TensorHub: Scalable and Elastic Weight Transfer for LLM RL Training

Chenhao Ye[†], Huaizheng Zhang, Mingcong Han, Baoquan Zhong, Xiang Li, Qixiang Chen, Xinyi Zhang, Weidong Zhang, Kaihua Jiang, Wang Zhang, He Sun, Wencong Xiao[*], Andrea C. Arpaci-Dusseau[†*], Remzi H. Arpaci-Dusseau[†*]

ByteDance Seed [†]*University of Wisconsin–Madison*

Abstract

Modern LLM reinforcement learning (RL) workloads require a high-performance weight transfer system to scale training across heterogeneous compute resources. However, efficiently transferring terabyte-scale model weights across thousands of GPUs remains challenging because the system must accommodate clusters that dynamically scale up and down while keeping coordination, data movement, and storage overhead low.

We introduce Reference-Oriented Storage (ROS), a new storage abstraction for RL weight transfer that exploits highly replicated model weights in place. ROS presents the illusion that certain versions of the model weights are stored and can be fetched on demand. Underneath, ROS does not physically store any copies of the weights; instead, it tracks the workers that hold these weights on GPUs for inference. Upon request, ROS directly uses them to serve reads. We build TensorHub, a production-quality system that instantiates the ROS idea with topology-aware transfer, model-parallel consistency, and fault tolerance. Evaluation shows that TensorHub saturates RDMA bandwidth and adapts to three distinct rollout workloads with minimal engineering effort. Specifically, TensorHub reduces total GPU stall time by up to 6.7× for standalone rollouts, accelerates weight updates for elastic rollouts by up to 4.8×, and cuts cross-datacenter rollout stall time by up to 19×. TensorHub has been deployed in ByteDance production to support cutting-edge RL training.

CCS Concepts: • **Computer systems organization → Distributed architectures**; • **Computing methodologies →** *Machine learning*.

Keywords: LLM Training, Reinforcement Learning, Weight Transfer, Distributed Storage, RDMA

[*]Joint senior authors.

SOSP '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2585-2/2026/09
https://doi.org/10.1145/3830418.3843859

ACM Reference Format:
Chenhao Ye, Huaizheng Zhang, Mingcong Han, Baoquan Zhong, Xiang Li, Qixiang Chen, Xinyi Zhang, Weidong Zhang, Kaihua Jiang, Wang Zhang, He Sun, Wencong Xiao, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2026. TensorHub: Scalable and Elastic Weight Transfer for LLM RL Training. In *ACM SIGOPS 32nd Symposium on Operating Systems Principles (SOSP '26), September 29–October 02, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3830418.3843859

1 Introduction

Reinforcement learning (RL) is central to building state-of-the-art large language models (LLMs) [5, 10, 14, 35]. With carefully designed rewards, RL aligns LLM responses with human preferences [7, 23] and improves reasoning [5, 10, 29], coding [18, 38], and mathematical capabilities [28, 31, 42].

Weight transfer plays a critical role in RL training. RL operates as a continuous feedback loop between inference and training, with weight transfer closing the loop. In each step, *rollout* workers run inference to generate responses for reward scoring, and *trainer* workers use the resulting data to update the model weights. After each update, the new weights must be transferred to rollout workers for the next training step. In addition, weight transfer is required for failure recovery, as failures are common at large scale. Model weights are the heaviest state that a rollout worker must recover, and a restarted rollout worker cannot resume inference until it pulls the latest weights.

Production RL increasingly relies on diverse compute resources, posing new challenges for weight transfer. For example, the ByteDance production cluster has abundant spot GPU instances, but their frequent preemptions and restarts cause cluster membership churn, complicating their use. Multi-datacenter deployments provide access to geographically distributed compute capacity [24, 44], but slow inter-datacenter links make weight transfer a bottleneck.

Ideally, a weight transfer system should adapt to network topology and runtime load, accommodate cluster membership changes while masking failures, and avoid unnecessary control-flow coordination and data movement. It should also require little additional storage or memory.

However, existing weight transfer approaches fail to satisfy these goals. Collective communication (*e.g.*, NCCL [22])

delivers high throughput, but its reliance on static communication groups makes it brittle under failures and ill-suited for dynamic clusters with membership changes. Point-to-point communication (*e.g.*, UCX [30]) offers flexibility, yet suffers from network contention under fan-out. Moreover, both approaches couple trainer and rollout control flow, forcing the RL framework to coordinate workers during weight transfer.

Another weight transfer approach is through distributed storage (*e.g.*, parameter servers [15, 17], Ray object store [20]). Unlike communication-based approaches, distributed storage offers a clean decoupling between trainers and rollouts: trainers *push* new weights to storage, while rollouts *pull* them on demand, all with minimal coordination; rollouts may scale up and down without trainers' awareness. Yet, this push-then-pull pattern doubles data movement and incurs storage space or memory overhead, both of which are prohibitively expensive for TB-sized LLM weights.

We seek to retain the decoupling advantages of storage interfaces while eliminating the overhead of extra data movement and storage space. This is made possible by an important observation: each version of the model weights is *immutable* and *highly replicated* for data-parallel inference. Such redundancy naturally provides multiple equivalent data sources to serve read requests for that version.

Our key idea is to utilize highly replicated model weights *in place* and provide access to them efficiently via a storage interface, called **Reference-Oriented Storage** (ROS). ROS presents the illusion that certain versions of the model weights are stored and can be fetched on demand. Underneath, ROS does not physically store any copies of the weights; instead, it tracks the workers that hold these weights on GPUs for inference. Upon request, ROS directly uses them to serve reads. By avoiding explicit data ownership and extra copies, ROS combines the flexibility and simplicity of storage with the efficiency of direct transfer.

Building such a system is challenging due to *integrity* and *availability* requirements. ROS addresses these challenges with two novel mechanisms: a *mutability contract* and a *retention protocol*. The mutability contract disciplines the mutation semantics that enable safe weight buffer reuse. The retention protocol defines which versions must be kept; in a corner case where a desired version may be lost, it offloads a copy to CPU memory to fulfill availability requirements.

We build **TensorHub**, which instantiates the ROS idea in a production system with topology-aware transfer, model-parallel consistency, and fault tolerance. It schedules transfers with topology awareness and leverages pipeline replication to scale throughput. TensorHub enforces a consistent view among workers within each model-parallel group to ensure correctness. TensorHub proactively detects failures and orchestrates recovery, enabling transparent operation under dynamic cluster scaling.

We evaluate TensorHub on large-scale RL workloads spanning dozens to a thousand GPUs, elastic clusters with spot

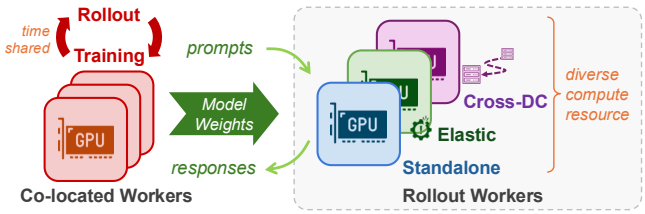


Figure 1. RL Workload with Diverse Rollout Patterns. *Rollout workers stream in prompts and model weights and stream out responses. While prompts and responses are lightweight, the model weights can be several terabytes in size, necessitating an efficient transfer system.*

churn, and multi-datacenter topologies. In a 1024-GPU training task, TensorHub reduces total GPU stall time by up to 6.7× compared to NCCL. With an elastic cluster, TensorHub's dynamic load balancing yields up to 4.8× faster updates than UCX. In a cross-datacenter setting, TensorHub cuts rollout GPU stall by up to 19× by avoiding redundant TCP transfers.

TensorHub has been deployed in ByteDance production to support cutting-edge RL training. With its strong performance and extensive functionality, TensorHub demonstrates a new way to build RL systems that operate effectively and efficiently at scale.

In summary, this paper makes the following contributions:

- Reference-Oriented Storage (ROS): an ownership-free storage abstraction specialized for model weights.
- TensorHub: a production-quality system that instantiates ROS with topology-aware transfer, model-parallel consistency, and fault tolerance.
- Three case studies on diverse rollout workloads to demonstrate TensorHub's performance and functionality benefits over other production baselines.

2 Background and Motivation

In modern LLM reinforcement learning (RL) workloads, weight transfer is essential for fully leveraging heterogeneous compute resources. This section covers the background of RL workloads, the goals of a weight transfer system, and the limitations of existing approaches.

2.1 RL Workload Background

A typical RL algorithm [27, 31] proceeds through three stages: (1) *rollout*, which runs inference and generates responses for given prompts; (2) *scoring*, which judges the responses with a reward module (*e.g.*, human ratings [2, 4, 23], a reward model [4, 23], or rule-based rewards [31]); and (3) *training*, which computes gradients and updates the model weights. The updated weights are used in the next rollout, forming a continuous feedback loop.

RL frameworks implement the algorithms by mapping rollout and training onto GPU workers. Figure 1 illustrates the rollout and training patterns used by representative RL frameworks. In a basic *co-located* architecture, rollout and

Approach	Example	Topology Awareness	Elasticity	Minimal Coordination	Minimum Data Movement	No Extra Space
Collective Communication	NCCL [22]	✓			✓	✓
Point-to-Point Communication	UCX [30]		✓		✓	✓
Distributed Storage	PS [17], Ray object store [20]	✓	✓	✓		
TensorHub		✓	✓	✓	✓	✓

Table 1. Comparison of Different Weight Transfer Approaches.

training time-share the same set of physical GPUs. We refer to these co-located workers as *trainers*. This architecture is widely adopted by veRL [33], ReaL [19], and RLHFuse [45] for its simplicity and high resource utilization.

