Rivet: Efficient In-Place Recovery from Hardware-Operable Failures in LLM Training

Geon-Woo Kim
The University of Texas at Austin
Austin, Texas, USA
gwkim@utexas.edu

Joon Ha Kim
The University of Texas at Austin
Austin, Texas, USA
jhk.james1110@utexas.edu

Daehyeok Kim
The University of Texas at Austin
Austin, Texas, USA
daehyeok@utexas.edu

Abstract

Hardware-operable failures (HOFs) interrupt large language model (LLM) training but permit recovery on the same hardware without reset, repair, or replacement. Existing recovery systems nevertheless reload checkpoints, recompute lost progress, and rebuild process state, idling GPUs that could otherwise continue training.

We present Rivet, a fault-tolerant training system that leverages surviving hardware to enable efficient in-place recovery. Our key insight is that the state needed to resume training can be retained or prepared outside the active training process while remaining on the same hardware. Rivet retains the working model state and the reusable process state, and preinitializes the remaining state in a shadow trainer. We devise two-tier erasure protection and chunk-level transactional updates to keep the retained model state recoverable and consistent, and reclaim the shadow state when active training needs its GPU memory. Evaluation on 6- and 72-GPU NVIDIA A100 clusters shows that Rivet recovers 3.6–6.5× faster than the best-performing checkpointing baselines and improves productive training time by up to 13.7 percentage points. Large-scale simulation shows over 95% productive training time on a 131,072-GPU cluster.

1 Introduction

Training large language models (LLMs) at scale can occupy thousands of GPUs for weeks or months [14, 20]. A single failed training process, or *trainer*, can stall the entire job and idle otherwise healthy GPUs. We focus on *hardware-operable failures* (HOFs), in which trainers fail but training can resume on the same nodes and GPUs without hardware reset, repair, or replacement. Such failures can result from contained memory errors or temporary network faults that disrupt a trainer while leaving its hardware usable [53, 72]. Restarting training on the same hardware resolved 22.7% of failures across 19 large-scale production jobs [72], while a 256-GPU study reports that process restarts can resolve 73% of failures [22]. These reports show that HOFs account for a substantial share of training failures.

Today's recovery systems for LLM training do not fully exploit this surviving hardware. They typically restore *model state*, including parameters and optimizer state, from the latest checkpoint and recompute the work completed since that checkpoint [18, 21, 42, 67, 71, 73]. They also rebuild *process state*, such as CUDA contexts and communication resources, for the trainer that takes over after a failure. Keeping checkpoints in memory speeds up loading, while preinitializing trainers on spare nodes reduces process setup time [34]. Yet recovery still loads an older model state and repeats computation to regain pre-failure progress; the spare must also finish process setup that depends on which trainer it replaces. In our measurements, recovering Kimi-MoE [68] on 64 NVIDIA A100 GPUs takes 88 seconds with both optimizations, including 35 seconds for remaining initialization and checkpoint loading and 53 seconds for recomputation (§2.3).

This paper asks: *Can a replacement trainer reuse state on surviving hardware to minimize recovery overhead?* Our observation is that, for a HOF, the state needed to resume training can be retained or prepared outside the active trainer while remaining on the same hardware. The working model state and the reusable process state could remain accessible to a replacement process, avoiding checkpoint loading and recomputation of completed iterations. For process state that cannot outlive its owner, the unchanged hardware and training rank offer a second opportunity to initialize that state in another process before a failure. Together, state retention and preinitialization could remove much of the work that makes recovery expensive.

We present Rivet, a fault-tolerant training system that realizes these opportunities through *in-place state retention* and *in-place preinitialization*. Rivet pairs each active trainer with a lightweight *state manager* and a *shadow trainer* on the same node. The state manager owns the memory holding the working model state and the reusable process state, allowing their allocations to survive the active trainer. The shadow trainer prepares the remaining process state, including CUDA contexts and NCCL communicators, for the same GPUs and training rank. After a HOF, the shadow takes over and reattaches to the retained state.

Realizing this approach correctly and with low overhead raises three challenges. First, retained state remains exposed to memory faults that can make multiple pages inaccessible even while the GPU remains operable. Such memory faults are a significant source of training and infrastructure failures [20, 74]. Second, a failure can interrupt an in-place model update, leaving retained state partially updated and inconsistent across trainers. Third, preinitializing a shadow consumes GPU memory alongside active training, whose

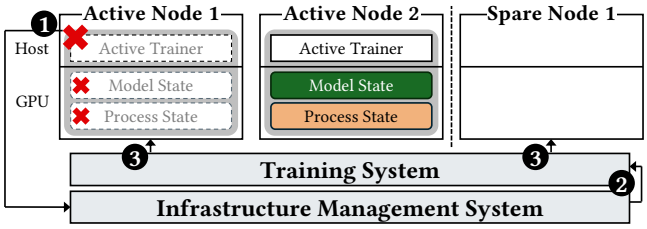


Figure 1. Training-job recovery. The infrastructure ❶ detects a trainer failure and ❷ checks hardware operability, then ❸ triggers in-place recovery (HOFs only) or spare-based recovery (either failure class).

changing memory demand can make the shadow cause an out-of-memory (OOM) failure.

We address these challenges through three key ideas. First, GPU-resident model state can be protected by redundancy in host memory, provided that the redundancy is itself protected. We therefore devise two-tier erasure protection in a *Resilient Memory Pool*, avoiding dedicated GPU memory for persistent parity. Second, fine-grained optimizer updates and incremental parity updates make it possible to preserve progress at chunk boundaries. We exploit this structure in the *Resumable Model Updater* to recover interrupted updates without full-model snapshots. Third, preinitialization can be deferred without losing training progress. Our *preinitialization controller* treats this state as reclaimable, trading recovery-time initialization for memory headroom during active training.

We implement Rivet on Megatron-Core, PyTorch, and NCCL in approximately 7.8K lines of Python and 2.4K lines of C++/CUDA, integrating shared state allocation, shadow reclamation on allocation failure, and GPU parity kernels. We will open-source our implementation.

We evaluate one dense and four mixture-of-experts (MoE) LLMs on two NVIDIA A100 clusters, the larger with nine eight-GPU nodes (72 GPUs). With one HOF injected every 10 minutes, Rivet recovers 3.6–6.5× faster than the best-performing state-of-the-art checkpointing baselines across measured workloads, including variants with spare-based preinitialization, improving productive training time by up to 13.7 percentage points. Large-scale simulation further shows that Rivet sustains over 95% of training time in productive computation on a 131,072-GPU cluster, even with a 10-minute mean time between HOFs.

2 Background and Motivation

2.1 Hardware-Operable Failures

A hardware fault can interrupt a training job without making its hardware unusable. We call a job failure a *hardware-operable failure* (HOF) if one or more training processes terminate or hang, but training can resume on the same nodes and GPUs without hardware reset, repair, or replacement. A *fatal failure* (FF), in contrast, requires such remediation before the affected hardware can be reused. Both interrupt training; the distinction is whether the affected hardware can be reused without remediation.

Modern hardware can contain faults within a process or memory region while leaving the device operable [40]. For example, a contained uncorrectable error in GPU high-bandwidth memory (HBM) can terminate the affected process while leaving the GPU available to run a replacement without a reset [53]. The faulty memory pages may become inaccessible, but the remaining memory is still usable. Similarly, a temporary network link outage can leave training processes hung in collective communication after connectivity returns, requiring a process restart while the nodes remain usable [29, 33, 72].

Production reports show that many failures can be resolved by restarting training on the same hardware. Across 19 large-scale training jobs at ByteDance, 22.7% of failures were resolved by restarting training on the same hardware [72]. Separately, a 256-GPU study reports that 73% of failures can be resolved by restarting the training processes [22].

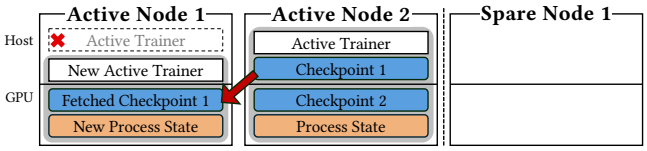
We expect HOFs to become more prevalent for two reasons. First, growing GPU counts and memory capacities increase fault exposure [2, 9]. A production study reports 3.2× more frequent uncorrectable memory errors per GPU on newer H100 GPUs (96 GB of HBM) than on A100 GPUs (40 GB), potentially reflecting the increase in memory capacity [11]. Second, improved fault containment lets more errors interrupt processes without disabling hardware, enabling recovery on the same hardware [40, 53].

2.2 Failure Recovery: Where and What to Recover

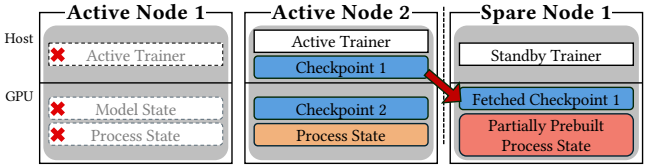
When a training process, or *trainer*, fails, another trainer must take over its role in the job. The infrastructure decides where this *replacement trainer* will run [20, 23, 26, 33, 72]. As Figure 1 illustrates, it ❶ detects the failure, ❷ identifies the affected nodes and checks whether their hardware remains operable, and ❸ triggers recovery [16, 72, 74].

Where to recover. Recovery follows one of two placement policies. *In-place recovery* launches replacement trainers on the same nodes and GPUs [54, 62]. *Spare-based recovery* launches them on spare nodes that have already passed hardware health checks [34, 72]. An FF requires spare-based recovery to resume training without waiting for hardware remediation. A HOF permits either form of recovery, with in-place recovery often favored to keep spare nodes available for other failures [22, 54, 72].

What to recover. Regardless of where it runs, a replacement trainer needs two classes of state before it can resume training. *Model state* includes the model parameters and optimizer state updated over successive training iterations. *Process state* includes the runtime resources needed to execute those iterations, such as CUDA contexts, communication resources, and GPU and host memory buffers.



(a) In-place recovery.



(b) Spare-based recovery with preinitialization.

Figure 2. HOF recovery using in-memory checkpointing (*e.g.*, Gemini [73]), with and without spare-based preinitialization.

These states normally belong to the trainer process, so launching a new process on the same GPU does not automatically restore them. Recovery systems such as Gemini [73] and MoEvement [18] restore the model state and initialize replacement trainers even after a HOF. Thus, *recovering on the same hardware does not by itself avoid rebuilding the model and process state needed to resume training.*

2.3 The Cost of Reconstructing State

Most training systems today rely on checkpointing to restore model state after a failure [18, 21, 42, 67, 71, 73]. Spare-based preinitialization complements checkpointing by preparing some process state in advance, but a preinitialized trainer still needs the model state before it can resume training. Figure 2 illustrates these recovery paths.

Checkpointing. Training systems periodically save copies of model state, called *checkpoints*, in storage that survives a trainer failure, such as remote host memory [18, 73] or local disk [42, 67]. Under in-place recovery (Figure 2a), a replacement trainer starts on Active Node 1, initializes its process state, and loads its latest checkpoint from Active Node 2. The trainers roll back to their checkpoints and recompute the work completed since those checkpoints. Keeping checkpoints in memory speeds up loading, but recovery still restores an older model state and repeats computation to regain pre-failure progress.

Spare-based preinitialization. A standby trainer on a spare node can initialize some process state before an unexpected failure, as in TrainMover [34]. In Figure 2b, the standby on Spare Node 1 already holds resources such as pinned host memory buffers. However, which trainer it replaces is determined only after a failure. The standby must then finish setup for the assigned rank in the distributed job, including connections to the other trainers. For checkpoint-based recovery, the standby then loads the failed trainer's checkpoint, and the job recomputes lost work. Preinitialization reduces process-state setup time while leaving checkpoint loading and recomputation necessary.

