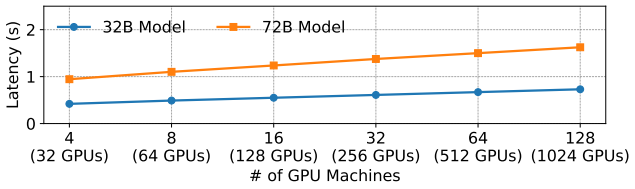


Figure 18. Relay broadcast latency with testbed in §8.

D Theoretical Analysis of Chain-Based Pipelined Broadcast

We provide a formal analysis of the latency for a chain-based pipelined broadcast. The goal is to show that for large messages, such as LLM weights, the total broadcast time is dominated by a term independent of the number of nodes, making the approach highly scalable.

D.1 Communication Model

We model the broadcast as a linear pipeline, where a master relay sends a message to $p-1$ other relays organized in a logical chain. Let the total number of nodes (master + relays) be p. The communication cost between any two adjacent nodes is defined by two parameters:

- T_{start}: The startup latency for sending a message, independent of its size. This includes costs like connection setup and initial processing.
- T_{byte}: The per-byte transmission time, determined by the network bandwidth ($T_{\text{byte}} = 1/\text{Bandwidth}$). The time t to transmit a single message of size s between two nodes is $t = s \cdot T_{\text{byte}} + T_{\text{start}}$.

In our pipelined approach, the total model weights, of size M bytes, are divided into k smaller chunks, each of size M/k.

D.2 Latency Derivation

The total time for the broadcast, $T(p,k)$, is the time from when the master sends the first chunk until the last relay receives the last chunk. This can be modeled as the sum of two components:

1. The time for the first chunk to travel down the entire chain of $p-1$ hops.
2. The time for the remaining $k-1$ chunks to be received by the final relay after it has received the first one.

The time to transmit one chunk between two nodes is $t_{\text{chunk}} = \frac{M}{k}T_{\text{byte}} + T_{\text{start}}$. The first chunk must traverse $p-1$ network hops to reach the final relay. Due to the pipelined nature, this takes $(p-1) \times t_{\text{chunk}}$. Once the first chunk arrives at the final relay, the subsequent $k-1$ chunks arrive sequentially, with one chunk arriving every t_{chunk}. This adds $(k-1) \times t_{\text{chunk}}$ to the total time. Therefore, the total latency is:

$$\begin{aligned} T(p,k) &= (p-1) \cdot t_{\text{chunk}} + (k-1) \cdot t_{\text{chunk}} \\ &= (p+k-2) \cdot t_{\text{chunk}} \\ &= (p+k-2)\left(\frac{M}{k}T_{\text{byte}} + T_{\text{start}}\right) \end{aligned}$$

Expanding this expression:

$$T(p,k) = MT_{\text{byte}} + (p-2)T_{\text{start}} + kT_{\text{start}} + \frac{(p-2)M}{k}T_{\text{byte}} \quad (1)$$

D.3 Scalability Analysis

To understand the scalability with respect to the number of nodes p, we analyze the terms in Equation 1. The choice of k (the number of chunks) is critical. We can find the optimal k that minimizes $T(p,k)$ by taking the derivative with respect to k and setting it to zero:

$$\frac{\partial T}{\partial k} = T_{\text{start}} - \frac{(p-2)M}{k^2}T_{\text{byte}} = 0$$

$$k^2 = \frac{(p-2)MT_{\text{byte}}}{T_{\text{start}}}$$

$$k^* = \sqrt{\frac{(p-2)MT_{\text{byte}}}{T_{\text{start}}}}$$

Substituting k^* back into Equation 1, we find the minimum possible broadcast time for a given p and M:

$$T^*(p) = MT_{\text{byte}} + (p-2)T_{\text{start}} + 2\sqrt{(p-2)MT_{\text{byte}}T_{\text{start}}}$$

Let's analyze the composition of this optimal time:

$$T^*(p) = \underbrace{MT_{\text{byte}}}_{\text{Bandwidth Term}} + \underbrace{(p-2)T_{\text{start}}}_{\text{Latency Term}} + \underbrace{2\sqrt{(p-2)MT_{\text{byte}}T_{\text{start}}}}_{\text{Pipeline Term}}$$

In the context of distributing large language models:

• **The message size M is very large** (e.g., 140 GB for a 70B model using BF16 precision).

• **The startup latency T_{start} is very small** for RDMA (on the order of microseconds). Consequently:

1. The **Bandwidth Term (MT_{byte})** is the time to serialize the entire model onto a single network link. Given the large M, this term is the dominant component of the total latency and is completely independent of p.
2. The **Latency Term ($(p-2)T_{\text{start}}$)** grows linearly with p. However, because T_{start} is extremely small, this term's contribution is negligible even for thousands of nodes (e.g., $2000 \times 5\mu s = 10ms$).
3. The **Pipeline Term** grows sub-linearly with the number of nodes ($O(\sqrt{p})$). While it grows faster than the latency term, its contribution remains significantly smaller than the bandwidth term for realistic system parameters.

Conclusion: The total broadcast time is overwhelmingly dominated by the constant bandwidth term (MT_{byte}). The terms dependent on the number of nodes p either have a very small coefficient (T_{start}) or grow sub-linearly ($O(\sqrt{p})$). This makes the total latency largely insensitive to the number of relays, proving that the chain-based pipelined broadcast is a highly scalable mechanism for distributing large model weights.