However, rollout time can account for more than 90% of training time [11, 12, 33], thereby becoming the bottleneck. This motivates many new patterns to scale out rollout by utilizing additional GPUs:

- *Standalone rollouts* are GPU workers dedicated to rollout tasks. This design is widely adopted by many frameworks [8, 12, 32, 39].
- *Elastic rollouts* run on spot GPU instances that can be preempted or restarted at any time. Their lower price enables greater cost efficiency. RLBoost [40] reports up to 1.97× higher training throughput with this approach.
- *Cross-datacenter rollouts* run on GPUs located in a datacenter separate from the trainers. They enable hyperscale training [24] and leverage heterogeneous resource availability [44]. Because GPUs are often procured and deployed in different batches, inference-optimized GPUs may not be available in the trainers' datacenter [44].

With rollouts running on dedicated GPUs, model weights must be transferred from trainers after each training step, typically every few to tens of minutes. At scale, each update disseminates terabyte-sized model weights to thousands of GPUs. This transfer lies on the critical path: rollout workers cannot perform inference using partially updated weights, leaving their GPUs idle until the weight transfer is complete.

In addition, rollout workers are organized into model-parallel groups, each holding a full model copy sharded across its workers. These groups are loosely coupled with one another, progress at different paces as they generate variable-length outputs, and become ready for new weights at different times. The weight transfer system must serve these transfers dynamically across groups.

2.2 Weight Transfer System Goals

To serve these diverse rollout patterns, an ideal weight transfer system must satisfy five goals:

Goal 1: Topology Awareness. To fully utilize network bandwidth, the weight transfer system must incorporate network topology and runtime load knowledge. For cross-datacenter training, it must minimize the network traffic over slow inter-datacenter links. Within the same datacenter, it should avoid single-node bandwidth congestion.

Goal 2: Elasticity. The weight transfer system must support a dynamic cluster that scales up and down while transparently masking failures. If a worker fails or is preempted, the system must safely evict it without disrupting healthy workers. If a worker restarts or joins the cluster, the system must promptly serve weights on demand.

Goal 3: Minimal Coordination. Weight transfer should minimize control-flow coordination among workers. Coordination introduces temporal coupling, where workers must stop computation to cooperate. As a result, it not only complicates the programming model but also leaves GPUs idle while waiting and amplifies stragglers at scale.

Goal 4: Minimum Data Movement. The optimal data transfer must occur directly between the source and destination GPUs, with no intermediate hop. An extra hop implies additional network or PCIe costs, hurting overall efficiency.

Goal 5: No Extra Storage or Memory Space. As LLM weights can be several terabytes, storing extra copies requires massive storage or memory space. Such extra resource requirements should be minimized for cost efficiency.

2.3 Limitations of Existing Approaches

Existing weight transfer approaches fall into three main categories: *collective communication*, *point-to-point communication*, and *distributed storage*. Table 1 summarizes how well each meets the above goals.

Collective Communication. The NVIDIA Collective Communication Library (NCCL) [22] is widely used for weight transfer, adopted by veRL [33], OpenRLHF [12], AReaL [8], vLLM [16], SGLang [43], Slime [48], and others. In these frameworks, trainers broadcast weights to rollouts via NCCL collectives. NCCL detects the hardware topology and constructs ring or tree schedules that saturate RDMA bandwidth.

However, collective communication fails on the elasticity and minimal coordination goals. First, collective communication requires a pre-defined communication group that cannot add or remove nodes afterward. This prevents it from being applied to a dynamic cluster where nodes may fail, be preempted, or be restarted; any failure will cause the entire cluster to crash. Second, broadcast requires all nodes to cooperate, which in practice translates to a global barrier in RL frameworks that interrupts all workers and does not resume execution until all transfers have finished [8, 12, 33]. This amplifies stragglers, which are common in RL workloads. For example, a rollout may delay responding to an interrupt

while generating a long sequence, or network contention may slow a subset of nodes during transfer.

Point-to-Point Communication. Point-to-point communication (*e.g.*, UCX [30]) enables flexible data transfer between workers without a communication group, and thus naturally supports dynamic clusters. In an ideal case, such direct transfer can saturate the RDMA bandwidth between two endpoints, delivering high throughput.

However, point-to-point communication is less commonly used in RL because it has no global view of the runtime topology and therefore cannot balance load effectively. Senders serve requests independently, making their outbound bandwidth the bottleneck under fan-out. For example, when multiple rollouts pull weights from a trainer, they contend for the trainer's uplink bandwidth, degrading performance. Moreover, point-to-point communication still requires coordination at the framework level. When a trainer issues a `send()`, the corresponding rollout must be interrupted to invoke `recv()`; the `send()` blocks until the receiver cooperates.

Distributed Storage. Distributed storage systems, *e.g.*, parameter servers (PS) [15, 17] and the Ray object store [20], provide a compelling *decoupled* interface for weight transfer: trainers *push* updated weights to storage, and rollouts *pull* weights on demand. By replicating weights across storage nodes, such systems can exploit topology and locality, serving requests from nearby or less-loaded replicas. Moreover, this decoupling also naturally supports elasticity, since rollouts can join, leave, or restart independently without trainers' awareness. Finally, trainers can proceed immediately after pushing weights, while rollouts fetch weights on demand, minimizing coordination between them.

Despite these compelling properties, storage has not been adopted as the primary weight transfer mechanism in major LLM RL frameworks due to poor performance. For parameter servers [15, 17], the push-then-pull workflow amplifies network traffic, and the storage server's network bandwidth becomes a potential bottleneck at scale. The Ray object store [20] mitigates the network bottleneck by co-locating storage with workers, but still incurs extra data movement, requiring GPU–CPU copies and (de)serialization before transferring and reconstructing weights. For example, transferring 40 GB of data between two 8-GPU machines via the Ray object store takes 32 s in our measurement, whereas GPU-direct RDMA completes the same transfer in ~0.2 s. Such orders-of-magnitude performance losses can hardly be justified by functionality benefits. Lastly, storing TB-sized LLM weights is too expensive in either remote or co-located storage. In our experiment, the Ray object store commonly crashes due to out-of-memory errors when transferring more than 300 GB of data at once.

More fundamentally, data *ownership* is the root cause of storage's extra movement and space overhead. Each write operation copies weights into storage-managed space (either a remote server or co-located CPU memory), allowing the system to manage data independently of workers. This ownership enables trainer-rollout decoupling but also introduces additional serialization, copying, network traffic, and storage overhead that ultimately degrade performance.

Summary. Existing approaches present a trade-off. Collective communication achieves high throughput with topology-aware transfer, but requires rigid membership and global coordination. Point-to-point communication supports dynamic clusters and direct transfer, but struggles with load balancing and still incurs coordination at runtime. Distributed storage offers clean decoupling and flexibility, but pays the price of additional data movement and storage overhead due to full data ownership. As a result, none of these approaches satisfies all five goals—topology awareness, elasticity, minimal coordination, minimum data movement, and no extra storage or memory space.

3 Reference-Oriented Storage

Distributed storage offers a clean and flexible interface for weight transfer, but its reliance on data ownership introduces fundamental inefficiencies. To address this limitation, we present *Reference-Oriented Storage* (ROS), a new storage abstraction that does not own data.

ROS is enabled by three important properties of model weights in RL. First, weights are *highly replicated* due to data parallelism. This redundancy means there are multiple symmetric sources available for transfer. Second, weights used in inference are *immutable*. As a result, reading them from a remote worker's GPU memory is safe and requires minimal coordination. Third, weight versions are *short-lived*. Training constantly generates new versions, and only the most recent few are relevant, so long-term durability is unnecessary.

ROS directly repurposes those model weights *in place* to provide the illusion that certain versions of the weights are stored and can be fetched on demand. Under the hood, ROS does not physically store any copies of the weights. Instead, it tracks the workers with these weights on GPUs for inference; upon request, ROS directly uses them to serve reads. By avoiding explicit data ownership and extra copies, ROS combines the flexibility of storage with the efficiency of direct transfer.

This section covers the elemental primitives required for a minimal ROS system. We first present an overview of ROS and then elaborate on two key mechanisms required for integrity and availability. A more sophisticated system with rich features will be covered in §4.

3.1 ROS Overview

ROS consists of a centralized reference server and a client library imported by each worker. Figure 2 illustrates an example where Worker-0 holds a version of model weights in GPU memory for inference, and Worker-1 needs that version.