Recovery cost. We measure this combination by training Kimi-MoE on 64 NVIDIA A100 GPUs with Gemini+, which adds spare-based preinitialization to Gemini's in-memory checkpointing. Recovery takes 88 seconds: 35 seconds for remaining initialization and checkpoint loading, and 53 seconds to recompute lost progress (§8.3). The time for initialization and checkpoint loading falls from 59 seconds with Gemini to 35 seconds with Gemini+. Even with both techniques, recovery spends tens of seconds preparing replacement trainers and restoring pre-failure training progress.

Recovery latency reduces productive training time across the distributed job. The Effective Training Time Ratio (ETTR) measures the fraction of wall-clock time spent on productive training [33]. Prior work projects a mean time to failure (MTTF) of 0.23 hours for 131,072-GPU jobs [33]. At that projected failure rate, an illustrative recovery cost of 88 seconds per failure would consume roughly 10.6% of wall-clock time, before accounting for other overheads.

2.4 Opportunities for In-Place Recovery

These recovery paths rebuild state and replay training even when the affected hardware remains operable, limiting productive training time. ***Can the surviving hardware help avoid this process?*** It offers two complementary opportunities for in-place recovery.

Opportunity 1: Retain state in place. State that can outlive the training process could remain resident on the still-operable hardware (*e.g.*, GPU HBM or host memory) and be accessed directly by a recovering trainer. Retaining the working model state and the reusable process state (*e.g.*, GPU communication buffers) outside the trainer's failure domain could therefore avoid checkpoint loading, recomputation of completed iterations, and buffer recreation.

Opportunity 2: Preinitialize process state. State tied to the training process, such as CUDA contexts and NCCL communicators, must be created for the recovering trainer. However, its initialization depends on the GPUs, network interfaces, and rank that the trainer will use. By preserving these assignments, in-place recovery could enable this state to be initialized before a failure, moving the remaining process-state initialization off the recovery critical path.

Together, these opportunities could shorten in-place recovery through state reuse and advance preparation.

3 Rivet Overview

We present Rivet, a fault-tolerant training system that realizes both opportunities in §2.4 to recover from HOFs in place. Rivet pairs each active trainer with a lightweight *state manager* and a *shadow trainer*, both on the same node. These components enable two recovery mechanisms:

In-place state retention. The state manager retains the working copy of the model state and the reusable process state (*e.g.,* GPU communication buffers) outside the active trainer's failure domain. After a HOF, the replacement trainer accesses this state directly, avoiding checkpoint loading and recomputation of completed iterations (Figure 2a).
In-place preinitialization. The shadow trainer initializes the remaining process state, including CUDA contexts and NCCL communicators, before a failure. Because it will take over the active trainer's GPUs, network interfaces, and rank, it can prepare states that are specific to these assignments in advance, unlike a spare awaiting its replacement assignment (Figure 2b). Upon a HOF, it takes over the active role, reusing the state prepared in advance.

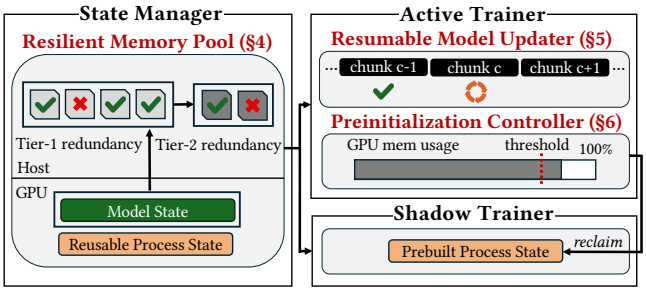


Figure 3. Rivet's architecture, with a state manager for retained state and a shadow trainer for preinitialization.

3.1 Design Goals

Faster recovery must preserve training results and improve effective throughput. We therefore pursue three design goals:
G1: Guarantee exact recovery. Given the same input-batch and random-number-generator sequences, initial model state, and training configuration, and assuming deterministic execution, the model state and loss at each completed iteration must match those of a fault-free execution, regardless of failure timing. Small perturbations can silently steer training to a different optimum and are hard to diagnose [25, 39].
G2: Minimize steady-state overhead. Rivet should minimize iteration-time overhead to improve effective throughput even when HOFs are a minority of failures.
G3: Minimize GPU memory overhead. Rivet should minimize the GPU memory consumed by recovery support, including prebuilt process state, to accommodate LLM training workloads whose HBM utilization often exceeds 90% [34].

3.2 Challenges

Realizing state retention and shadow preinitialization under these goals raises three challenges.
C1: Memory page failures in retained state. The state manager retains state outside the active trainer's failure domain, but that state remains exposed to faults in the underlying memory. A contained HBM error can leave the GPU operable while making multiple pages of retained state inaccessible [36, 53]. Rivet must tolerate multiple page failures with minimal GPU-memory overhead.
C2: Inconsistent model state from mid-update failures. The state manager retains the working copy that the active trainer updates in place. A failure during an update can leave this copy partially updated and inconsistent across ranks, even when all its pages survive. Rivet must recover a consistent training state without the memory cost of a full extra copy or the steady-state cost of snapshotting.
C3: Memory contention from prebuilt process state. The shadow trainer's preinitialized GPU state shares HBM with active training, whose memory demand varies over time, especially with MoE routing imbalance [52, 81]. A shadow that initially fits can therefore later cause an out-of-memory (OOM) failure in the active trainer. Rivet must preserve preinitialization's recovery benefit while meeting active training's GPU-memory needs.

3.3 Key Ideas

We propose three techniques to address these challenges.
I1: Resilient memory pool with two-tier erasure protection (§4). We design the *Resilient Memory Pool* (RMP) to retain model and reusable process state, protecting model contents against memory page failures. RMP uses erasure-code parity to reconstruct lost contents, but storing that parity in HBM would consume scarce training memory. We therefore devise *two-tier erasure protection* that keeps both persistent parity tiers in host memory. The first tier protects retained HBM data, while the second protects first-tier parity against host-memory failures.
I2: Resumable model updater with chunk-level transactions (§5). We design the *Resumable Model Updater* (RMU) to make in-place model updates crash-consistent. We exploit the fine granularity of model updates and incremental erasure-code parity updates to devise chunk-level transactions that commit the model state and its parity together. This lets each rank resume its interrupted update and reach a consistent iteration boundary without snapshotting the full model. RMU also reuses activation memory for update buffers once backward propagation completes, avoiding any increase in peak HBM usage.
I3: OOM-free preinitialization with runtime reclamation (§6). We observe that the training runtime's memory allocator provides an opportunity to reclaim memory before reporting an OOM, and that prebuilt process state can be recreated if discarded. We design a *preinitialization controller* that uses this opportunity to reclaim the shadow trainer's GPU memory when active training needs it. By adapting preinitialization to available memory, our design preserves fast recovery when memory permits while preventing preinitialization from causing training OOMs.

3.4 HOF Recovery Workflow in Rivet

Figure 3 shows the components involved in Rivet's HOF recovery. When an active trainer fails, RMP retains its model state and reusable process state, and localizes and marks any pages made inaccessible by the failure.

To use their prebuilt NCCL communicators, the shadow trainers take over together. A shadow whose GPU state was reclaimed completes its deferred initialization before the role switch. Surviving active trainers release their process-owned GPU memory and become shadows; Rivet launches a new shadow for the failed trainer.

Each promoted trainer reattaches to the retained state, and its RMU reconstructs failed protected pages using the available data and parity. If no rank has begun updating its model state, training restarts the interrupted iteration. Otherwise, each rank's RMU restores any interrupted chunk and resumes from its last committed chunk boundary until all ranks complete that iteration's update.

Rivet targets HOF recovery; an FF prevents reuse of the affected hardware and may destroy its retained state. For these failures, Rivet uses existing checkpointing systems unchanged and restores training from their checkpoints (§2.3).

4 Resilient State Retention

Retained state must remain recoverable despite memory page failures, with minimal GPU-memory overhead. We design the *Resilient Memory Pool* (RMP), a memory-management component within each state manager that owns allocations for retained model and reusable process state.

4.1 Two-Tier Erasure Protection

GPU HBM uses error-correcting codes (ECC) to correct single-bit errors but cannot repair all error patterns [53]. When a double-bit flip occurs within an ECC-protected domain (*e.g.,* 32 bytes in HBM3 and later), the memory controller detects the uncorrectable error and reports it to the GPU driver. The driver identifies the coarse-grained failure domain containing the error (*e.g.,* a DRAM row) and retires all virtual memory pages mapped to that physical region (*e.g.,* 16 virtual pages on AMD MI300X [36]), preventing further access to them [36, 53]. A localized fault thus creates multiple page *erasures*, meaning that RMP can identify the failed pages but cannot read their contents.

We use Reed–Solomon (RS) erasure coding [66] to reconstruct these lost pages. RS computes redundant parity pages from data pages and stores them separately; the number of parity pages determines how many page erasures the code can tolerate. Within the code's configured tolerance, surviving data and parity suffice to reconstruct the lost contents.

The challenge is to provide enough parity for multiple page failures without reducing the HBM available to memory-intensive LLM training [20, 34]. For example, Kimi-MoE uses up to 14.1 GiB of GPU memory for its model state in our evaluation (§8.1). With the minimum HBM virtual-memory allocation granularity of 2 MiB on NVIDIA A100 GPUs, tolerating 16 page failures within each 2 GiB DRAM die requires 16 parity pages per die. Storing this parity in HBM would consume an additional 256 MiB for this model state. Stronger protection raises this cost: tolerating two concurrent failures, each spanning 32 pages within a DRAM die, requires 1 GiB of additional HBM.

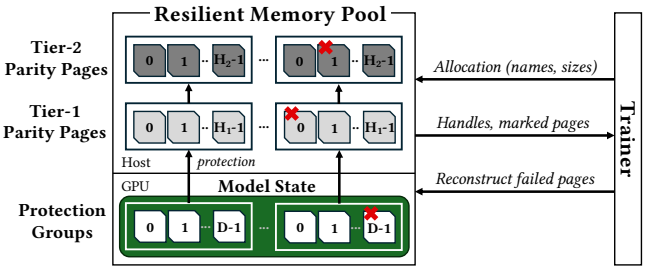


Figure 4. RMP organizes model state into protection groups of D HBM data pages, protected by H_1 first-tier and H_2 second-tier parity pages in host memory.

Our approach. Our key insight is that retained data and their recovery redundancy need not occupy the same memory tier. We exploit this separation to devise *two-tier erasure protection*, which keeps persistent parity in host memory to avoid reserving HBM for both parity tiers. As illustrated in Figure 4, RMP partitions the retained model state into fixed-size *protection groups* of HBM pages and stores each group's first-tier parity in host memory. Host memory, however, can also suffer uncorrectable ECC errors, leaving that parity unavailable. We therefore protect the first-tier parity with a second tier of parity pages, also in host memory.

Each protection group contains D data pages, with H_1 first-tier and H_2 second-tier parity pages; each page has size S. By default, RMP uses $D = 1024$, $H_1 = 16$, $H_2 = 1$, and $S = 2$ MiB, yielding a 2 GiB protection group. The first tier can reconstruct up to 16 failed HBM pages per group, while the second tier can restore one failed first-tier parity page. If retained state cannot be reconstructed, Rivet falls back to checkpoint recovery.

4.2 Retained State Lifecycle

Our design separates memory ownership from data access so that the state manager can retain state without touching potentially faulty pages. Trainers initialize and update their model state, compute parity pages, and reconstruct failed pages. RMP allocates memory, maintains allocation and failure metadata, and remaps unavailable virtual pages to healthy memory after failure localization.