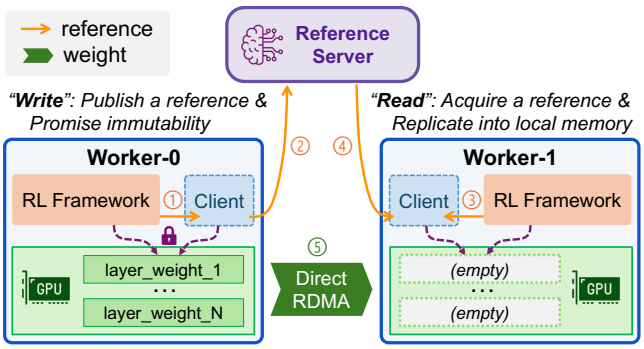


Figure 2. Reference-Oriented Storage Workflow. *The server only operates on lightweight references. The bulk weight transfer is directly between clients.*

ROS exposes `publish` and `replicate` primitives for write and read operations. ❶ To write weights into ROS, Worker-0 calls `publish` and passes a reference to its in-GPU weights to its client. ❷ The client forwards this reference to the reference server. ❸ To read weights, Worker-1 invokes `replicate` with a reference to its local weight buffers. ❹ Its client queries the server, which returns a source reference pointing to Worker-0's GPU memory. ❺ Worker-1's client then issues RDMA to read the weights directly from Worker-0 into its local buffers. Upon completion, Worker-1's copy becomes another replica of the model weights. Future replication requests can then be served from either worker.

Critically, these data flows require no additional data movement or storage overhead. Unlike traditional distributed storage, the ROS server never stores or transfers weight data. It only operates on the lightweight references. The bulk data transfer is directly between the source and destination memory without intermediate hops or staging space.

ROS treats all copies of the same weights as symmetric replicas, whether created via `publish` or `replicate`. A rollout thus need not fetch weights from a trainer; it can retrieve them from any peer, enabling efficient peer-to-peer data dissemination. Upon failure, a worker can restart and recover from any surviving replica, making the cluster self-healing.

Challenges. Since ROS only *references* weights but does not *own* them, it must address two challenges to be functional.

- *Integrity*: Weights require the flexibility for occasional mutations (*e.g.*, buffer reuse for a new version). ROS must define clear mutation semantics to avoid data corruption.
- *Availability*: Because ROS does not own the data, a weight version disappears if no worker holds a copy. The system must define which weights should remain accessible and introduce protocols to preserve that.

ROS introduces two key mechanisms to address the challenges: the *mutability contract* and the *retention protocol*.

3.2 Integrity via Mutability Contract

ROS relies on weight immutability for safe sharing, but RL training requires the flexibility for mutability. For example, as weights are too large to be duplicated on GPUs, a rollout may want to reuse the weight buffers in place when pulling the new weights. Similarly, a trainer must be able to mutate the weights to produce a new version.

ROS addresses this demand with the `unpublish` primitive and the *mutability contract*. When holding a replica, a worker commits not to mutate the weight data. When mutation is required, the worker must call `unpublish` to explicitly revoke the immutability commitment. Upon request, the reference server ensures this worker's weights are no longer visible to others; if there is an ongoing transfer from this worker, the server waits for the transfer to drain and only acknowledges unpublish success afterward. This ensures every concurrent transfer still receives uncorrupted data.

Fortunately, adhering to the mutability contract is easy in practice. An RL worker's execution flow is typically organized into stages, in which the immutability boundary is straightforward to identify. In addition, RL frameworks are often implemented in Python, whose memory safety avoids many dangling-pointer bugs.

3.3 Availability via Retention Protocol

ROS stores no copies, so if all workers unpublish a version, that version is unavailable. Such behavior is appropriate for overly stale weights, as they are not needed by any workers and thus safe to drop. However, losing the latest (few) versions of the weights is unacceptable. This scenario can occur in a corner case if all trainers unpublish the latest version before it is replicated by any rollout. To avoid such unavailability, the system requires clear semantics for "what data must be available" and a protocol to enforce that.

ROS fulfills this requirement with the *retention protocol*. ROS exposes `retain` semantics under which a worker can declare the versions to keep available (*e.g.*, the latest k versions). When processing `unpublish` requests, the server checks if the requesting replica is the last copy of a retained version; if so, it notifies the client to offload a copy of the weights to CPU memory and publish that as an *offload replica*.

Offloading is a safeguard for corner cases and is rarely triggered in practice. Rollouts typically see and replicate the new version promptly; as long as one copy is replicated, trainers can unpublish without triggering offload. When offloading does occur, only a single copy across the entire cluster is kept, minimizing memory overhead. Measurements show that offloading over the PCIe bus is fast (~48 GB/s in our setup), resulting in a minor delay for unpublish (<2 s even when offloading the entire GPU's weights). The offload replica is automatically released once it has been replicated by another worker (and is thus no longer the last copy) or once a newer version's publish makes it no longer retained.

The elegance of the retention protocol is that it provides similar availability to traditional ownership-based storage, while eliminating the ownership overhead in common cases.

Name	Description
open(...) -> ShardHandle	Acquire a shard handle. Optionally, declare the desired versions to retain for the retention protocol.
register(named_tensors)	Register weight tensors.
unregister()	Unregister weight tensors.
publish(version)	Publish a reference to the registered tensors under the given version.
unpublish()	Revoke the previously published reference.
replicate(version)	Replicate the given version of weights into registered tensors. *Supports relative versions.*
update(version) -> bool	Atomically check whether the given version is available and differs from the current one; if so, unpublish the current version and replicate the new one; return true if updated. *Supports relative versions.*
list() -> Dict	Return a dictionary mapping each available version to its set of replica names.
wait(predicate)	Wait until predicate(list()) evaluates to true.
close()	Destroy the handle; automatically unpublish and unregister if applicable.

Table 2. TensorHub APIs. open() *returns a handle, and the rest of the APIs are methods of this handle.*

3.4 ROS Summary

ROS achieves Goals 3, 4, and 5 (defined in §2.2) by design. By providing a storage interface, ROS enables rollouts to pull weights on demand without trainers' explicit coordination (Goal 3). By not taking ownership of weight data, ROS avoids the overhead of data movement (Goal 4) and extra storage space (Goal 5) in the common case.

ROS also establishes a good foundation for Goals 1 and 2. In principle, its centralized server architecture offers a global view of network topology and runtime load, making topology-aware transfer easier to implement (Goal 1). Elasticity can be achieved as long as the ROS system itself is fault-tolerant (Goal 2). In summary, both goals fit into ROS's design but require a well-optimized implementation.

4 TensorHub Design and Implementation

We build TensorHub, a production-quality system that instantiates the ROS idea with rich features required by real-world use cases. As LLM sizes increase, model parallelism is required, with each GPU worker holding only one shard of weights. Therefore, sharding must be natively incorporated in addition to the previous ROS requirements.

Overall, a production-quality system must satisfy four practical requirements. First, it must provide concise and expressive interfaces for easy integration into existing RL frameworks. Second, for efficient transfer at scale, the system must schedule transfer based on the network topology and runtime load. Third, for correctness, the system must guarantee that workers in the same model-parallel group see a consistent view of the system state, preventing control-flow divergence. Finally, to support dynamic cluster scaling up and down, it must gracefully handle failures.

This section first describes the naming scheme (§4.1) and APIs (§4.2) and then introduces topology-aware transfer (§4.3), model-parallel consistency (§4.4), fault tolerance (§4.5), and implementation (§4.6).

4.1 Naming Scheme

We begin with a hierarchical naming scheme to establish a precise vocabulary. In TensorHub, each model forms an independent domain, managed by a single reference server. Each model evolves over *versions*, each produced by one step of training. A version consists of one or more *replicas*, each representing a full copy owned by a model-parallel worker group. Finally, each replica is divided into *shards*, with each shard owned by a single worker. Figure 3 illustrates an example with three replicas, each composed of two shards.

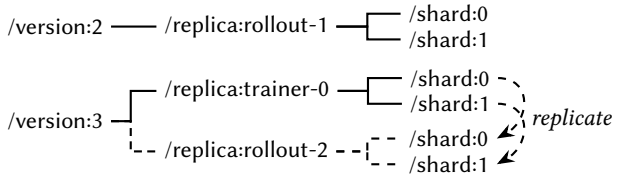


Figure 3. TensorHub Naming Scheme. *Rollout-2 fetches version 3 by replicating a copy; after doing so, it also becomes a replica capable of serving subsequent replication requests.*

Each version is uniquely identified by an integer, called the *absolute* version. TensorHub also supports *relative* versions, where the strings "latest" or "latest-k" are resolved to the latest version or the latest minus k, respectively. Relative versions are useful because RL typically only cares about freshness relative to the latest one. Off-by-k-step RL training workloads [8, 32, 39, 41, 47] may require multiple versions to coexist in the system, where "latest-k" is useful.

4.2 TensorHub APIs

TensorHub APIs are summarized in Table 2. As the same set of tensor buffers is commonly used across versions, TensorHub exposes ShardHandle to operate on them.

- **Open.** A worker begins by calling tensorhub.open() to acquire a handle for a shard within a replica. If it wishes to retain certain versions, it specifies them upon open.
- **Registration.** Before publishing or replicating, a worker must register weight tensors with the handle; before deallocating these tensors, it must unregister. These APIs support both CPU and GPU tensors, offering flexibility in memory management. For example, trainers in fully disaggregated architectures [39, 44] execute training in a tight

```python
import tensorhub
handle = tensorhub.open(
  model_name="actor",
  replica_name="trainer-0",
  num_shards=WORLD_SIZE,
  shard_idx=RANK,
)

tensors = get_weights()
handle.register(tensors)

for step in range(NUM_STEPS):
  handle.publish(version=step)
  do_rollout()

  handle.unpublish()
  do_train()
  # convert to the
  # inference-ready format
  postprocess_weights()

handle.close()
```

(a) Trainer.

```python
import tensorhub
handle = tensorhub.open(
  model_name="actor",
  replica_name="rollout-1",
  num_shards=WORLD_SIZE,
  shard_idx=RANK,
  retain="latest",
)
tensors = get_weights()
handle.register(tensors)

# fetch the initial weights
handle.replicate("latest")

while has_work():
  is_updated = \
    handle.update("latest")
  if is_updated:
    clear_kv_cache()
  do_rollout()

handle.close()
```

(b) Standalone rollout.