State allocation. When training begins, a trainer requests each retained state from RMP by a stable name and size. RMP allocates HBM pages for each retained state and returns handles to the trainer. For model state, it additionally allocates

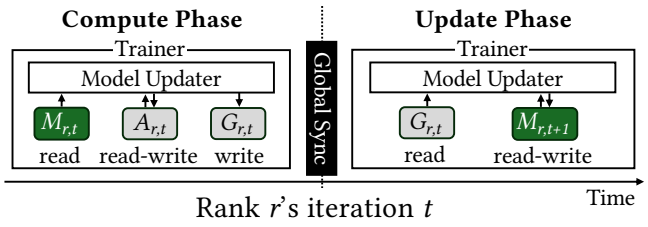


Figure 5. Read, write, and read-write accesses to model state (M), activation buffers (A), and gradients (G) during the compute and update phases of rank r's iteration t.

Table 1. Average fault-free iteration time over 30 iterations on OLMoE (Table 2), split into compute and update phases. Reference is the model updater without crash-consistency mechanisms; Sync and Async Snapshotting are the two strawman solutions.

Model Updater	Compute (ms)	Update (ms)	Total (ms)
Reference	12566	47	12613
Sync Snapshotting	13424	82	13506 (+7.1%)
Async Snapshotting	13372	82	13454 (+6.7%)
RMU	**12572**	**111**	**12683 (+0.6%)**

the associated first- and second-tier parity pages in host memory. For reusable process state, RMP retains the HBM allocation without parity protection because its contents are temporary intermediate values that need not survive a failure. Failed pages are remapped to healthy memory without restoring their contents. A replacement trainer uses the same name to reattach to the retained allocation.

Failure localization. After a HOF, RMP identifies unavailable pages before the promoted trainer accesses retained state. For host memory, it scans coarse-grained regions with the madvise system call [37] and recursively bisects any region that returns EHWPOISON, indicating that it contains a failed page. For HBM, RMP checks each page's mapping with a device query API (*e.g.,* cuPointerGetAttribute [51]). RMP marks the failed pages and remaps them to healthy memory. Remapping restores usable storage; protected contents still require reconstruction from parity.

Page reconstruction. The promoted trainer receives the retained-state handles and the failed-page marks. For protected state, second-tier parity repairs missing first-tier parity pages, and first-tier parity repairs failed HBM pages. The trainer must use data and parity from a consistent update boundary for reconstruction. RMU coordinates reconstruction and recovery from interrupted model updates (§5).

5 Crash-Consistent Model Update

Retaining model state and protecting its pages does not ensure that it remains consistent after a failure. A HOF can interrupt in-place updates to the model state and its parity, leaving them inconsistent even when no pages are lost. We design the *Resumable Model Updater* (RMU) to execute these updates within each trainer and make them crash-consistent with low steady-state and GPU-memory overhead.

5.1 The Crash-Consistency Problem

An LLM training iteration consists of a compute phase followed by an update phase (Figure 5). During the compute phase of iteration t, rank r computes gradients $G_{r,t}$ from its input batch, using activation buffers $A_{r,t}$ to hold intermediate values for backpropagation. The model state $M_{r,t}$ remains read-only throughout this phase. If a HOF interrupts computation, the partial activations and gradients can therefore be discarded and recomputed from the unchanged $M_{r,t}$. After computing gradients, ranks synchronize to obtain any global gradient statistics needed for the update (*e.g.,* the global gradient norm used for clipping [57]).

During the update phase, the gradients $G_{r,t}$ are read-only, while the model state becomes read-write. The model updater applies the optimizer (*e.g.,* Adam [31]) to change the model state in place from $M_{r,t}$ to $M_{r,t+1}$ and refreshes the corresponding host-resident parity. A HOF can interrupt either operation, leaving the model state partially updated or its parity inconsistent with it. Recovery must restore a consistent model state and matching parity at each rank and bring all ranks to the same iteration boundary.

Strawman solution. A straightforward approach is to snapshot the model state $M_{r,t}$ and its parity into separate host-memory buffers before the update phase. Each rank completes its snapshot before the global synchronization preceding the update phase, ensuring that no rank starts updating until all snapshots are ready. After a failure during the update, each rank restores its own pre-update state $M_{r,t}$, and all ranks retry the interrupted iteration. Once the update completes, each rank encodes new parity for $M_{r,t+1}$ and copies it to the host-resident parity pages.

Preserving a full snapshot, however, imposes substantial steady-state overhead. The GPU-to-host copies increase iteration time by 7.1% (Table 1). Asynchronous snapshotting overlaps copies with computation, but resource contention still increases iteration time by 6.7%.

5.2 Chunk-Level Transactional Updates

RMU reduces the cost of crash consistency by preserving progress at chunk boundaries.

Key observations. Two properties of model updates and erasure coding enable this design. First, modern LLM optimizers update model state at fine granularity. Adam [31] operates element-wise, while Muon [38], adopted in recent LLM training [68], operates layer-wise. This structure lets us divide updates into independently committable chunks. Second, linear erasure codes support incremental parity updates. We exploit this property of RS coding to update a protection group's parity from each chunk's delta, avoiding recomputation over the entire group.

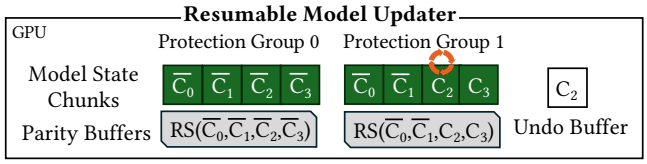


Figure 6. RMU chunk-level transactions. Bars mark committed chunks; $RS(\cdot)$ denotes Reed–Solomon encoding. The undo buffer preserves C_2 while its update is in progress.

Our approach. Building on these observations, we devise *chunk-level transactions* that commit the model state and its parity together. As shown in Figure 6, RMU partitions each RMP protection group into chunks and updates them one at a time. An *undo buffer* preserves the current chunk's pre-update value, while a *parity buffer* holds parity for the protection group at its latest committed chunk boundary.

In the example, Protection Group 0 has finished updating. In Protection Group 1, chunks $\overline{C}_0$ and $\overline{C}_1$ are committed, C_2 is being updated, and C_3 remains unmodified. The undo buffer holds C_2's pre-update value. If a HOF interrupts C_2's update, the parity buffer encodes the last committed state, $RS(\overline{C}_0, \overline{C}_1, C_2, C_3)$, whose data remain in the protection group or undo buffer. Within RMP's configured tolerance, RMU can reconstruct failed pages from this data and parity, restore C_2 from the undo buffer, and resume the update.

Compute and update phases. Pseudocode 1 gives the complete iteration with RMU. The compute phase begins with the regular forward and backward passes to compute gradients (line 1). RMU then protects each group's gradients with parity (lines 2–3) before the rank enters the global synchronization (line 4). This ordering ensures all ranks' gradients are recoverable before any model state changes, preserving the inputs needed to finish an interrupted update.

After synchronization, RMU begins the update phase. For each protection group, it copies the host-resident parity pages into a temporary GPU parity buffer (line 6). For each chunk, it preserves the pre-update model state in the undo buffer (line 8) and performs the in-place update (line 9). It then computes updated parity in a scratch buffer using the chunk delta (line 10). Committing the chunk makes this scratch parity authoritative and swaps the parity and scratch buffers (line 11). Once all chunks in the group are committed, RMU copies the resulting parity back to host memory (line 12) and refreshes its second-tier parity.

RMU records its progress in a small host-memory marker that RMP allocates with parity protection. Lightweight CUDA kernels update the marker in the same stream after each RMU GPU operation. A promoted trainer uses this marker to identify the last completed operation and the authoritative parity buffer, allowing it to resume the interrupted update correctly. All ranks retain their committed progress and finish the remaining chunks to complete the same update.

Pseudocode 1: One training iteration with RMU

Input: For $0 \le i < n$, protection group P_i holds k chunks C_j with gradients G_j, $0 \le j < k$, host parity pages H_i, parity buffer Q_i, and gradient parity buffer R_i; scratch parity buffer Q'; undo buffer U.

1 Compute gradients
2 **foreach** *protection group* P_i **do**
3 $\quad R_i \leftarrow \mathsf{RS}(G_0, \ldots, G_{k-1})$ `// Protect gradients`
4 Global synchronization
5 **foreach** *protection group* P_i **do**
6 $\quad Q_i \leftarrow$ `Host-to-GPU`(H_i)
7 $\quad$ **foreach** *chunk* C_j *in* P_i **do**
8 $\quad\quad U \leftarrow C_j$ `// Preserve pre-update chunk`
9 $\quad\quad C_j \leftarrow$ `Update`(C_j, G_j) `// In-place chunk update`
10 $\quad\quad Q' \leftarrow Q_i \oplus \mathsf{RS}_j(U \oplus C_j)$ `// Parity update`
11 $\quad\quad$ `Commit`(C_j, Q') and `swap`(Q_i, Q')
12 $\quad H_i \leftarrow$ `GPU-to-Host`(Q_i)

Minimizing memory and transfer overhead. Activation buffers and RMU's temporary recovery buffers are needed at different times, allowing them to share GPU memory. RMU allocates this memory through RMP without separate parity protection because chunk-level commits allow recovery to the last committed chunk boundary after page failures. During the compute phase, RMU lends this memory to the trainer for activations. Once backward propagation finishes, it reuses the memory for its gradient-parity, undo, and parity buffers. To reduce transfer overhead, RMU also pipelines protection groups, overlapping the next group's host-to-GPU transfer and the previous group's GPU-to-host transfer with the current group's computation. These techniques enable crash-consistent updates without increasing peak HBM usage, with only 0.6% fault-free iteration overhead (Table 1).

6 OOM-Free Preinitialization

RMP and RMU keep retained state recoverable and consistent, but the process state prebuilt by a shadow trainer consumes GPU memory alongside active training. We design the *preinitialization controller* to reclaim this state when active training needs the memory.

Dynamic memory pressure. A shadow trainer may initially fit in GPU memory but later cause an OOM as active memory demand varies, particularly in MoE models where uneven expert routing changes per-GPU token loads [52, 81]. Figure 7 illustrates this effect on OLMoE. The active trainer alone remains below the 80 GiB GPU capacity, but its shadow adds 2.2 GiB of process state, comprising a 494 MiB CUDA context, 1,512 MiB of NCCL communicator metadata, and 196 MiB of runtime-loaded modules (*e.g.*, cuBLAS [50]). The active trainer then crashes with an OOM error.

Our approach. Prebuilt process state accelerates recovery but can be discarded without losing training progress. We exploit this property by integrating reclamation into the active trainer's memory allocation path. When a request cannot

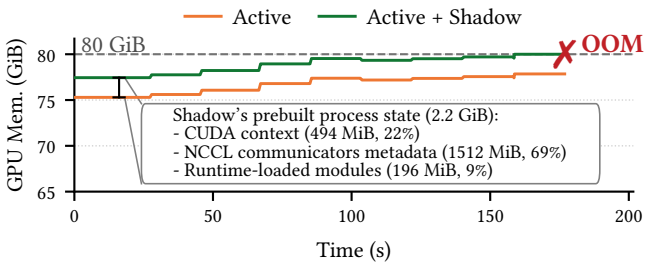


Figure 7. GPU memory usage on OLMoE with global batch size 182. The shadow trainer's prebuilt state causes an OOM in active training after 177 s.

be satisfied, the allocator (*e.g.,* PyTorch's CUDACachingAllocator [47]) invokes the preinitialization controller before reporting an OOM. The controller synchronously terminates the shadow trainer and waits for its GPU memory to be released. The allocator then retries the request, transparently avoiding an OOM caused by the shadow.