Figure 4. TensorHub Examples.

loop without co-located rollout and thus constantly mutate their on-GPU weights; they can publish weights from CPU memory.

- **Publish.** To make its registered tensors visible under version v, the worker calls `publish(v)` with an immutability commitment. When the worker desires mutation, it must invoke `unpublish()` first. These two calls form the core mutability discipline for reference safety.
- **Replicate.** A worker obtains weights by naming the desired version (absolute or relative). `replicate(v)` materializes version v into the handle's registered tensors, blocking if necessary until the version exists.
- **Update.** With a version held, a worker can call `update(v)` to atomically transition to a newer version if available. Atomicity avoids races between a version check and replication. This API is useful for a rollout to poll whether a new version of the weights is available; if not, it continues inference with the current weights.
- **Query.** TensorHub provides lightweight introspection through `list()` and `wait(predicate)`, which expose global metadata (such as available versions or replicas) for implementing custom policies. For example, rollouts can discard excessively stale responses, or trainers can wait for lagging rollouts to catch up [8, 32, 39, 41, 47].
- **Close.** When a worker finishes using the handle, it calls `close()` to release resources and clean up state.

Example. Figure 4a shows the workflow of a trainer that runs both training and rollout tasks. At the beginning of each step, it publishes the current weights and starts rollout. After completing a configured amount of rollout work, it unpublishes the weights and switches to training. Once training completes, it converts the updated weights to an inference-ready format (*e.g.*, resharding and quantization), publishes the new version, and begins the next rollout step.

Figure 4b shows an example of a standalone rollout. Upon initialization, it calls `replicate()` to fetch a copy of the weights before starting inference. Then, after finishing a batch of inference or receiving a control message, it calls `update()` to fetch a new version if available; otherwise, it continues inference with the existing weights.

4.3 Topology-Aware Replication at Scale

In RL training, every newly published version often needs to be fetched by multiple rollouts. A system that funnels all replication through a single node will collapse under network contention. TensorHub is therefore architected with a centralized reference server to schedule transfer based on network topology and runtime load, while the clients remain decentralized for scalable data movement.

4.3.1 Load-Balanced Scheduling at the Server. Efficient transfer begins with choosing the right source for each request. TensorHub centralizes scheduling on the server, exploiting its global view of the system for informed decisions.

The server always prioritizes a source replica located in the same datacenter. Within each datacenter, the server maintains a reference count for each replica—representing the number of replication requests it is currently serving—and always selects the least-loaded replica. Under pipeline replication (§4.3.3), each replica serves at most one request at a time, barring failures. This load-balancing policy also keeps unpublish latency low: if a source replica requests to unpublish, it needs to wait for in-flight replication to drain, which is brief and bounded by a single request. If no source replica is available within the same datacenter, the server falls back to cross-datacenter replication (§4.3.4).

In practice, a single server is sufficient to scale a training task to thousands of workers (more measurements in §5.1.4). As future work, if the server becomes a scalability bottleneck at even larger scales, TensorHub could partition workers into sub-clusters, each managed independently by a server. These servers would only need to lazily synchronize on critical events (*e.g.*, a new version is published).

4.3.2 High-Performance Transfer between Clients. TensorHub's data-plane efficiency relies on a high-performance transfer method between clients. To achieve this, each client embeds a hardware-affinity-aware transfer engine that uniformly supports RDMA and TCP transfers over both GPU and CPU memory. More specifically, the transfer engine offers three transfer modes, customized for workloads and available transport methods:

- *RDMA Direct*: In workloads where tensors remain allocated for long periods, the transfer engine registers tensor buffers directly with the RDMA NIC. Remote clients can then issue one-sided RDMA reads to pull data directly. This zero-copy path bypasses CPUs entirely and achieves the best performance.
- *RDMA Copy*: When users frequently reallocate tensors, repeated RDMA NIC registration is too costly. In this case, the transfer engine employs background threads to serve

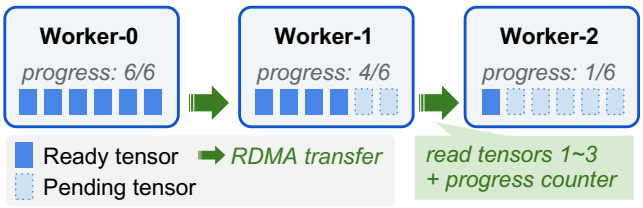


Figure 5. Pipeline Replication. *Worker-0 is the only source, while both Worker-1 and Worker-2 are requesting data. To scale throughput, TensorHub schedules a pipeline where Worker-2 reads partially replicated data on Worker-1.*

remote requests by copying the tensor data slice by slice into pre-registered buffers and then issuing RDMA writes. This approach eliminates re-registration overhead while still preserving RDMA's high throughput.

- *TCP*: In environments where RDMA is unavailable (*e.g.*, across datacenters), the transfer engine uses TCP.

The transfer engine is hardware-affinity-aware and optimized for modern multi-GPU, multi-NIC servers. For GPU-resident tensors, the engine selects the NIC closest to the tensor's GPU; in RDMA copy mode, it uses the pre-registered RDMA regions on the same GPU devices.

Tiny-Tensor Optimization. LLMs contain a large number of tiny tensors, which are inefficient to register and transfer. To address this, TensorHub compacts all tiny tensors (<2 MB) into contiguous buffers; only these compacted buffers are registered and transferred. During replication, the receiver unpacks them into their corresponding slices.

This optimization reduces registration costs and improves bandwidth utilization, at the cost of very minor memory overhead. For example, a 36B model has approximately five hundred tensors, half of which are tiny. When compacted, they require only about 3 MB of extra memory in total, which is negligible for a 19 GB shard.

4.3.3 Pipeline Replication.

Pipeline replication enables TensorHub to scale throughput beyond what a single source could provide. A key challenge in RL training is that large bursts of rollouts may simultaneously request the latest version, creating extreme fan-out. Pipeline replication allows partially replicated workers to begin serving others immediately, turning replication into an expanding pipeline rather than a single-rooted broadcast (as shown in Figure 5). Pipeline replication builds on the observation that most RDMA NICs are full-duplex, and therefore, when a NIC is receiving data, its uplink is idle and can serve data.

Pipeline replication uses the metadata and transfer mechanisms described below. Each replicating worker tracks a *progress counter* that indicates how many tensors of the target version it has received locally. When a new worker requests a version, the reference server selects the least-loaded source, which can be either fully replicated or still in progress. If the source is only partially replicated, the requester reads the source's progress counter and then fetches the prefix that is available. It then re-reads the progress counter and repeats until all data is received.

Pipeline replication transforms the topology from a fixed N-way fan-out into a bandwidth-amplifying DAG, where each replicating worker increases the system's bandwidth. Because transfers are decentralized and occur directly between peers, throughput grows with the number of active clients, and latency remains low even under heavy demand.

4.3.4 Cross-Datacenter Replication.

TensorHub's replication model generalizes to clusters spanning multiple datacenters, with a two-stage replication pattern.

When a new version is published, the first replica located in a different datacenter must fetch the data over TCP; we refer to this replica as the *seeding replica*. Once seeding completes, subsequent replicas in that datacenter can fetch the data over RDMA, enabling full pipeline replication locally without repeatedly invoking cross-datacenter transfer.

TCP-based seeding has substantially lower bandwidth than RDMA and can therefore stall pipeline replication if other replicas immediately try to update and fetch from the partially filled seeding replica. To avoid this stall, TensorHub augments the `update()` API with *smart skipping*: if a polling replica observes that a seeding replica is actively fetching over TCP, it treats the new version as temporarily unavailable and retries later, allowing seeding to complete before datacenter-local replication begins. This avoids serializing the entire pipeline behind slow cross-datacenter transfer.

To further reduce GPU stall time for seeding replicas, TensorHub offers an *offload seeding* option. When seeding is required, the client library creates a temporary offload buffer in CPU memory and initiates cross-datacenter transfer into this buffer in the background. The `update()` call returns immediately, and subsequent polling prioritizes consuming the offloaded copy if completed. This approach trades a bounded amount of CPU memory for reduced GPU stall time by creating at most one offload seeding replica per datacenter.

4.4 Consistency for Model-Parallel Sharding

LLM weights often exceed the memory of a single GPU, so model parallelism is required. In this setup, weights are sharded across a group of workers that must execute identical code, following an SPMD (single-program, multiple-data) paradigm. Divergence in a group risks hanging or corruption.

For correctness, all workers in a model-parallel group should ideally observe a consistent view of the system state, such as whether version N is available or which version is the latest. In practice, however, requests from different workers may reach the server in an interleaved order and observe inconsistent states. Figure 6 shows one such example.

TensorHub guarantees that all workers in a model-parallel group (*i.e.*, replica) observe the same control-plane state, thereby avoiding inconsistent decisions and SPMD control-flow divergence. The server implements this guarantee with

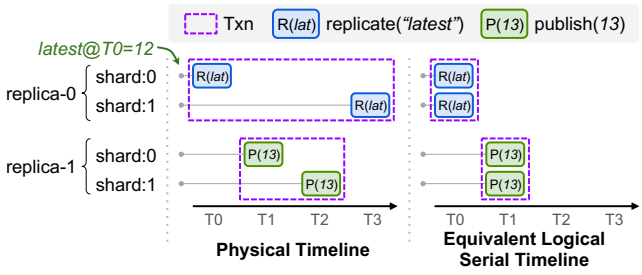


Figure 6. Sharding Consistency Example. *Left: the physical request order lets replica-0's shards observe different latest versions: shard:0 sees version 12 at T_0, while shard:1 sees version 13 after replica-1's two shards publish it at T_1 and T_2. Without coordination, this difference can cause SPMD control-flow divergence. Right: TensorHub serializes replica-0's replicate requests as one transaction, so both shards observe version 12.*

transactions at replica granularity. When the first request from a group arrives, the server starts a transaction, evaluates the operation for the entire group, and stores the resulting information (*e.g.*, the latest version) in a transaction-local buffer. Subsequent requests from that group consume this information. Transactions are serialized in their start order.

TensorHub does not enforce control-plane consistency across replicas. Each replica polls TensorHub independently and may observe a newer version at a different time. This behavior is acceptable for asynchronous, off-policy RL workloads, which allow rollouts to use slightly stale weights [21]. If cross-replica agreement is required, the RL framework can simply broadcast a message to all workers to pull a specified version from TensorHub.

4.5 Robustness to Churn and Failures

Large-scale training runs on clusters with highly variable reliability. Rollout workers on spot instances can be preempted at any time, and even non-spot nodes occasionally fail at scale. Our storage abstraction must therefore assume frequent replica churn and provide correctness without manual recovery. TensorHub is designed for this failure model: it detects failed replicas, safely re-routes ongoing replication, and integrates cleanly with our retention semantics.

General Failure Handling. TensorHub considers a client failed if no heartbeat arrives within the timeout window. Failure is handled at the replica granularity: the server removes the entire failed replica and releases associated resources.

Failures can occur during active replication. If a replica fails while fetching data, the timeout causes the server to invalidate the incomplete replica and release the reference count it acquired. Conversely, if a replica fails while acting as a source, the receiving replica detects the transfer failure and reports it to the server. The server marks the source as failed, preventing future replication from it, and selects an alternate source to recover the transfer.

Retention under Frequent Churn. Spot instances change the failure model: failures are not rare events but the norm. To ensure retention with spot instances, when counting the replicas that satisfy a retention rule, TensorHub excludes replicas hosted on spot instances.

In the pathological case where the last non-spot replica of a retained version fails, the system can no longer serve that version. In that event, TensorHub returns a graceful error indicating that the version is unavailable, and the client can retry later on another version. This behavior is acceptable in RL training because a new version will be trained and published shortly. The consequence of this corner case is a brief delay, not loss of training progress.

Reference Server Failure. Because the reference server maintains only soft state (*e.g.*, references), recovery is simple and lightweight. Upon detecting a server failure, a client resets its state to unpublished and connects to a preconfigured backup server. Critically, this new server does not need to recover any state from its predecessor; it simply waits to be populated by the next round of publishing from trainers. Before the new server is populated, rollouts continue using the existing weights without interruption. Eventually, all clients notice the failure and switch to the new server.

A special case arises when the server fails in the middle of a replica transaction, so some shards complete the operation while others observe the failure. To avoid divergence, the backup server uses a per-replica barrier: it waits for all shards to switch before allowing the replica to proceed. If some shards do not switch within a timeout, it returns an error to the RL framework, which can then restart this replica.

4.6 Implementation

TensorHub is implemented with 3.7 K lines of Python and 2.4 K lines of C++ code. The reference server is implemented as a Ray [20] named actor, instantiated lazily upon the first client call to `open()`. The TensorHub transfer engine is built on the Mooncake transfer engine [25] for RDMA.

Consistency Testing. To validate the server's consistency guarantees, we construct unit tests in which a single test process issues requests on behalf of multiple clients to deterministically simulate arbitrary request interleavings. Failures are injected by simply halting requests for a given client. These tests operate solely at the reference-request layer, without involving data transfer, and therefore do not require RDMA NICs or GPUs. Because all requests originate from a single process, the resulting executions are deterministic and reproducible, greatly simplifying debugging. This approach to simulated concurrency testing is inspired by FoundationDB [46], which has demonstrated in production its effectiveness at uncovering subtle concurrency bugs in distributed systems.

End-to-End Checksum. To verify end-to-end correctness, we further implement checksums. Upon publishing, the client computes the checksum of each tensor and attaches it to the reference. When another client acquires this reference as

a data source, it validates the checksum after transfer. In addition, these checksums help confirm correct enforcement of the mutability contract without corruption. Since tensors typically reside on GPUs, checksum computation is fast and can often be overlapped with RDMA transfer.

5 Evaluation

Our evaluation answers five questions: *(1) Can TensorHub transfer weight tensors efficiently and fully utilize RDMA bandwidth? (2) Can TensorHub scale replication throughput under bursty requests? (3) Can TensorHub mask failure transparently? (4) Can a single reference server sustain control-plane operations at scale? (5) Does TensorHub uniformly handle a variety of realistic RL workloads?*

We begin with microbenchmarks to evaluate TensorHub's data-plane efficiency, scalability under bursts, resilience to failure, and reference server capacity. We then present three case studies of distinct rollout workloads, focusing on (i) standalone rollouts, (ii) elastic rollouts on spot instances, and (iii) cross-datacenter rollouts, showing how TensorHub optimizes transfer with topology, supports dynamic cluster membership, and minimizes worker coordination, all without extra data movement or space.

Correctness and Validation. We validate the correctness of all experiments using the mechanisms described in §4.6. We implemented 1.3 K lines of concurrency tests that cover model-parallel consistency and failure cases. Each experiment runs a separate pass to verify end-to-end checksums on weights, and we trained real models to confirm that loss and accuracy match those from ByteDance production.

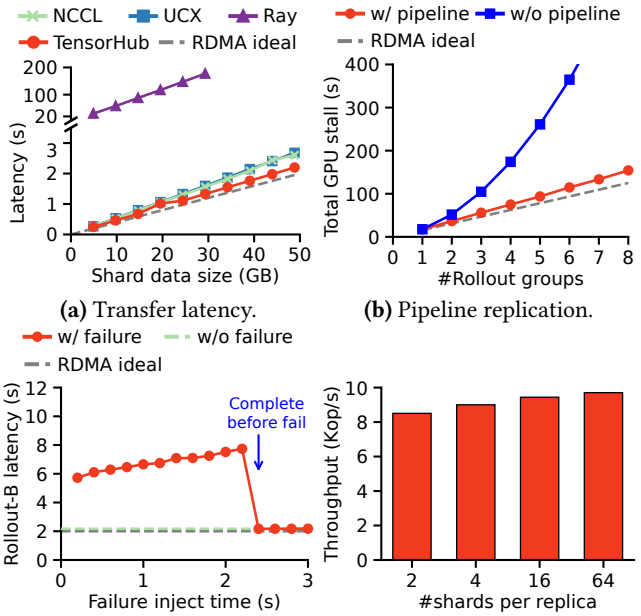
Hardware Specification. Unless otherwise specified, our experiments use machines with 160 CPU cores, 8 NVIDIA Hopper GPUs, 1800 GB of DRAM, 4 Mellanox InfiniBand RDMA NICs (400 Gbps), and 1 VPC NIC (200 Gbps). The reference server microbenchmark (§5.1.4) uses a CPU-only cluster with 32 CPU cores and 32 GB of DRAM per machine.

5.1 Microbenchmarks

We use four microbenchmarks to evaluate TensorHub's data-plane efficiency, scalability under bursts, failure resilience, and reference server capacity. Unless otherwise specified, each worker group consists of 8 shards on an 8-GPU machine.

5.1.1 Data-Plane Bandwidth Efficiency.

In the first microbenchmark, one machine serves as the trainer group and sends tensors, while another serves as the rollout group and receives tensors from the corresponding shards. We vary the number of tensors per shard to control shard size (50 MB per tensor) and measure the latency required to complete the transfer. We compare with NCCL [22], UCX [30], and the Ray object store [20], and show an additional *RDMA ideal* curve that denotes the theoretical number if the RDMA bandwidth is fully saturated (25 GB/s per shard).



(a) Transfer latency. **(b)** Pipeline replication.
(c) Transfer latency upon failure. **(d)** Reference server throughput.

Figure 7. Microbenchmark Results.

Figure 7a shows the results, where TensorHub consistently delivers the lowest latency. The Ray object store is two orders of magnitude slower than other approaches because it is primarily optimized for CPU memory and TCP, incurring prohibitive costs for GPU–CPU movement and serialization. In addition, Ray crashes at a per-shard size of 35 GB. Focusing on the three direct transfer approaches, TensorHub transfers 50 GB of data in 2.2 s with 22 GB/s throughput (88% of the theoretical), while NCCL and UCX only achieve 18.8 GB/s and 18.1 GB/s. These results show that TensorHub's data-plane implementation is highly efficient and can saturate RDMA bandwidth.