To avoid repeatedly rebuilding and reclaiming the same state, the controller adjusts how far the shadow preinitializes. It supports three *preinitialization levels*: (L1) before creating GPU state, (L2) after GPU runtime initialization but before NCCL communicator creation, and (L3) fully preinitialized. L2 omits NCCL metadata, which accounts for 69% of the shadow's footprint in Figure 7. After reclamation, the controller relaunches the shadow at the most advanced level that fits the remaining memory headroom. Further reclamations can only lower this level until the next HOF recovery, when the shadow completes any deferred initialization. After recovery, the controller uses the observed peak active-training memory to raise the new shadow's level when sufficient headroom is expected, reducing future recovery time.

7 Implementation

We implemented Rivet on Megatron-Core v0.13.1 [46], PyTorch 2.8.0 [60], and NCCL 2.27.3 [49] in approximately 7.8K lines of Python and 2.4K lines of C++/CUDA.

Retained state allocation. We extend Megatron's model-state tensor allocation path to determine tensor shapes and sizes without allocating GPU memory, using PyTorch's meta device [59]. Based on this metadata, trainers request allocation from RMP over Unix domain sockets. RMP allocates GPU memory using the CUDA virtual-memory API [48] and host memory using memfd_create, and transfers the resulting POSIX file descriptors in batches using SCM_RIGHTS [28]. Trainers then import the shared memory and expose it as PyTorch tensors through zero-copy DLPack views [10]. We also extend NCCL's buffer-allocation path so communicator buffers can be allocated from RMP.

Erasure coding computation. RMU's Reed–Solomon operations for parity-page encoding and erased-page reconstruction are implemented as CUDA kernels over the finite field GF(2^{32}), using carry-less multiply instructions (clmad [55]). Tiled asynchronous memory movement and lazy finite-field reduction overlap data movement with computation and maximize clmad throughput.

PyTorch memory allocator integration. We implement a custom GPU memory pool in the Megatron runtime and register it with PyTorch's CUDACachingAllocator [47]. The pool reserves temporary buffers required by RMU while allowing the allocator to use this memory for activation buffers during the compute phase. We also add an OOM callback to CUDACachingAllocator; before the allocator reports an OOM, the callback coordinates with the preinitialization controller to reclaim the shadow trainer's GPU memory, after which the allocator retries the failed allocation.

Table 2. Models and parallelism configurations used in our evaluation. DP, TP, PP, and EP denote the degrees of data, tensor, pipeline [44], and expert [64] parallelism, respectively. *Opt. states sharded* indicates whether optimizer states are sharded across data-parallel ranks [65, 80].

Model	Total params	Active params	Parallelism				Opt. states sharded
			DP	TP	PP	EP	
Qwen [76]	4.0B	4.0B	2	2	1	1	✗
OLMoE [43]	6.9B	1.3B	2	2	1	4	✓
DeepSeek-MoE [12]	16.4B	2.8B	8	8	1	16	✗
GPT-MoE [56]	20.9B	3.6B	8	8	1	16	✓
Kimi-MoE [68]	49.1B	3.1B	4	4	4	16	✓

8 Evaluation

8.1 Experimental Setup and Methodology

Testbed. Our AWS cluster has 9 EC2 p4d.24xlarge nodes, each with 8 NVIDIA A100 40 GB GPUs, 96 vCPUs, 1,152 GiB host memory, and four 100 Gbps EFA interfaces (NCCL via aws-ofi-nccl [3]). Each node in our 3-node private cluster has 2 NVIDIA A100 80 GB PCIe GPUs, two Intel Xeon Silver 4314 CPUs (32 physical cores), 256 GiB host memory, and a 100 Gbps RoCE NIC with GPUDirect RDMA. Both use 600 GB/s intra-node NVLink, Linux 6.8.0, NVIDIA driver 580.173.02, and the NGC PyTorch 25.05 container (Python 3.12.3, CUDA 12.9). Spare-based recovery reserves one warm spare per cluster; the replaced node becomes the next spare.

Workloads. We evaluate five dense and MoE LLMs with varied sizes and parallelism strategies (Table 2). We train Qwen and OLMoE on the private cluster with sequence length 512, and the remaining models on AWS with sequence length 2048. Unless specified otherwise, we use a global batch size of 128; §A.1 lists other defaults.

Baselines. We compare Rivet against three in-memory checkpointing systems, Gemini [73], MoEvement [18], and JIT-Ckpt [21]. We run MoEvement only on MoE models, for which its sparse checkpointing is designed. JIT-Ckpt requires replicated optimizer state for recovery, so we evaluate it on

Qwen and DeepSeek-MoE, whose optimizer states are unsharded across DP ranks. Each baseline also has a "+" variant with spare-based preinitialization (*e.g.,* Gemini+; Figure 2b).

Recovery policy. The three baselines recover in place on a HOF and use spares on an FF; their "+" variants use spares in both cases. Rivet recovers in place on a HOF and adopts Gemini+ for FFs on OLMoE, GPT-MoE, and Kimi-MoE and JIT-Ckpt+ on Qwen and DeepSeek-MoE.

Checkpointing interval. The *checkpoint interval* is the number of iterations over which the full model state is checkpointed (*e.g.,* MoEvement's window [18], and 1 for JIT-Ckpt, which relies on redundancy in the model state). We select checkpoint intervals per workload, MTTF, and system (§A.2). When Rivet uses Gemini+ as its underlying checkpointing system, only FFs require rollback to a checkpoint. Given an overall MTTF M and HOF ratio h, the FF rate is $(1-h)/M$, yielding an FF MTTF of $M/(1-h)$. We substitute this FF MTTF into Young's formula [77] to estimate the optimal checkpoint interval and round it to a whole number of iterations. At $h = 1$, the formula gives no finite optimal interval, so we use the largest interval tested in each setting.

Failure injection. Each run follows a predefined schedule of injection times, target ranks, and failure classes (HOF or FF), which determine recovery placement. We send SIGKILL to the target rank; a monitor detects termination in under 1 s in our runs and initiates recovery.

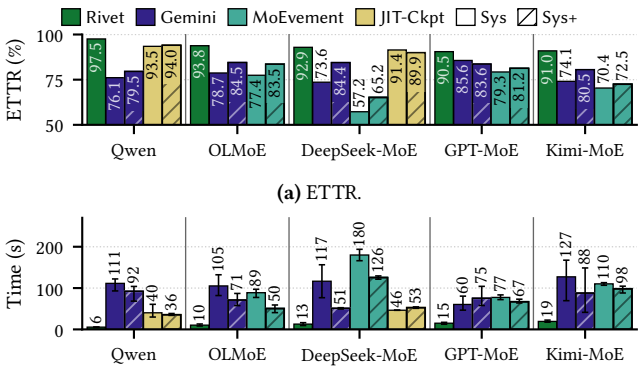
Metrics. We define ETTR and recovery time as follows.

ETTR (%): The fraction of wall-clock time spent on productive training (§2.3), measured as ETTR $= T_{\text{ref}}(i)/T_{\text{eval}}(i)$ for the evaluated run's last completed iteration i. Both times span the first iteration's start through iteration i's completion; the reference run is otherwise identical but has no fault-tolerance mechanisms or injected faults.

Recovery time (s): The time from failure injection until the job recovers its pre-failure training progress, comprising initialization and recomputation. For a failure at t_{fault} interrupting iteration $i+1$, let t_j denote when training resumes, t_{i+1} when that iteration completes again, and t_{remain} its fault-free remaining time at failure. We measure initialization as $I = t_j - t_{\text{fault}}$, recomputation as $R = t_{i+1} - t_j - t_{\text{remain}}$, and recovery as $I + R$.

8.2 End-to-End Performance

Overall performance. We run all applicable systems on the five workloads for 30 minutes, injecting one HOF every 10 minutes (Figure 8). Rivet uses the same checkpoint intervals as Gemini and Gemini+ for FF recovery on OLMoE, GPT-MoE, and Kimi-MoE. We examine the impact of longer checkpoint intervals for Rivet below. Rivet sustains 91–98% ETTR, exceeding each workload's best baseline by 1.5–10.4 percentage points. Rivet recovers in 6–19 s, 3.6–6.5× faster than the fastest baseline for each workload (Figure 8b). Baselines replay from checkpoints; Rivet reuses the live model state, repeating at most the interrupted iteration.



(a) ETTR.

(b) Mean recovery time with min–max error bars.

Figure 8. Performance with one HOF every 10 minutes. Hatched bars denote spare-based preinitialization ("+"); missing bars indicate unsupported workloads.

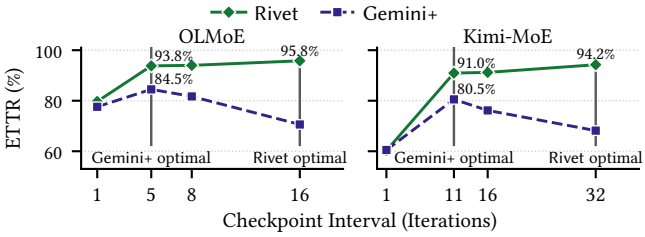


Figure 9. ETTR of Rivet and Gemini+ on OLMoE and Kimi-MoE with one HOF every 10 minutes, varying the checkpoint interval; vertical lines mark each system's ETTR-maximizing interval.

Impact of the checkpoint interval. Figure 9 examines sensitivity to the checkpoint interval on OLMoE (1, 5, 8, and 16 iterations) and Kimi-MoE (1, 11, 16, and 32) with only HOFs. Gemini+ peaks at intervals of 5 and 11, reaching 84.5% and 80.5% ETTR, respectively; shorter intervals incur more checkpointing overhead, while longer intervals increase recomputation.

Rivet retains checkpoints for FFs, but recovers from HOFs without rollback. In this HOF-only experiment, longer intervals therefore reduce checkpointing overhead without increasing recomputation. Rivet's ETTR rises from 93.8% to 95.8% on OLMoE as the interval grows from 5 to 16, and from 91.0% to 94.2% on Kimi-MoE as it grows from 11 to 32. At these intervals, Rivet exceeds Gemini+'s peak ETTR by 11.3 and 13.7 percentage points, respectively.

Impact of the HOF ratio. We next vary the HOF ratio from 0% (all FFs) to 100% on OLMoE (Figure 10), comparing against Gemini+, its best baseline in Figure 8. Gemini+ uses its tuned interval of 5; we reuse its measured 84.5% ETTR across ratios because both failure classes follow the same recovery path. Using the interval-estimation method in §8.1, we set Rivet's interval to 5, 6, and 9 iterations at 0%, 33.3%, and 66.7% HOFs; at 100%, we use the largest tested interval of

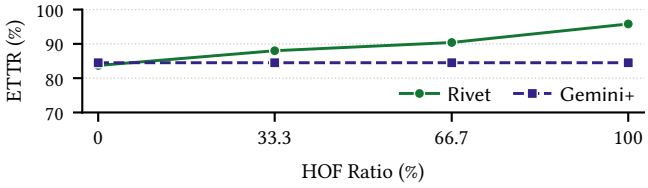


Figure 10. ETTR of Rivet and Gemini+ on OLMoE with one failure every 10 minutes, varying the HOF ratio.

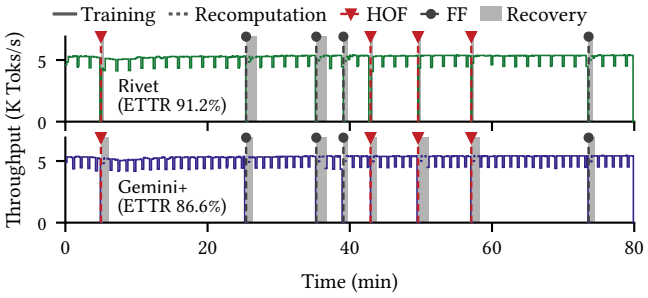


Figure 11. Throughput on OLMoE with failure arrivals from a public trace and a 50% HOF ratio. Shading marks recovery, dotted segments recomputation, and periodic notches checkpointing.