5.1.2 Scaling with Bursts.

In the second microbenchmark, we evaluate TensorHub's scalability under bursty request arrival. We use one machine for a trainer group and 1–8 machines for rollout groups; each shard contains 50 GB of data. All rollouts simultaneously request replication.

We define the *total GPU stall time* as the sum of stall times over all GPUs involved in the replication. This metric reflects the aggregated compute resource loss across the cluster. We report the total GPU stall time for both the default TensorHub and the one with pipeline replication disabled, together with the RDMA ideal number for reference.

Figure 7b shows total GPU stall time. With pipeline replication, the trainer group serves only one rollout group, and the remaining rollouts fetch peer-to-peer. Each shard transfers 50 GB in 2.2 s, independent of replica count. Total stall time scales linearly with the group count and remains close to the RDMA roofline. Without pipeline replication, all replicas fetch from the trainer, causing bandwidth contention. Stall time grows quadratically with group count. This scaling

gap demonstrates that pipeline replication is necessary for efficient weight dissemination at scale.

5.1.3 Transparent Failure Masking. We evaluate the system's resilience to failure with the third microbenchmark. Since failure at rest does not impact performance, we focus on failure during a transfer. A resilient system should mask failure without disrupting other healthy workers.

To evaluate this, we construct a pipeline where a trainer group sends data to rollout group A, which then forwards data to rollout group B. Both rollout groups start simultaneously. Each shard has 50 GB of data. We inject a failure into rollout-A sometime after the transfer has begun, and then measure whether rollout-B can complete the transfer without interruption, as well as the total time required.

Figure 7c presents the results, where rollout-B always completes the transfer. When the failure occurs immediately after the transfer starts, rollout-B quickly detects it and retries. The RDMA layer takes a conservative timeout of ~4 s before B confirms A's failure and requests recovery from the server. The server schedules B to directly read from the trainer to complete the transfer. For failures injected at later points, B takes longer to complete because retransmission is delayed. If the failure is introduced after 2.2 s, rollout-B has already finished the transfer and is therefore unaffected. These results demonstrate that TensorHub can transparently mask failure with only minor delay.

5.1.4 Reference Server Capacity. In the last microbenchmark, we measure the sustained throughput of a single reference server. This benchmark isolates the control plane: clients issue requests but perform no RDMA transfers. We use one machine to run the server and 64 machines to run a total of 8192 clients. Each client submits publish, update, and unpublish requests in a closed loop. We group clients into replicas and vary the number of shards per replica.

Figure 7d shows that a single server sustains 8.5–9.7 Kop/s. This capacity is sufficient for the control-plane load of RL clusters with thousands of workers. Throughput does not degrade as replica size increases, showing that model-parallel transactions add little overhead. Larger replicas reduce the number of replicas and the associated per-replica state, yielding slightly higher throughput.

5.2 RL Training with Standalone Rollouts

We now evaluate TensorHub in realistic RL settings. We use the ByteDance production framework, which extends veRL's [33] co-located architecture with standalone rollouts. The framework relies on a Ray driver to orchestrate weight transfer across workers, which is a common and representative design of RL frameworks [8, 12, 33].

Baseline: NCCL and UCX. We compare against two weight transfer systems used in the ByteDance production stack: one based on NCCL [22] and another based on UCX [30]. As discussed in §2.3, the communication-based weight transfer

Model size	9B	36B	260B	*mocked* 1T
#shards	2	4	8	16
Shard size (GB)	10	19	34	66
Trainer #GPUs	16	16	64	768
Standalone #GPUs	8	8	16	256

Table 3. Training Workload Parameters.

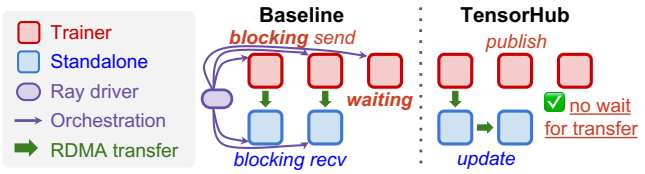


Figure 8. Weight Transfer Flows with Standalone. *TensorHub does not require the Ray driver to orchestrate weight transfer. Each standalone rollout pulls weights on demand.*

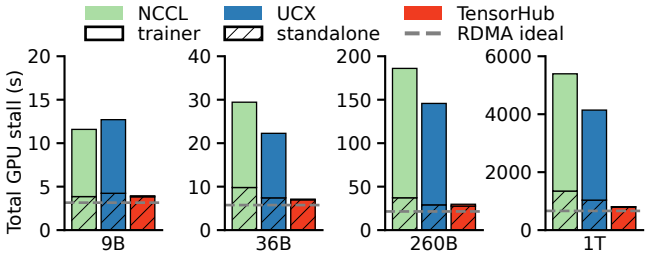


Figure 9. Standalone Rollout Results. *Ideally, only standalone rollouts need to stall due to their dependence on weights; trainers can proceed without waiting. TensorHub not only eliminates trainer stalls, but also keeps standalone stall time consistently lower.*

systems require the RL framework to coordinate workers. In practice, it is implemented as an RPC broadcast that interrupts all workers for a global weight transfer stage. More fine-grained coordination is theoretically possible, but it requires a major refactor of the framework. Though we are unable to compare with every possible implementation, we again include a roofline curve for reference (labeled "RDMA ideal"), denoting an ideal case with zero coordination cost or contention, bottlenecked only by the rollouts' RDMA bandwidth. Performance close to this reference implies near-optimal results. We did not include a comparison with other storage-based approaches because we have shown in §5.1 that they are orders of magnitude slower.

Engineering Effort for Integration. NCCL requires ~450 lines of code changes to the ByteDance production RL framework, mostly for coordination. UCX, with more low-level control, requires ~1200 lines, trading simplicity for flexibility. TensorHub requires only ~40 lines of change (similar to the example in Figure 4). This highlights that TensorHub offers a simple and user-friendly interface.

Workloads. We train four models for evaluation (Table 3), covering various sharding and GPU count setups. The shard count is the number of GPUs in one model-parallel replica,

and the total GPU counts determine the number of trainer and standalone replicas. For example, the 260B configuration uses 8-shard replicas, yielding 8 trainer replicas and 2 standalone replicas. The 9B, 36B, and 260B models are initialized from pretrained weights. To stress the system at an extreme scale, we construct a 1T-parameter model with mocked weight data by duplicating the 260B-parameter model's layer weights four times: training such a huge model requires 768 GPUs for trainers and 256 GPUs for standalone rollouts (1024 GPUs in total).

Results. Figure 8 compares the workflows of the baselines and TensorHub, and Figure 9 shows that TensorHub reduces total GPU stall time by up to 6.7× compared with NCCL and by 5.2× compared with UCX. These reductions stem from TensorHub's minimal-coordination design. First, `publish` is a lightweight reference-passing request that does not wait for actual data transfer; trainers immediately resume for their own co-located rollout tasks. In contrast, NCCL and UCX require coordination by the framework's Ray driver, and the driver does not move forward to its next stage until the weight transfer stage is completed.

Second, for large models, stragglers are amplified at scale: the weight transfer stage completes only when the slowest worker finishes. For the 1T model, transferring 66 GB should ideally take 2.6 s at full bandwidth, but NCCL and UCX stall all 1024 GPUs for 5.3 s and 4.0 s, respectively. TensorHub localizes the stall: mean latency is 3.1 s, and only the standalones (256 GPUs) need to wait. This demonstrates the benefit of decoupling trainer and rollout control flow.

5.3 Elastic Rollouts on Spot Instances

In the second case study, we evaluate TensorHub when rollout workers run on a hybrid of stable GPUs (for standalone) and preemptible spot GPUs (for elastic). In the ByteDance production environment, an autoscaler monitors training progress and instantiates new elastic rollouts when rollout tasks are backlogged and spare spot GPUs are available. When their GPU resources are preempted, elastic rollouts are killed immediately without a grace period.

Baseline. Only UCX-based weight transfer supports elastic rollout, because NCCL assumes static membership. Supporting elastic rollouts took substantial engineering effort for the ByteDance production UCX-based implementation. When a new elastic worker appears, trainers may be mid-training and unable to serve weights, so standalone rollouts must also act as data senders. This introduces a trainer-to-standalone-to-elastic transfer topology.

Workloads. We use the 260B model described in Table 3, but with one standalone machine (8 GPUs) and up to three elastic machines (24 GPUs). To keep experiments reproducible, we intercept the autoscaler and deterministically issue scale-up and scale-down events. Randomly chosen elastic replicas are terminated during scale-down.