16. Its ETTR rises from 83.7% to 95.8%, exceeding Gemini+ by 3.5 percentage points even at a 33.3% HOF ratio. With only FFs, its steady-state overhead costs 0.8 percentage points.

Realistic failures with a public trace. To emulate irregular training failures, we schedule injections using eviction times from an 80-minute segment of a public GCP A100 spot-instance trace used in prior work [18, 24, 69]. The segment contains eight eviction events, averaging one every 10 minutes. We inject a training failure at each event, randomly assigning four as HOFs and four as FFs.

Figure 11 compares Rivet and Gemini+ on OLMoE at their optimal checkpoint intervals of 8 and 5, respectively. Rivet achieves 91.2% ETTR versus 86.6% for Gemini+ and reduces total recovery time from 7.8 to 4.6 minutes, or from 59 to 34 s per failure on average. Its HOF recovery averages 11 s, while FF recovery averages 57 s using Gemini+. Fast HOF recovery therefore improves training efficiency even with irregular arrivals and checkpoint rollback for half the failures.

8.3 Performance Breakdown

Table 3 separates recovery on Kimi-MoE into initialization and recomputation to show where Rivet saves time. Spare-based preinitialization reduces initialization from 59 s to 35 s (Gemini to Gemini+). RMP and RMU reduce recomputation from 53 s to 5 s. RMP retains the live model state to avoid checkpoint replay, while RMU resumes an interrupted update from its last committed chunk. In-place preinitialization further reduces initialization from 35 s to 15 s by letting the shadow take over with state prebuilt for the same hardware and rank. At the same checkpoint interval of 11, these mechanisms yield 4.5× faster recovery than Gemini+ (88 s to 19 s) and raise ETTR from 80.5% to 91.0%. Because only FFs require checkpoint rollback, Rivet can further reduce checkpointing overhead by extending its interval to 32, reaching 94.2% ETTR with 16 s recovery.

Table 3. Recovery-time breakdown and ETTR on Kimi-MoE for Gemini, Gemini+, and Rivet at the indicated checkpoint intervals. Interval 11 maximizes ETTR for Gemini and Gemini+, while 32 does so for Rivet (Figure 9).

System	Recovery	Init.	Recomp.	ETTR
Gemini (interval=11)	127 s	59 s	68 s	74.1%
Gemini+ (interval=11)	88 s	35 s	53 s	80.5%
Rivet (interval=11)	19 s	15 s	5 s	91.0%
Rivet (interval=32)	**16 s**	**13 s**	**3 s**	**94.2%**

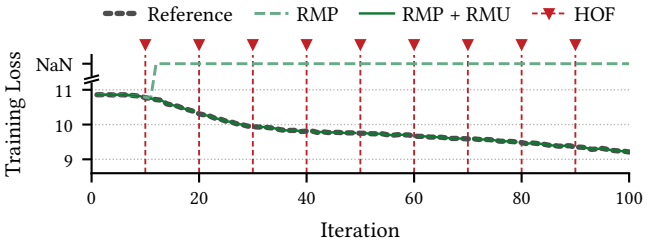


Figure 12. Training loss of OLMoE over 100 iterations for the fault-free reference, RMP alone, and RMP with RMU, with a HOF injected during the update phase every 10 iterations.

8.4 Recovery Correctness

We verify that Rivet recovers exactly from HOFs that interrupt the update phase and lose part of the retained state. We train OLMoE for 100 iterations and, every 10 iterations, kill one rank during its model update, interrupting the optimizer kernel, the parity-encoding kernel, or the parity copy in turn; we instrument the kernels with NVBit [70] to abort mid-execution and fill the copy's destination with random values. After each failure, we emulate uncorrectable HBM errors by unmapping retained RMP pages with `cuMemUnmap`, 16 per protection group, the maximum its parity tolerates, and a host-memory error on one parity page by poisoning it with `madvise(MADV_HWPOISON)` [32], following prior work [78].

Figure 12 shows the training loss. With RMP alone, the parity is inconsistent with the partially updated model state, so the state reconstructed after the first HOF is wrong: the loss diverges from the reference at the next iteration and becomes NaN thereafter. With RMP and RMU, Rivet recovers exactly from every injected HOF, reproducing the fault-free reference loss. Per recovery, RMP localizes failures in 36 ms and RMU reconstructs pages in 140 ms.

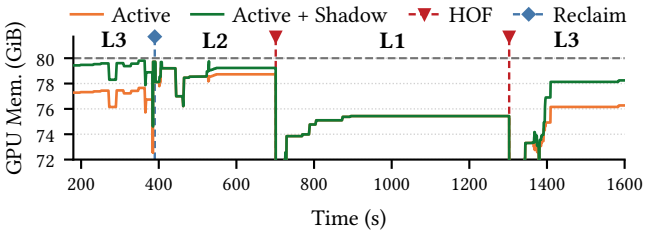


Figure 13. GPU memory usage of Rivet on OLMoE under memory pressure. The gap between the curves is the shadow's footprint. Levels denote full preinitialization (L3), GPU runtime only (L2), and no GPU state (L1).

8.5 Recovery under Memory Pressure

We evaluate whether the preinitialization controller (§6) prevents OOMs from shadow memory use while preserving recovery benefits. The *preinitialization level* specifies how far the shadow trainer initializes before a failure. We train OLMoE at global batch size 182 and inject one HOF every 10 minutes for an hour. Without runtime reclamation, the fully preinitialized shadow causes an OOM (Figure 7).

As Figure 13 shows for the first three HOFs of the run, the controller adapts the shadow to available memory. When memory pressure rises, it reclaims the 2.2 GiB L3 shadow and relaunches it at L2. If L2 no longer fits, the controller lowers the shadow to L1; it restores L3 once pressure subsides. Despite these transitions, every HOF recovers through a shadow, taking 26–27 s from L3, 33–34 s from L2, and 52 s from L1. Over the hour, Rivet achieves 91.8% ETTR, 4.1 percentage points above Gemini+: reclamation gives up only as much preinitialization as memory pressure demands, and HOF recovery never rolls back to a checkpoint.

8.6 Large-Scale Simulation

We evaluate scalability beyond our testbed using a simulator that extrapolates single-stage profiles to predict iteration time, recovery time, and ETTR under a given failure schedule. We validate the simulator against training performance measured on our GPU testbed, with ETTR prediction errors of 3.1% and 0.7% for Gemini+ on DeepSeek-MoE and Kimi-MoE, respectively. We will open-source the simulator. See Appendix C for details.

We increase model and cluster size together, simulating DeepSeek-V2 [13] (236B total, 21B active parameters) on 1,024 GPUs, DeepSeek-V3 [14] (671B total, 37B active) on 16,384 GPUs, and DeepSeek-V4 [15] (1.6T total, 49B active) on 131,072 GPUs. For each configuration, we set the MTTF to 10 minutes and vary the HOF ratio over 0%, 50%, and 100%.

As Figure 14 shows, Rivet's advantage grows with the HOF ratio at every scale. With 100% HOFs, Rivet sustains 95–96% ETTR, compared with 83.0–85.3% for Gemini+. Even at 50% HOFs, Rivet improves ETTR by 3.5–4.3 percentage points; with no HOFs, it trails by just 0.3 percentage points.

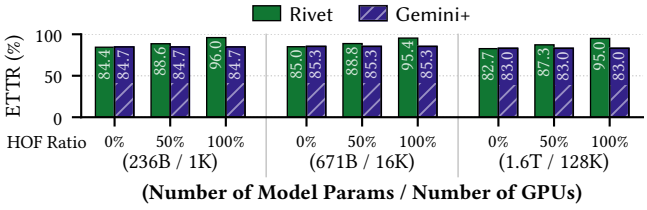


Figure 14. Simulated ETTR of Rivet and Gemini+ with a 10-minute MTTF, varying the HOF ratio across model and cluster sizes.

These results suggest that state reuse remains beneficial at scale even when HOFs account for only half of all failures. See Appendix B for results at 30-minute and 1-hour MTTFs.

9 Related Work

Checkpointing systems. Gemini [73] keeps checkpoints in remote host memory, while MoEvement [18] exploits MoE sparsity for lightweight checkpointing and localized recovery. JIT-Checkpointing [21] requires replicated optimizer state, precluding its use with optimizer-state sharding in ZeRO and FSDP [65, 80]. CheckFreq [42] adapts checkpoint frequency and overlaps checkpointing with training, while PCcheck [67] pipelines transfers to persistent storage and supports concurrent checkpoints. MoC-System [6] checkpoints a subset of experts; ByteCheckpoint [71] and Universal Checkpointing [35] support recovery under different parallelism configurations. Checkmate [5] maintains per-iteration recovery state using dedicated CPU nodes and programmable switches. Rivet instead reuses working state for in-place HOF recovery without loading checkpoints.

Runtime reconfiguration and redundant computation. Runtime reconfiguration [4, 17, 19, 24] tolerates resource loss by adapting training to fewer nodes. Rivet instead exploits surviving hardware to preserve the job's scale and avoid reconfiguration costs [34]. Bamboo [69] avoids checkpoint rollback through redundant computation for neighboring pipeline stages, but incurs steady-state overhead [24]. Rivet requires no redundant training computation.

Erasure protection for memory-resident state. Prior systems use checksums and erasure coding to protect persistent-memory and memory-resident data [63, 75, 78, 82], some maintaining consistent redundancy across transactions [63, 78]. Rivet protects GPU-resident model state without storing persistent parity in HBM. ECRM [79] erasure-codes embedding tables with sparse updates to limit network overhead, but replicates dense parameters. Rivet keeps model state and parity consistent across dense optimizer updates.

10 Conclusion

We presented Rivet, which decouples training-state and process lifetimes to recover from HOFs through state reuse and advance preparation on surviving hardware. Resilient state retention, crash-consistent updates, and reclaimable

preinitialization deliver 3.6–6.5× faster recovery than the best-performing checkpointing baselines and up to 13.7 percentage points more productive training time. Rivet opens new opportunities for accelerator runtimes to preserve application progress across process failures.

References

[1] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. Vidur: A Large-Scale Simulation Framework For LLM Inference. arXiv:2405.05465 [cs.LG] https://arxiv.org/abs/2405.05465

[2] Eduardo Alvarez, Vishal Mehta, and Farshad Ghodsian. 2026. Inside NVIDIA Rubin GPU Architecture: Powering the Era of Agentic AI. NVIDIA Technical Blog. Published July 21, 2026; accessed August 1, 2026. https://developer.nvidia.com/blog/inside-nvidia-rubin-gpu-architecture-powering-the-era-of-agentic-ai/

[3] Amazon Web Services. 2026. Get started with EFA and NCCL for ML workloads on Amazon EC2. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/efa-start-nccl.html. Accessed August 12, 2026.

[4] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 472–487. doi:10.1145/3492321.3519584

[5] Ankit Bhardwaj, Weiyang Wang, Jeremy Carin, Adam Belay, and Manya Ghobadi. 2026. Checkmate: Zero Performance Overhead Model Checkpointing via Network Gradient Replication. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 339–358. https://www.usenix.org/conference/nsdi26/presentation/bhardwaj

[6] Weilin Cai, Le Qin, and Jiayi Huang. 2025. MoC-System: Efficient Fault Tolerance for Sparse Mixture-of-Experts Model Training. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 655–671. doi:10.1145/3676641.3716006

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. (2016). arXiv:1604.06174 [cs.LG] https://arxiv.org/abs/1604.06174

[8] Jaehong Cho, Hyunmin Choi, Guseul Heo, and Jongse Park. 2026. LLMServingSim 2.0: A Unified Simulator for Heterogeneous and Disaggregated LLM Serving Infrastructure. arXiv:2602.23036 [cs.DC] https://arxiv.org/abs/2602.23036

[9] Jesse Clayton. 2026. NVIDIA NVLink: The Scale-Up Network for AI Factories. NVIDIA Technical Blog. Published July 20, 2026; accessed August 1, 2026. https://developer.nvidia.com/blog/nvidia-nvlink-the-scale-up-network-for-ai-factories/

[10] DLPack Community. 2025. Python Specification for DLPack. https://dmlc.github.io/dlpack/latest/. Accessed August 28, 2026.

[11] Shengkun Cui, Archit Patke, Hung Nguyen, Aditya Ranjan, Ziheng Chen, Phuong Cao, Gregory Bauer, Brett Bode, Catello Di Martino, Saurabh Jha, Chandra Narayanaswami, Daby Sow, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2025. Story of Two GPUs: Characterizing the Resilience of Hopper H100 and Ampere A100 GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25)*. Association for Computing Machinery, New York, NY, USA, 1145–1164. doi:10.1145/3712285.3759821

[12] Damai Dai, Chengqi Deng, Chenggang Zhao, Runxin Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y.K. Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. 2024. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1280–1297.

[13] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. arXiv:2405.04434 [cs.CL] https://arxiv.org/abs/2405.04434

[14] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma,

Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[15] DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chengyu Hou, Chenhao Xu, Chenze Shao, Chong Ruan, Conner Sun, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Donghao Li, Dongjie Ji, Erhang Li, Fang Wei, Fangyun Lin, Fangzhou Yuan, Feiyu Xia, Fucong Dai, Guangbo Hao, Guanting Chen, Guoai Cao, Guolai Meng, Guowei Li, Han Yu, Han Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoling Zhang, Haoming Luo, Haoran Wei, Haotian Yuan, Haowei Zhang, Haowen Luo, Haoyu Chen, Haozhe Ji, Hengqing Zhang, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, J Yang, JQ Zhu, Jia Luo, Jia Song, Jia Yu, Jialiang Huang, Jialu Cai, Jian Liang, Jiangting Zhou, Jiasheng Ye, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jieyu Yang, Jin Chen, Jin Yan, Jingchang Chen, Jingli Zhou, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jingzi Zhou, Jinhua Zhu, Jiping Yu, Joseph Sun, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junmin Zheng, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Leyi Xia, Li Zhang, Liang Zhao, Lihua Guo, Lingxiao Luo, Linwang Ma, Linyan Zhu, Litong Wang, Liyu Cai, Liyue Zhang, Longhao Chen, MS Di, MY Xu, Max Mei, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Mingxu Zhou, Minmin Han, Ning Wang, Panpan Huang, Panpan Wang, Peixin Cong, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Qiwei Jiang, Rui Tian, Ruifan Xu, Ruijie Lu, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqian Chen, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, Ruyi Chen, SH Liu, Shanghao Lu, Shangmian Sun, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoheng Nie, Shaoqing Wu, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Shuying Yu, Songyang Zhou, Tao Ni, Tao Yun, Tian Jin, Tian Pei, Tian Ye, Tianle Lin, Tianran Ji, Tianyi Cui, Tianyuan Yue, Tingting Yu, Tun Wang, W Zhang, WL Xiao, Wangding Zeng, Wei An, Weilin Zhao, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjing Yao, Wenjun Gao, Wenkai Yang, Wenlve Huang, Wenqing Hou, Wentao Zhang, Wenting Ma, Xi Gao, Xiang He, Xiangwen Wang, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingchen Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyu Zhang, Xu Chen, Xuanyu Wang, Xuecheng Su, Xueyin Chen, Xuheng Lin, Xuwei Fu, YC Yan, YQ Wang, YW Ma, Yanfeng Luo, Yang Zhang, Yanhong Xu, Yanru Ma, Yanwen Huang, Yao Li, Yao Li, Yao Xu, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Shao, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yijia Wu, Yiliang Xiong, Yiling Ma, Ying He, Ying Tang, Ying Zhou, Yingjia Luo, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiang Zhang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, YuKun Li, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuanhao Li, Yuduan Wang, Yuehan Yang, Yuer Xu, Yuhan Wu, Yuhao Meng, Yuheng Zou, Yukun Zha, Yunfan Xiong, Yupeng Chen, Yuping Lin, Yuqian Cao, Yuqian Wang, Yushun Zhang, Yuting Yan, Yutong Lin, Yuxian Gu, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuxuan Zhou, Yuyang Zhou, Yuzhen Huang, ZF Wu, Zehao Wang, Zehua Zhao, Zehui Ren, Zekai Zhang, Zhangli Sha, Zhe Fu, Zhe Ju, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zheren Gao, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhixuan Chen, Zhiyu Wu, Zhizhou Ren, Zhongyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihua Qu, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Ziyi Wan, Zizheng Pan, and Zongqing Yao. 2026. DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence. arXiv:2606.19348 [cs.CL] https://arxiv.org/abs/2606.19348

[16] Yangtao Deng, Xiang Shi, Zhuo Jiang, Xingjian Zhang, Lei Zhang, Zhang Zhang, Bo Li, Zuquan Song, Hang Zhu, Gaohong Liu, Fuliang Li, Shuguang Wang, Haibin Lin, Jianxi Ye, and Minlan Yu. 2025. Minder: Faulty Machine Detection for Large-scale Distributed Model Training. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 505–521. https://www.usenix.org/conference/nsdi25/presentation/deng

[17] Jiangfei Duan, Ziang Song, Xupeng Miao, Xiaoli Xi, Dahua Lin, Harry Xu, Minjia Zhang, and Zhihao Jia. 2024. Parcae: Proactive, Liveput-Optimized DNN Training on Preemptible Instances. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1121–1139. https://www.usenix.org/conference/nsdi24/presentation/duan

[18] Swapnil Gandhi and Christos Kozyrakis. 2026. Sparse Checkpointing for Fast and Reliable MoE Training. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 1395–1415. https://www.usenix.org/conference/nsdi26/presentation/gandhi

[19] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. 2024. ReCycle: Resilient Training of Large DNNs using Pipeline Adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 211–228. doi:10.1145/3694715.3695960

[20] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh,

Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[21] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. 2024. Just-In-Time Checkpointing: Low Cost Error Recovery from Deep Learning Training Failures. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1110–1125. doi:10.1145/3627703.3650085

[22] Tao He, Xue Li, Zhibin Wang, Kun Qian, Jingbo Xu, Wenyuan Yu, and Jingren Zhou. 2023. Unicron: Economizing Self-Healing LLM Training at Scale. arXiv:2401.00134 [cs.DC] https://arxiv.org/abs/2401.00134

[23] 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

[24] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. 2023. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 382–395. doi:10.1145/3600006.3613152

[25] Yuxuan Jiang, Ziming Zhou, Boyu Xu, Beijie Liu, Runhui Xu, and Peng Huang. 2025. Training with Confidence: Catching Silent Errors in Deep Learning Training with Automated Proactive Checks. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 313–329. https://www.usenix.org/conference/osdi25/presentation/jiang

[26] 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 *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'24)*. USENIX Association, USA, Article 41, 16 pages.

[27] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. 2019. A study of bfloat16 for deep learning training. *arXiv preprint arXiv:1905.12322* (2019).

[28] Michael Kerrisk. 2025. Linux manual page. https://man7.org/linux/man-pages/man7/unix.7.html. Accessed August 28, 2026.

[29] Xin Zhe Khooi, Zhuo Jiang, Pan Xie, Zhigang Cui, Meng Wang, Yuze Jin, Pengfei Huo, Dongyang Wang, Lulu Chen, Lei Wang, Liaoyuan Feng, Xiaodong Liu, Peng Li, Qinlong Wang, Yang Bai, Yongcan Wang, Hao Jin, Jinshuai Sun, Shan Lu, Xiang Shi, Yingkai Zhao, Haiquan Chen, Yi Li, Jianxi Ye, and Mun Choon Chan. 2026. Handling Network Faults in Distributed AI Training: Failover is Now an Option. 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, 263–278. doi:10.1145/3767295.3769322

[30] Joon Ha Kim, Geon-Woo Kim, Anoop Rachakonda, and Daehyeok Kim. 2026. Dooly: Configuration-Agnostic, Redundancy-Aware Profiling for LLM Inference Simulation. arXiv:2605.07985 [cs.DC] https://arxiv.org/abs/2605.07985

[31] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG] https://arxiv.org/abs/1412.6980

[32] Andi Kleen. 2009. hwpoison. The Linux Kernel documentation. Accessed: 2026-08-01. https://docs.kernel.org/mm/hwpoison.html

[33] Apostolos Kokolis, Michael Kuchnik, John Hoffman, Adithya Kumar, Parth Malani, Faye Ma, Zachary DeVito, Shubho Sengupta, Kalyan Saladi, and Carole-Jean Wu. 2025. Revisiting Reliability in Large-Scale Machine Learning Research Clusters . In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 1259–1274. doi:10.1109/HPCA61900.2025.00096

[34] ChonLam Lao, Jiaqi Gao, Jiamin Cao, Zhipeng Zhang, Pengcheng Zhang, Jiangfei Duan, Zhilong Zheng, Yu Guan, Yichi Xu, Yong Li, Zhengping Qian, Aditya Akella, Minlan Yu, Ennan Zhai, Dennis Cai, and Jingren Zhou. 2026. TrainMover: An Interruption-Resilient Runtime for ML Training. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. USENIX Association, Seattle, WA, 2047–2064. https://www.usenix.org/conference/osdi26/presentation/lao

[35] Xinyu Lian, Sam Ade Jacobs, Lev Kurilenko, Masahiro Tanaka, Stas Bekman, Olatunji Ruwase, and Minjia Zhang. 2025. Universal Checkpointing: A Flexible and Efficient Distributed Checkpointing System for Large-Scale DNN Training with Reconfigurable Parallelism. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, Boston, MA, 1519–1534. https://www.usenix.org/conference/atc25/presentation/lian

[36] Linux Kernel Developers. 2026. AMDGPU UMC v12.0 Memory Error Handling and Page Retirement. Linux kernel source code. Files `umc_v12_0.h` and `umc_v12_0.c`, commit 8934827; accessed August 28, 2026. https://github.com/torvalds/linux/tree/8934827db5403eae57d4537114a9ff88b0a8460f/drivers/gpu/drm/amd/amdgpu

[37] Linux man-pages project. 2026. madvise(2): Linux manual page. https://man7.org/linux/man-pages/man2/madvise.2.html. Linux man-pages 6.18. Accessed: 2026-09-03.

[38] Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, Yanru Chen, Huabin Zheng, Yibo Liu, Shaowei Liu, Bohong Yin, Weiran He, Han Zhu, Yuzhi Wang, Jianzhou Wang, Mengnan Dong, Zheng Zhang, Yongsheng Kang, Hao Zhang, Xinran Xu, Yutao Zhang, Yuxin Wu, Xinyu Zhou, and Zhilin Yang. 2025. Muon is Scalable for LLM Training. (2025). arXiv:2502.16982 [cs.LG] https://arxiv.org/abs/2502.16982

[39] Jeffrey Jian Ma, Hengzhi Pei, Leonard Lausen, and George Karypis. 2025. Understanding Silent Data Corruption in LLM Training. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 20372–20394. doi:10.18653/v1/2025.acl-long.996

[40] Bhushan Mehendale, Carlos Vallin, Drew Walton, Rama Bhimanadhuni, Venkatesh Ramamurthy, David Nieto, Linda Wu, Qian Wang, Philip Shirvani, Vishal Jain, Varinder Singh, Anil Agrawal, Balaji Vembu, Sukay Luhadia, Peter VanNess, Taniya Siddiqua, and Vilas Sridharan. 2025. *OCP GPU and Accelerators RAS Requirements*. Technical Report. Open Compute Project (OCP). https://www.opencompute.org/documents/ocp-gpu-and-accelerators-ras-requirements-v1-7-10-23-2025-pdf

[41] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *International Conference on Learning Representations*. https://openreview.net/forum?id=r1gs9JgRZ

[42] 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

[43] Niklas Muennighoff, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Jacob Morrison, Sewon Min, Weijia Shi, Pete Walsh, Oyvind Tafjord, Nathan Lambert, Yuling Gu, Shane Arora, Akshita Bhagia, Dustin Schwenk, David Wadden, Alexander Wettig, Binyuan Hui, Tim Dettmers, Douwe Kiela, Ali Farhadi, Noah A. Smith, Pang Wei Koh, Amanpreet Singh, and Hannaneh Hajishirzi. 2025. OLMoE: Open mixture-of-experts language models. In *International Conference on Learning Representations*.