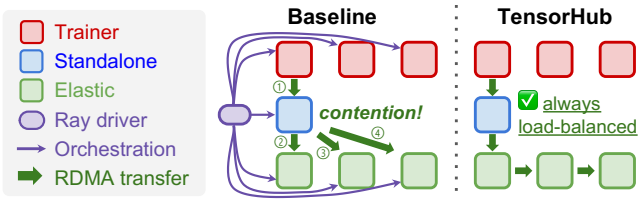
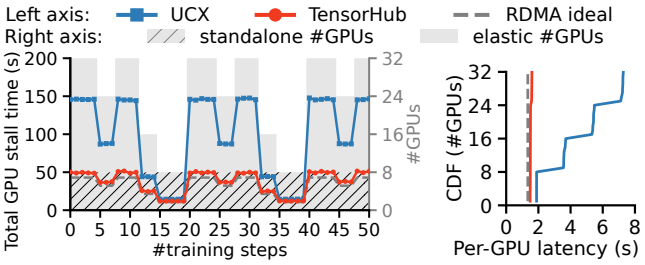


Figure 10. Weight Transfer Flows with Standalone and Elastic. *Note that TensorHub only shows one possible data flow; an elastic rollout can also fetch weights from a trainer.*



(a) Rollout GPU stall time over steps. *The curves (left axis) show stall time; the shading (right axis) shows GPU counts. Note that only standalone and elastic stall time is shown; UCX's additional trainer stall is omitted.*

(b) *Latency CDF at step 10; 8 GPUs for the standalone rollouts and 24 for the elastic rollouts.*

Figure 11. Elastic Rollout Results.

Results. Figure 10 compares the workflows, and Figure 11a shows total rollout stall time over training steps with GPU count varied. The stall time covers both standalone and elastic rollout GPUs; UCX's additional trainer stall is omitted.

In the UCX baseline, elastic rollouts must wait for standalone rollouts to pull weights from trainers first, introducing a delay. When elastic workers outnumber standalones, bandwidth contention further amplifies tail latency. Figure 11b shows per-GPU stall time at step 10: UCX produces a stair-shaped CDF, with the last batch of elastic GPUs waiting up to 7.2 s for weights.

With TensorHub, the reference server balances load with topology awareness at runtime, so elastic rollouts can fetch weights directly from trainers, standalones, or peer elastic rollouts, and pipeline replication spreads load across all available replicas. As a result, per-GPU stall time remains near-constant (~1.5 s) independent of the elastic GPU count, approaching the RDMA ideal. TensorHub reduces transfer latency by up to 4.8× compared with UCX. This result demonstrates that TensorHub naturally accommodates dynamic membership and runtime topology.

5.4 Cross-Datacenter RL Training

The final case study evaluates TensorHub in a multi-datacenter deployment. This setting reflects production scenarios where spare inference resources are available in a separate datacenter, and utilizing them requires weight transfers over slower inter-datacenter networks.

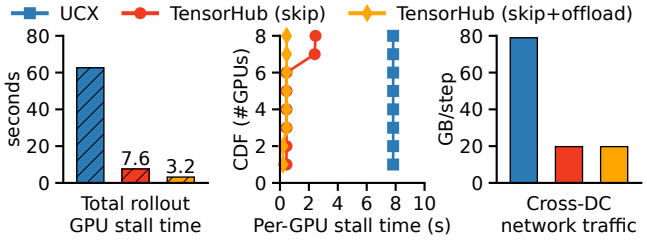


Figure 12. Cross-Datacenter Rollout Results. *Note that the left figure shows only the rollout stall time; UCX incurs additional trainer stall time that is omitted here.*

Baseline. We use a UCX-based implementation with TCP transport as the baseline. The transfer topology is similar to that of standalone rollouts (Figure 8), except the data path is now cross-datacenter over TCP.

Workloads. We use the 9B model from Table 3, with eight 2-shard trainer replicas (16 GPUs) and four 2-shard rollout replicas (8 GPUs). Trainers are located in a datacenter with training-optimized Hopper GPUs; rollouts run in a second datacenter with inference-optimized Hopper GPUs. The two datacenters are connected only via the 200 Gbps VPC NIC (one per machine) with TCP.

Results. Figure 12 reports total stall time across rollout GPUs, per-GPU latency distribution, and total cross-datacenter TCP traffic. TensorHub reduces total rollout GPU stall time by up to 19× compared with UCX. With the UCX baseline, every rollout replica fetches weights directly over TCP. Since all flows contend for the single VPC NIC per machine, stall time is dominated by cross-datacenter bandwidth: each GPU waits 7.8 s for weights. Such waiting is costly, given that an end-to-end training step takes only 68 s.

With TensorHub and smart skipping, only the first rollout replica (2 GPUs) waits for cross-datacenter TCP transfer (2.5 s), and the contention on the VPC NIC is reduced. Other rollout replicas can fetch data via high-throughput RDMA with merely 0.45 s latency. The result is a latency distribution with a single long tail for the seeding replica and near-ideal RDMA latency for all others. The end-to-end training step time is reduced to 60 s, 13% faster than UCX.

With offload seeding enabled, TensorHub further hides the cross-datacenter transfer latency by using an offload replica. The seeding replica stalls only for the 0.21 s PCIe transfer. In summary, TensorHub exploits datacenter locality and achieves near-local performance after a single cross-datacenter seeding transfer.

6 Experience and Lessons

We share our experience and lessons learned when operating the weight transfer systems in production.

Clean Interfaces Avoid Deadlocks. Weight transfer introduces synchronization across worker groups and hence is vulnerable to deadlock. In earlier NCCL/UCX-based weight transfer systems, we observed a series of related bugs. For example, one deadlock arose from a circular dependency between the Ray driver and workers: the driver asked all rollouts to stop; a standalone rollout acquired a write lock on its weights, while a bootstrapping elastic rollout blocked trying to read those weights from the standalone and could not respond to the driver; the driver then waited indefinitely. Such bugs arise because weight transfer creates a large critical section, and each worker runs customized logic to enter it, potentially introducing more dependencies.

In contrast, TensorHub APIs avoid this dependency pattern. `publish` is non-blocking; `update` returns once the transfer is complete; and `unpublish` waits only for ongoing readers to drain. Each API has fixed blocking behavior to minimize the critical section. No external, framework-specific coordination logic enters the critical section, preventing circular dependencies by design.

Decoupling Isolates Delays. TensorHub's decoupled interfaces also help to isolate the impact of suboptimal implementations. For example, in earlier systems we noticed cases where a trainer incorrectly ran manual garbage collection or expensive weight preprocessing in the middle of a weight transfer stage, causing unnecessary stalls for rollouts. TensorHub APIs block a rollout only for its transfer time, preventing unrelated preprocessing from ever entering the critical section.

7 Related Work

Reinforcement Learning for LLMs. Many frameworks [12, 33, 45] have been proposed to accelerate RL training. veRL [33] combines single-controller and multi-controller paradigms to drive stage execution efficiently. RLHFuse [45] further improves training compute efficiency with stage fusion. RollPacker [9] optimizes the long-tail decoding in rollout, while SpecActor [3] introduces decoupled and Best-of-N speculation to accelerate the rollout phase, ensuring on-policy model training. RLBoost [40] harvests preemptible GPUs to speed up the rollout phase with resource elasticity. Recent work also explores off-policy approaches for training acceleration. AsyncRLHF [21], AReaL [8], StreamRL [44], and Laminar [32] introduce asynchronous RL post-training with tailored optimizations to increase job throughput. TLT [13] trains a draft model in parallel for speculation when RL model training is performed; thus, the draft model also requires weight synchronization.

These frameworks optimize RL scheduling, which increasingly requires efficient weight transfer; they can thus benefit from TensorHub's clean weight transfer abstraction.

Storage Systems for Machine Learning. Prior studies have built storage systems for machine learning workloads. Parameter Server [17] stores model weights across a cluster of servers. Ray [20] provides an object store for RL, but is primarily optimized for CPU workloads. Hoplite [49] extends Ray with fault-tolerant collective primitives. Unlike

TensorHub, these systems require a system-owned copy of model weights, which is prohibitively expensive for LLM weights at TB scale.

Checkpointing systems are another class of storage for weights. Unlike RL weight transfer, they retain model states for recovery and are less latency-sensitive. GeminiFS [26] accelerates checkpointing and activation offloading by optimizing the metadata through a POSIX-like interface. ByteCheckpoint [36] is designed for efficient model weight storage and supports dynamic resharding across different training parallelism configurations. Gemini [37] stores in-memory checkpoint replicas for faster failure recovery.

Peer-to-Peer Data Dissemination. The idea of serving data requests from peer clients instead of servers dates back to cooperative caching [6] and xFS [1]. However, retaining additional copies on clients is too costly for LLM weights. TensorHub avoids this space overhead by borrowing the application-owned copies via a mutability contract and a retention protocol. Ekko [34] disseminates recommender-model updates peer-to-peer across WANs, exploiting their sparsity and locality with eventual consistency. In contrast, TensorHub focuses on transferring dense LLM weights on the performance-critical path of RL workloads.

8 Conclusion

We present Reference-Oriented Storage (ROS), a new approach to weight transfer in large-scale LLM reinforcement learning systems. By eliminating explicit data ownership and instead leveraging the inherent redundancy and immutability of model weights, ROS reconciles a long-standing trade-off between flexibility and efficiency in RL weight transfer systems. Building on this abstraction, TensorHub enables topology-aware, low-overhead weight transfer with minimal coordination, while preserving model-parallel consistency and fault tolerance in dynamic environments. Our evaluation shows that TensorHub can saturate network bandwidth and deliver near-optimal end-to-end performance. TensorHub has been deployed in ByteDance production for cutting-edge RL training.