[44] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 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.

[45] NVIDIA. 2025. cuBLAS: Results Reproducibility. https://docs.nvidia.com/cuda/cublas/index.html.

[46] NVIDIA. 2025. NVIDIA Megatron Core 0.13.1. https://github.com/NVIDIA/Megatron-LM/releases/tag/core_v0.13.1.

[47] NVIDIA Corporation. 2025. CUDA semantics. https://docs.pytorch.org/docs/2.8/notes/cuda.html. Accessed August 12, 2026.

[48] NVIDIA Corporation. 2025. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__VA.html. Accessed August 12, 2026.

[49] NVIDIA Corporation. 2025. NCCL 2.27.3 Release. https://docs.nvidia.com/deeplearning/nccl/release-notes/rel_2-27-3.html#rel_2-27-3. Accessed August 12, 2026.

[50] NVIDIA Corporation. 2026. *cuBLAS Library* (version 13.3 ed.). NVIDIA Corporation. Accessed: 2026-09-07. https://docs.nvidia.com/cuda/cublas/

[51] NVIDIA Corporation. 2026. *CUDA Driver API* (version 13.3.1 ed.). NVIDIA Corporation. Accessed: 2026-09-03. https://docs.nvidia.com/cuda/cuda-driver-api/

[52] NVIDIA Corporation. 2026. *Mixture of Experts Package* (megatron core developer guide, version 0.15.0 ed.). NVIDIA Corporation. Accessed: 2026-09-07. https://docs.nvidia.com/megatron-core/developer-guide/0.15.0/api-guide/moe.html

[53] NVIDIA Corporation 2026. *NVIDIA GPU Memory Error Management*. NVIDIA Corporation. Accessed August 28, 2026. https://docs.nvidia.com/deploy/a100-gpu-mem-error-mgmt/

[54] NVIDIA Corporation. 2026. NVIDIA Resiliency Extension. https://github.com/NVIDIA/nvidia-resiliency-ext. Accessed August 13, 2026.

[55] NVIDIA Corporation. 2026. *PTX ISA*. NVIDIA Corporation. Version 9.3. https://docs.nvidia.com/cuda/parallel-thread-execution/

[56] OpenAI, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily (Xiaoxuan) Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925* (2025).

[57] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28* (Atlanta, GA, USA) *(ICML'13)*. JMLR.org, III–1310–III–1318.

[58] Guilherme Penedo, Hynek Kydlicek, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. 2024. The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale. arXiv:2406.17557 [cs.CL] https://arxiv.org/abs/2406.17557

[59] PyTorch. 2025. Meta Device. https://docs.pytorch.org/docs/2.8/meta.html.

[60] PyTorch. 2025. PyTorch 2.8 Release Notes. https://github.com/pytorch/pytorch/releases/tag/v2.8.0.

[61] PyTorch. 2025. Reproducibility. https://docs.pytorch.org/docs/2.8/notes/randomness.html.

[62] PyTorch. 2026. torchrun (Elastic Launch). https://docs.pytorch.org/docs/stable/elastic/run.html. Accessed August 13, 2026.

[63] Han Jie Qiu, Sihang Liu, Xinyang Song, Samira Khan, and Gennady Pekhimenko. 2023. Pavise: Integrating Fault Tolerance Support for Persistent Memory Applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 109–123. doi:10.1145/3559009.3569662

[64] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale. In *International Conference on Machine Learning*. 18332–18346.

[65] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. IEEE Press. doi:10.1109/SC41405.2020.00024

[66] I. S. Reed and G. Solomon. 1960. Polynomial Codes Over Certain Finite Fields. *J. Soc. Indust. Appl. Math.* 8, 2 (1960), 300–304. arXiv:https://doi.org/10.1137/0108018 doi:10.1137/0108018

[67] Foteini Strati, Michal Friedman, and Ana Klimovic. 2025. PCcheck: Persistent Concurrent Checkpointing for ML. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 811–827. doi:10.1145/3669940.3707255

[68] Kimi Team, Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, Wentao Li, Enzhe Lu, Weizhou Liu, Yanru Chen, Weixin Xu, Longhui Yu, Yejie Wang, Yu Fan, Longguang Zhong, Enming Yuan, Dehao Zhang, Yizhi Zhang, T. Y. Liu, Haiming Wang, Shengjun Fang, Weiran He, Shaowei Liu, Yiwei Li, Jianlin Su, Jiezhong Qiu, Bo Pang, Junjie Yan, Zhejun Jiang, Weixiao Huang, Bohong Yin, Jiacheng You, Chu Wei, Zhengtao Wang, Chao Hong, Yutian Chen, Guanduo Chen, Yucheng Wang, Huabin Zheng, Feng Wang, Yibo Liu, Mengnan Dong, Zheng Zhang, Siyuan Pan, Wenhao Wu, Yuhao Wu, Longyu Guan, Jiawen Tao, Guohong Fu, Xinran Xu, Yuzhi Wang, Guokun Lai, Yuxin Wu, Xinyu Zhou, Zhilin Yang, and Yulun Du. 2025. Kimi Linear: An Expressive, Efficient Attention Architecture. arXiv:2510.26692 [cs.CL] https://arxiv.org/abs/2510.26692

[69] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2023. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 497–513. https://www.usenix.org/conference/nsdi23/presentation/thorpe

[70] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 372–383.

[71] 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

[72] 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. In

Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 186–203. doi:10.1145/3731569.3764838

[73] 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

[74] Yifan Xiong, Yuting Jiang, Ziyue Yang, Lei Qu, Guoshuai Zhao, Shuguang Liu, Dong Zhong, Boris Pinzur, Jie Zhang, Yang Wang, Jithin Jose, Hossein Pourreza, Jeff Baxter, Kushal Datta, Prabhat Ram, Luke Melton, Joe Chau, Peng Cheng, Yongqiang Xiong, and Lidong Zhou. 2024. SuperBench: Improving Cloud AI Infrastructure Reliability with Proactive Validation. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 835–850. https://www.usenix.org/conference/atc24/presentation/xiong

[75] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 478–496. doi:10.1145/3132747.3132761

[76] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).

[77] John W Young. 1974. A first order approximation to the optimum checkpoint interval. *Commun. ACM* 17, 9 (1974), 530–531.

[78] Lu Zhang and Steven Swanson. 2019. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 897–912. https://www.usenix.org/conference/atc19/presentation/zhang-lu

[79] Tianyu Zhang, Kaige Liu, Jack Kosaian, Juncheng Yang, and Rashmi Vinayak. 2023. Efficient Fault Tolerance for Recommendation Model Training via Erasure Coding. *Proc. VLDB Endow.* 16, 11 (July 2023), 3137–3150. doi:10.14778/3611479.3611514

[80] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3848–3860. doi:10.14778/3611540.3611569

[81] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M Dai, zhifeng Chen, Quoc V Le, and James Laudon. 2022. Mixture-of-Experts with Expert Choice Routing. 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., 7103–7114. doi:10.52202/068431-0515

[82] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 55–71. https://www.usenix.org/conference/osdi22/presentation/zhou-yang

A Evaluation Details

A.1 Additional Default Training Configuration

We train with BF16 mixed precision [27, 41] and apply activation checkpointing at the input to every layer [7]. Micro-batch sizes are 4 for the pipeline-parallel model and 16 for the others. For the pipeline-parallel model (*i.e.,* Kimi-MoE), we use interleaved one-forward-one-backward (1F1B) scheduling with two model chunks per pipeline-parallel rank [44]. For the other models, the global batch size of 128 is divided evenly across the DP ranks. We use Adam [31] with β_1=0.9, β_2=0.95, and weight decay 0.1. The learning rate warms up over the first 10% of iterations to a peak of 2.2×10^{-4}, then follows a cosine schedule to a floor of 2.2×10^{-5} (one tenth of the peak) by the end of the run. Gradients are clipped to a global norm of 1.0, weights are initialized from $\mathcal{N}(0,\ 0.006^2)$, and the attention softmax is computed in FP32. All runs train on FineWeb [58]. We enable the deterministic modes of PyTorch and CUDA libraries [45, 61] so that training is bit-reproducible and recovered runs can be compared exactly against the fault-free reference.

Table 4. Tuned checkpoint interval I and sparse window W, in iterations, per workload and MTTF. Dashes indicate that MoEvement is not evaluated on the dense workload, Qwen.

Workload	I (Gemini, Gemini+)			W (MoEvement, MoEvement+)		
	10 min	30 min	1 hr	10 min	30 min	1 hr
Qwen	22	37	53	–	–	–
OLMoE	5	9	12	2	4	4
DeepSeek-MoE	15	25	36	8	8	8
GPT-MoE	12	21	29	4	8	8
Kimi-MoE	11	19	27	4	8	8

A.2 Checkpoint Intervals

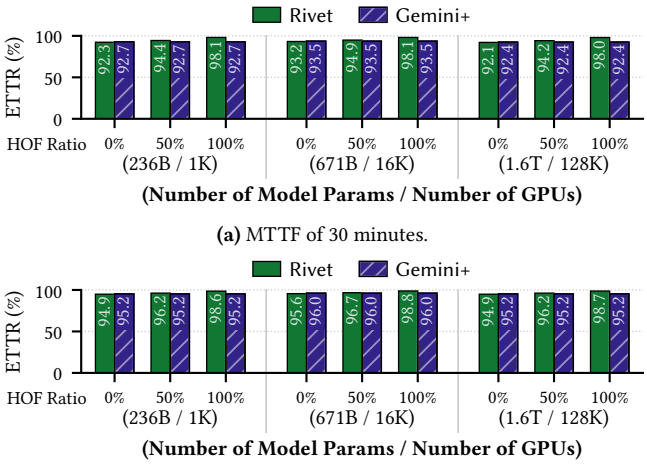
For interval-based checkpointing systems, we measure the normalized per-checkpoint cost p as the excess time of an iteration that commits a checkpoint over one that does not, divided by the uncheckpointed iteration time. We apply Young's formula [77] to estimate the checkpoint interval in iterations as $I^\star = \sqrt{2p\,\mathrm{MTTF}/t_{\mathrm{iter}}}$, where t_{iter} is the uncheckpointed iteration time. We use the same checkpoint-cost measurements for Gemini+ and Gemini, since they differ in recovery placement rather than checkpoint cost.

For window-based checkpointing systems, we select the window with the lowest estimated cost among those that fit in the pinned-memory pool, using a calibrated sparse-window model. JIT-Ckpt and JIT-Ckpt+ commit every iteration by construction ($I = 1$), on the two workloads whose optimizer states are replicated (Qwen and DeepSeek-MoE).

Rivet maintains checkpoints for FFs, while its HOF recovery does not load a checkpoint. In the HOF-ratio experiments, we estimate its checkpoint interval using the mean time between fatal failures, $M/(1-h)$, as described in §8.1. Table 4 lists baseline intervals; the evaluation specifies the intervals used in each experiment.

B Simulated ETTR at Longer MTTFs

We predict ETTR for Rivet and Gemini+ at MTTFs of 30 minutes and 1 hour, extending the results in Figure 14. As Figure 15 shows, Rivet's advantage again grows with the HOF ratio at every scale: it leads Gemini+ by 0.7–1.8 percentage points at 50% HOFs and by 2.8–5.6 points at 100%, and trails by 0.3 to 0.4 points with 0% HOFs due to its steady-state overhead.



(a) MTTF of 30 minutes.

(b) MTTF of 1 hour.

Figure 15. Simulated ETTR at longer MTTFs, using the model and cluster configurations in Figure 14.

C Simulation Details

This section describes the simulator used to predict ETTR for the large-scale training scenarios in Figure 14. Table 5 lists the model and parallelism configurations used in these simulations.

Table 5. Models and parallelism configurations used in our large-scale simulations. DP, TP, PP, and EP denote the degrees of data, tensor, pipeline [44], and expert [64] parallelism, respectively. *Opt. states sharded* indicates whether optimizer states are sharded across data-parallel ranks [65, 80].

Model	Total params	Active params	Parallelism				Opt. states sharded
			DP	TP	PP	EP	
DeepSeek-V2 [13]	236B	21B	8	8	16	16	✓
DeepSeek-V3 [14]	671B	37B	128	8	16	64	✓
DeepSeek-V4 [15]	1.6T	49B	1024	8	16	64	✓

C.1 Modeling Iteration Time

To predict a training system's ETTR, the simulator must accurately model its iteration time (T_{iter}). We take inspiration

from existing simulators for LLM workloads [1, 8, 30] and use a profile-based approach to simulation. Given a deployment scenario and model, we profile the training job at the granularity of a single pipeline stage and use the profile to predict the iteration time of the entire job.

We model iteration time as the sum of the pipeline, synchronization, and update times. These terms correspond to forward and backward propagation under the pipeline schedule, data-parallel gradient synchronization, and the optimizer step, respectively.

$$T_{\text{iter}} = T_{\text{pipeline}} + T_{\text{sync}} + T_{\text{update}}$$

C.1.1 Pipeline Time (T_{pipeline}). T_{pipeline} covers the scheduled forward and backward passes. Its profiling unit is the *bracket*, a contiguous chunk of layers within one pipeline stage, traversed forward and backward by a single microbatch. We measure each bracket in isolation using the deployment's hardware placement, including all intra-layer communication (tensor-parallel collectives and MoE all-to-all) in the timed interval. Given bracket times, we estimate the pipeline time as

$$T_{\text{pipeline}} = \left(M + \frac{S-1}{v}\right) \cdot \max_{s} \text{stage}(s)$$

Here, M is the number of microbatches, S is the number of pipeline stages, and v is the virtual-pipeline interleaving degree. stage(s) sums the times for stage s's brackets and inter-stage point-to-point transfers, plus the embedding on the first stage or the output head and loss on the last stage. We estimate transfer times from a fitted send/receive curve at the profiled activation size; the isolated bracket measurements exclude these transfers. Taking the maximum captures the bottleneck imposed by the slowest concurrent stage.

C.1.2 Synchronization and Update Time (T_{sync}+T_{update}). T_{sync} is the time for data-parallel gradient synchronization, modeled using collective operations that we profile in isolation and fit to a parametric model. We also profile concurrent collectives to capture the cost of two collectives sharing the same network fabric.

T_{update} is the optimizer-step time, modeled using machine and network constants so that no per-model sweep is required. We profile the optimizer step and scalar reduction in isolation and fit their timings to a parametric model. For sharded optimizers, we also profile and model the all-gather that reconstructs the full parameter tensor for the next forward pass.

C.1.3 Communication–Computation Overlap. Overlapping gradient synchronization with backward computation affects T_{iter}, but individual timers do not directly capture the resulting interference. We therefore measure interference differentially, running the same configuration with overlap disabled and enabled. The measured saving is $saving = T^{\text{off}}_{\text{iter}} - T^{\text{on}}_{\text{iter}}$, while a lightweight event logger in the enabled run records the exposed communication time, E. Because the two runs differ only in the overlap setting, we decompose the serial synchronization time T_{sync} into time saved, exposed communication, and residual interference.

$$T_{\text{sync}} = saving + E + R.$$

The residual R captures the additional cost of resource contention between the in-flight collective and backward computation.

Modeling E. We replay the framework's own overlap mechanism as a queue, with no fitted parameter. Backward walks the stage's layers in reverse over a window of length W, taken from the bracket profile of §C.1.1. Megatron packs gradients into fixed-size *buckets* of $\max(40\text{M}, 1\text{M} \cdot d_p)$ parameters, where d_p is the data-parallel degree and M denotes one million. A bucket's collective launches once backward propagation produces its last gradient. Bucket i has a transfer time w_i, obtained by evaluating the gradient-collective model of §C.1.2 at its byte size and multiplying by the measured concurrency penalty. We approximate its ready time as $r_i = \rho_i W$, where ρ_i is the fraction of the stage's parameters held by buckets $1, \ldots, i$, assuming backward progress is proportional to the number of parameters processed. The framework keeps one collective in flight, so buckets use a single serial channel. Bucket i finishes at $f_i = \max(r_i,\ f_{i-1}) + w_i$, with $f_0 = 0$. For n buckets, the communication remaining after the backward window is

$$E = \max(0,\ f_n - W).$$

Modeling R. We model R using a calibrated constant for each deployment class with interference, normalizing the measured residual by the quantity that drives it. For *dense* models, no other traffic shares the fabric during backward computation, so interference arises from contention with computation. We model this cost as proportional to the communication hidden under computation, $T_{\text{sync}} - E$, giving $c = R/(T_{\text{sync}} - E)$ and arrive at

$$T^{\text{on}}_{\text{sync}} = E \ + \ c \cdot (T_{\text{sync}} - E).$$

For *MoE* models whose expert groups span nodes, the gradient collective (G) and MoE all-to-all (A) contend for the same inter-node fabric. Two flows sharing a bottleneck finish no later than in series ($G + A$) and no earlier than the longer alone ($\max(G, A)$). This means that the delay attributable to sharing lies in $[0, \min(G, A)]$. The calibrated fraction is $f = R/\min(G, A)$, always lies within the range $[0, 1]$ for the profiled deployments. We therefore model the effective synchronization time as:

$$T^{\text{on}}_{\text{sync}} = E' \ + \ f \cdot \min(G, A),$$

where E' is the queue's exposure evaluated against the stretched window $W + f \cdot \min(G, A)$. MoE deployments

whose all-to-all stays within a node use no interference constant. When overlap is enabled, we substitute $T_{\text{sync}}^{\text{on}}$ for T_{sync} in the iteration-time model.

C.2 Iteration Time Prediction Accuracy

With the above methods, we predict the iteration time of the models in Table 2. As shown in Table 6, the predicted iteration times have a mean absolute percentage error of 2.18%.

Table 6. Simulator accuracy on iteration time. Predicted and measured T_{iter} (s) for every workload in Table 2, with communication–computation overlap enabled. Measurements are the median of three independent runs from iteration 11 onward.

	Qwen	OLMoE	DeepSeek-MoE	GPT-MoE	Kimi-MoE
Predicted (s)	3.464	11.939	5.742	5.198	9.246
Measured (s)	3.556	12.002	5.927	5.363	9.097
Error (%)	−2.6	−0.5	−3.1	−3.1	+1.6

C.3 ETTR Prediction

To predict ETTR, we model both the reference and evaluated execution times.

The reference run uses no fault-tolerance mechanisms and experiences no failures, so every iteration takes the unmodified iteration time T_{iter}.

$$T_{\text{ref}}(i) = i\,T_{\text{iter}}. \tag{1}$$

The evaluated run differs in two ways. Each iteration takes T_{eff} rather than T_{iter}, because fault-tolerance mechanisms add overhead even without failures. Each failure also incurs recovery time T_{rec}. Failures arrive at rate 1/MTTF in elapsed time, recovery included, so a run lasting T_{eval} experiences $T_{\text{eval}}/\text{MTTF}$ failures in expectation. We therefore model the elapsed time as

$$T_{\text{eval}}(i) = i\,T_{\text{eff}} \;+\; \frac{T_{\text{eval}}(i)}{\text{MTTF}}\,T_{\text{rec}}. \tag{2}$$

Collecting the terms in $T_{\text{eval}}(i)$ and solving gives

$$T_{\text{eval}}(i) - T_{\text{eval}}(i)\,\frac{T_{\text{rec}}}{\text{MTTF}} = i\,T_{\text{eff}}, \tag{3}$$

$$T_{\text{eval}}(i)\left(1 - \frac{T_{\text{rec}}}{\text{MTTF}}\right) = i\,T_{\text{eff}}, \tag{4}$$

$$T_{\text{eval}}(i) = \frac{i\,T_{\text{eff}}}{1 - T_{\text{rec}}/\text{MTTF}}. \tag{5}$$

Dividing Eq. 1 by Eq. 5 cancels the iteration count, giving

$$\begin{aligned}\text{ETTR} &= \frac{i\,T_{\text{iter}}}{i\,T_{\text{eff}}\,/\,(1 - T_{\text{rec}}/\text{MTTF})} \\ &= \frac{T_{\text{iter}}}{T_{\text{eff}}}\left(1 - \frac{T_{\text{rec}}}{\text{MTTF}}\right).\end{aligned} \tag{6}$$

The first factor captures the effect of steady-state overhead; the second captures the effect of recovery time.

Eq. 6 holds under four assumptions. (1) $T_{\text{rec}} < \text{MTTF}$: recovery is short relative to the time between failures, so the second factor stays positive and its linear form is accurate; the ratio reaches 0.3 at most in our evaluation. (2) Failures do not overlap: each is fully recovered before the next arrives, so every failure is charged one independent T_{rec}; no run in our evaluation contains a failure inside another's recovery. (3) Failures arrive uniformly at random in time, which gives the expected rollback of $I/2 + 1$ iterations for the interval-based systems, which overlaps checkpointing with next iteration's computation phase, and half an iteration for Rivet. (4) The run is long enough that the failure count is near its expectation and each recovery costs the same T_{rec}.

We decompose recovery time as $T_{\text{rec}} = T_{\text{det}} + T_{\text{init}} + L\,T_{\text{eff}}$ where T_{det} is the failure-detection time, T_{init} is the time to prepare a replacement trainer, and $L\,T_{\text{eff}}$ is the time to recompute L iterations of lost progress.

Eq. 6 applies to both systems, with the following system-specific terms.

$$\text{Gemini+:}\quad T_{\text{eff}} = T_{\text{iter}} + T_{\text{ckpt}}/I, \qquad L = \tfrac{I}{2} + 1, \tag{7}$$

$$\text{Rivet:}\quad T_{\text{eff}} = (1+\omega)\,T_{\text{iter}}, \qquad L = \tfrac{1}{2}. \tag{8}$$

During recovery, Gemini+ restores checkpointed state on a promoted standby and redoes the work since the newest usable checkpoint. A checkpoint is taken at the boundary of every I-th iteration but becomes usable only once its transfer completes, at the following iteration's step, so a uniformly arriving failure loses between 1 and I completed iterations, $(I+1)/2$ in expectation, plus the elapsed half of the iteration in flight: $L = I/2 + 1$. Rivet loses no completed iteration and only the in-flight half, $L = 1/2$.

Rivet keeps a checkpointing interval for FF recovery, so its steady cost is T_{ckpt}/I plus the per-iteration overhead of RMU ω. We estimate ω by profiling the duration of RMU's update phase and divide it by the estimated iteration time. Recovery promotes the shadow without transferring state; the only lost work is the elapsed portion of the interrupted iteration, half an iteration in expectation.