Acknowledgments

We thank our shepherd, Juncheng Yang, and the anonymous reviewers for their helpful comments. We are also grateful to Jing Liu, Shivaram Venkataraman, Ming Liu, Mai Zheng, and Yuhong Zhong for their constructive feedback, and to Siyuan Chai for assistance with the reference server experiment.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. 1995. Serverless network file systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 109–126. doi:10.1145/224056.224066

[2] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, Nicholas Joseph, Saurav Kadavath, Jackson Kernion, Tom Conerly, Sheer El-Showk, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Tristan Hume, Scott Johnston, Shauna Kravec, Liane Lovitt, Neel Nanda, Catherine Olsson, Dario Amodei, Tom Brown, Jack Clark, Sam McCandlish, Chris Olah, Ben Mann, and Jared Kaplan. 2022. Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback. arXiv:2204.05862 [cs.CL] https://arxiv.org/abs/2204.05862

[3] Rongxin Cheng, Kai Zhou, Xingda Wei, Siyuan Liu, Mingcong Han, Mingjing Ai, Yeju Zhou, Baoquan Zhong, Wencong Xiao, Rong Chen, and Haibo Chen. 2025. Fast LLM Post-training via Decoupled and Best-of-N Speculation. arXiv:2511.16193 [cs.DC] https://arxiv.org/abs/2511.16193

[4] Paul F. Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 4302–4310.

[5] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261* (2025).

[6] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. 1994. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *First Symposium on Operating Systems Design and Implementation (OSDI 94)*. USENIX Association, Monterey, CA. https://www.usenix.org/conference/osdi-94/cooperative-caching-using-remote-client-memory-improve-file-system-performance

[7] Josef Dai, Xuehai Pan, Ruiyang Sun, Jiaming Ji, Xinbo Xu, Mickel Liu, Yizhou Wang, and Yaodong Yang. 2023. Safe RLHF: Safe Reinforcement Learning from Human Feedback. arXiv:2310.12773 [cs.AI] https://arxiv.org/abs/2310.12773

[8] 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.LG] https://arxiv.org/abs/2505.24298

[9] Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, Jiamang Wang, Lin Qu, Bo Zheng, and Wei Wang. 2025. RollPacker: Mitigating Long-Tail Rollouts for Fast, Synchronous RL Post-Training. arXiv:2509.21009 [cs.DC] https://arxiv.org/abs/2509.21009

[10] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. 2025. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature* 645, 8081 (2025), 633–638.

[11] Jingkai He, Tianjian Li, Erhu Feng, Dong Du, Qian Liu, Tao Liu, Yubin Xia, and Haibo Chen. 2025. History Rhymes: Accelerating LLM Reinforcement Learning with RhymeRL. arXiv:2508.18588 [cs.LG] https://arxiv.org/abs/2508.18588

[12] Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Zilin Zhu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, Weikai Fang, Xianyu, Yu Cao, Haotian Xu, and Yiming Liu. 2025. OpenRLHF: An Easy-to-use, Scalable and High-performance RLHF Framework. arXiv:2405.11143 [cs.AI] https://arxiv.org/abs/2405.11143

[13] Qinghao Hu, Shang Yang, Junxian Guo, Xiaozhe Yao, Yujun Lin, Yuxian Gu, Han Cai, Chuang Gan, Ana Klimovic, and Song Han. 2025. Taming the Long-Tail: Efficient Reasoning RL Training with Adaptive Drafter. arXiv:2511.16665 [cs.LG] https://arxiv.org/abs/2511.16665

[14] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. GPT-4o system card. *arXiv preprint arXiv:2410.21276* (2024).

[15] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 463–479. https://www.usenix.org/conference/osdi20/presentation/jiang

[16] 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* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[17] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 583–598. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu

[18] Shaoteng Liu, Haoqi Yuan, Minda Hu, Yanwei Li, Yukang Chen, Shu Liu, Zongqing Lu, and Jiaya Jia. 2024. RL-GPT: Integrating reinforcement learning and code-as-policy. *Advances in Neural Information Processing Systems* 37 (2024), 28430–28459.

[19] Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. 2025. ReaL: Efficient RLHF Training of Large Language Models with Parameter Reallocation. arXiv:2406.14088 [cs.DC] https://arxiv.org/abs/2406.14088

[20] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. https://www.usenix.org/conference/osdi18/presentation/moritz

[21] Michael Noukhovitch, Shengyi Huang, Sophie Xhonneux, Arian Hosseini, Rishabh Agarwal, and Aaron Courville. 2025. Asynchronous RLHF: Faster and More Efficient Off-Policy RL for Language Models. arXiv:2410.18252 [cs.LG] https://arxiv.org/abs/2410.18252

[22] Nvidia. [n. d.]. Nvidia/NCCL: Optimized Primitives for collective multi-gpu communication. https://github.com/NVIDIA/nccl

[23] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 27730–27744. https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf

[24] Dylan Patel, Daniel Nishball, and Jeremie Eliahou Ontiveros. 2024. Multi-Datacenter Training: OpenAI's Ambitious Plan to Beat Google's Infrastructure. https://newsletter.semianalysis.com/p/multi-datacenter-training-openais

[25] 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)*. USENIX Association, Santa Clara, CA, 155–170. https://www.usenix.org/conference/fast25/presentation/qin

[26] Shi Qiu, Weinan Liu, Yifan Hu, Jianqin Yan, Zhirong Shen, Xin Yao, Renhai Chen, Gong Zhang, and Yiming Zhang. 2025. GeminiFS: A Companion File System for GPUs. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 221–236. https://www.usenix.org/conference/fast25/presentation/qiu

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

[28] Amrith Setlur, Saurabh Garg, Xinyang Geng, Naman Garg, Virginia Smith, and Aviral Kumar. 2024. RL on incorrect synthetic data scales the efficiency of LLM math reasoning by eight-fold. *Advances in Neural Information Processing Systems* 37 (2024), 43000–43031.

[29] Amrith Setlur, Chirag Nagpal, Adam Fisch, Xinyang Geng, Jacob Eisenstein, Rishabh Agarwal, Alekh Agarwal, Jonathan Berant, and Aviral Kumar. 2024. Rewarding progress: Scaling automated process verifiers for llm reasoning. *arXiv preprint arXiv:2410.08146* (2024).

[30] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. 2015. UCX: an open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 40–43.

[31] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. arXiv:2402.03300 [cs.CL] https://arxiv.org/abs/2402.03300

[32] Guangming Sheng, Yuxuan Tong, Borui Wan, Wang Zhang, Chaobo Jia, Xibin Wu, Yuqi Wu, Xiang Li, Chi Zhang, Yanghua Peng, Haibin Lin, Xin Liu, and Chuan Wu. 2026. Laminar: A Scalable Asynchronous RL Post-Training Framework. In *Proceedings of the 21st European Conference on Computer Systems* (McEwan Hall/The University of Edinburgh, Edinburgh, Scotland UK) *(EUROSYS '26)*. Association for Computing Machinery, New York, NY, USA, 400–422. doi:10.1145/3767295.3803580

[33] 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

[34] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. 2022. Ekko: A Large-Scale Deep Learning Recommender System with Low-Latency Model Update. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 821–839. https://www.usenix.org/conference/osdi22/presentation/sima

[35] Grok 4 Team. 2025. Grok 4 Report. *xAI grok4 news* (2025). https://x.ai/news/grok-4

[36] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, et al. 2025. ByteCheckpoint: A Unified Checkpointing System for Large Foundation Model Development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 559–578.

[37] 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

[38] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh,

and Sida I Wang. 2025. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449* (2025).

[39] 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.LG] https://arxiv.org/abs/2505.24034

[40] Yongji Wu, Xueshen Liu, Haizhong Zheng, Juncheng Gu, Beidi Chen, Z. Morley Mao, Arvind Krishnamurthy, and Ion Stoica. 2026. RLBoost: Harvesting Preemptible Cloud Resources for Cost-Efficient Reinforcement Learning on LLMs. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 1323–1339. https://www.usenix.org/conference/nsdi26/presentation/wu-yongji

[41] Jianhao Yan, Yafu Li, Zican Hu, Zhi Wang, Ganqu Cui, Xiaoye Qu, Yu Cheng, and Yue Zhang. 2025. Learning to reason under off-policy guidance. *arXiv preprint arXiv:2504.14945* (2025).

[42] 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

[43] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. SGLang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.

[44] 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.LG] https://arxiv.org/abs/2504.15930

[45] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, and Xin Jin. 2025. Optimizing RLHF Training for Large Language Models with Stage Fusion. In *22nd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2025, Philadelphia, PA, USA, April 28-30, 2025*, Theophilus A. Benson and Radhika Niranjan Mysore (Eds.). USENIX Association, 489–503. https://www.usenix.org/conference/nsdi25/presentation/zhong

[46] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2653–2666. doi:10.1145/3448016.3457559

[47] Yuzhen Zhou, Jiajun Li, Yusheng Su, Gowtham Ramesh, Zilin Zhu, Xiang Long, Chenyang Zhao, Jin Pan, Xiaodong Yu, Ze Wang, et al. 2025. April: Active partial rollouts in reinforcement learning to tame long-tail generation. *arXiv preprint arXiv:2509.18521* (2025).

[48] 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.

[49] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. 2021. Hoplite: efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 641–656. doi:10.1145/3452296.3472897