



USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Sparse Checkpointing for Fast and Reliable MoE Training

Swapnil Gandhi, *Stanford University*;
Christos Kozyrakis, *Stanford University and NVIDIA*

<https://www.usenix.org/conference/nsdi26/presentation/gandhi>

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

May 4-6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

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



جامعة الملك عبد الله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Sparse Checkpointing for Fast and Reliable MoE Training

Swapnil Gandhi
Stanford University

Christos Kozyrakis
Stanford University & NVIDIA

Abstract

As large language models scale, training them requires thousands of GPUs over extended durations—making frequent failures an inevitable reality. While checkpointing remains the primary fault-tolerance mechanism, existing methods fall short when applied to Mixture-of-Experts (MoE) models. Due to their substantially larger training state, MoE models exacerbate checkpointing overheads, often causing costly stalls or prolonged recovery that severely degrade training efficiency.

We present MoEevent, a distributed, in-memory checkpointing system tailored for MoE models. MoEevent is built on three key ideas: (1) *sparse checkpointing*, which incrementally snapshots subsets of experts across iterations to reduce overhead; (2) a *sparse-to-dense checkpoint conversion* mechanism that incrementally reconstructs consistent dense checkpoints from sparse snapshots; and (3) *upstream logging* of activations and gradients at pipeline-stage boundaries, enabling localized recovery without re-executing unaffected workers. Evaluations across diverse MoE models with up to 64 experts show that MoEevent reduces checkpointing overhead by up to $4\times$ and recovery overhead by up to $31\times$ compared to state-of-the-art approaches, sustaining ETTR ≥ 0.94 even under frequent failures (MTBF as low as 10 minutes) and delivering up to $8\times$ overall training speedup, all without compromising synchronous training semantics. Overall, MoEevent offers a robust and scalable fault-tolerance solution for the next generation of sparsely activated models.

1 Introduction

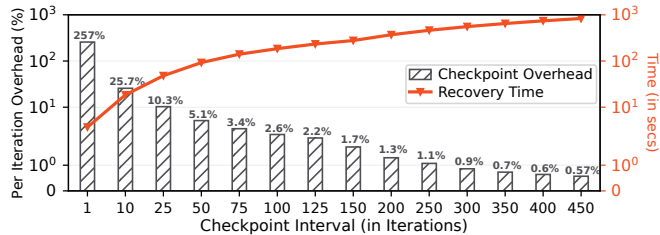
The rise of large foundation models has driven orders-of-magnitude growth in distributed training infrastructure [3, 20, 44, 58, 69–71, 90]. Training frontier models, such as DeepSeek-V3 with 671B parameters trained on 14.8 trillion tokens [47], requires thousands of GPUs over extended periods [5, 23, 29, 36, 59, 62]. At these scales, system failures—triggered by hardware faults, network disruptions, or software bugs—shift from being rare anomalies to frequent events. Major organizations including Microsoft, ByteDance, Alibaba, and Google have reported failures as frequently as once every 45 minutes, with the mean time between failures (MTBF) decreasing as GPU count rises [21, 26, 36, 40, 92]. Meta projects an MTBF as low as 14 minutes for jobs utilizing 131,072 GPUs [40], underscoring an emerging reality: *at scale, failure is not an anomaly—it is the norm.*

Checkpointing—periodically capturing model and optimizer states for durable persistence—is the standard fault-tolerance mechanism to limit computation lost to failures [9, 20, 21, 26, 36, 54, 62]. To reduce checkpoint overhead, recent techniques leverage overlapping snapshot operations with computation [51, 56, 78], high-performance interconnects [76, 80, 82], checkpoint compression [1, 15, 50], and redundancy across data-parallel nodes [22]. However, these approaches primarily target dense models, where all parameters contribute uniformly to computation and convergence.

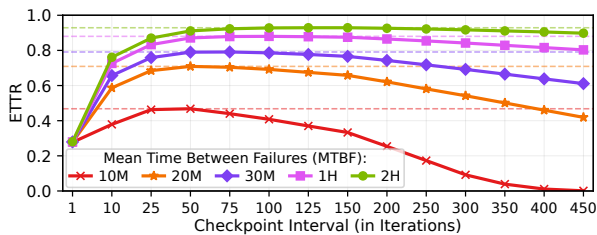
In contrast, Mixture-of-Experts (MoE) models follow a fundamentally different paradigm—replacing dense layers with tens to hundreds (occasionally millions [27]) of sparsely activated expert operators [12, 14, 16, 35]. Each token activates only a few experts (commonly four or eight out of hundreds [7]), enabling an order of magnitude larger parameter counts without proportional increases in computation [43, 49]. This sparse, dynamically varying operator activation pattern presents fundamental challenges for traditional checkpointing techniques. In particular, MoE models expose three key limitations in existing approaches:

Challenge #1: Runtime-Recovery Tradeoff: State-of-the-art checkpointing techniques—such as CheckFreq [56], which pipelines the *snapshot* (copying state to local CPU memory) and *persist* (flushing to durable storage) phases, and Gemini [82], which uses high-bandwidth remote CPU memory to overlap snapshot I/O with computation—while effective for dense models, fall short for MoE models, which expand training state by an order of magnitude without increasing iteration times. This expansion makes short checkpoint intervals prohibitively expensive: checkpointing every iteration for DeepSeek-16.4B/64E slows training by $2.5\times$ under Gemini (Fig. 1a, left Y-axis). Increasing the interval reduces per-iteration checkpoint overhead but lengthens recovery time (Fig. 1a, right Y-axis), as the average recomputation after a failure grows with interval length [13]. When failures are infrequent, this tradeoff has limited impact, but as MTBF drops—such as in large-scale runs where failures occur frequently—recovery costs compound rapidly, driving down ETTR¹. As shown in Figure 1b, Gemini’s ETTR peaks at 0.93 (2-hour MTBF) but is limited to at most 0.79 at 30-minute MTBF, plunging further to 0.47 at 10-minute MTBF.

¹Effective Training Time Ratio (ETTR) is the fraction of wall-clock time spent on useful training, excluding checkpointing and recovery. [40].



(a) Checkpoint interval vs. per-iteration % overhead (bar, log-scale, left Y-axis) and recovery time (line, log-scale, right Y-axis)



(b) ETTR across checkpoint intervals for varying MTBFs. Dashed lines indicate the maximum ETTR achieved for each MTBF.

Figure 1: Performance of Gemini [82] during training of DeepSeek-16.4B/64-Experts MoE model [12] using 96 A100 GPUs.

At million-dollar training budgets [10], such drops translate to hundreds of thousands of dollars in wasted computation, making these inefficiencies prohibitive at scale.

In this paper, we introduce MoEvent², which breaks the runtime–recovery tradeoff via *sparse checkpointing* (§3.2). Instead of checkpointing the full training state in a single iteration, MoEvent incrementally snapshots subsets of operators across multiple iterations, evenly distributing and fully overlapping checkpointing I/O with computation. By prioritizing operators based on activation frequency (§3.5), MoEvent enables low-overhead, high-frequency checkpoints. This removes the need to choose between frequent-but-expensive and infrequent-but-costly-to-recover checkpoints, allowing MoEvent to sustain $ETTR \geq 0.94$, even at low MTBFs.

Challenge #2: Correctness–Efficiency Tension: Recent MoE-specific checkpointing approaches, like MoC-System [8], attempt to reduce checkpoint overhead by snapshotting only a subset of experts per iteration in a round-robin fashion. While this lowers the overhead of frequent checkpoints, it compromises correctness: during recovery, experts without recent checkpoints revert to stale parameters, causing token loss and violating synchronous training semantics [53].

MoEvent overcomes the correctness–efficiency tradeoff with a *sparse-to-dense checkpoint conversion* mechanism (§3.3). Each sparse checkpoint stores full FP32 state for a subset of operators and FP16 state for the rest³. During recovery, operators are incrementally restored from FP16 to FP32, with iterations recomputed as needed until a consistent dense checkpoint is reconstructed. MoEvent preserves synchronous semantics, maintains accuracy, and retains reproducibility, ensuring no loss of tokens or training progress.

Challenge #3: Global Rollback Scope: To ensure synchronous semantics, existing checkpointing techniques roll back all workers, faulty or not, to a common checkpoint [22, 56, 76, 78, 82]. This global rollback amplifies recomputation overhead and prolongs recovery, a cost that grows sharply at

scale, where a single worker failure can force hundreds of otherwise healthy workers to revert training progress.

MoEvent employs *Upstream Logging* (§3.4), a targeted recovery mechanism that narrows rollback scope by logging intermediate activations and gradients at pipeline stage boundaries. On failure, only the affected data-parallel group rolls back to its most recent sparse checkpoint—typically just a few iterations—and completes *sparse-to-dense conversion* directly from the stored logs. This localized recovery shortens recovery time and eliminates the global rollback overhead.

We implemented MoEvent on top of DeepSpeed [71] and evaluated it across diverse MoE models spanning both vision and language domains, where it consistently sustains $ETTR \geq 0.94$ even under frequent failures, delivering up to $8\times$ overall training speedup. These gains stem from reducing checkpointing overhead by up to $4\times$ compared to MoC-System [8] and accelerating recovery by up to $31\times$ and $17\times$ relative to CheckFreq [56] and Gemini [82], respectively. Crucially, these efficiency improvements come with no loss in model accuracy. By breaking the runtime–recovery tradeoff, resolving correctness–efficiency tensions in prior checkpointing methods, and eliminating unnecessary global rollbacks, MoEvent provides a robust, scalable fault-tolerance solution purpose-built for large-scale MoE training.

2 Background & Related Work

2.1 Sparse Mixture-of-Experts Models

Mixture-of-Experts (MoE) models have emerged as a scalable and compute-efficient architecture for training large-scale neural networks across various domains. MoEs extend standard transformer by replacing dense feed-forward layers with multiple parallel subnetworks, known as *experts* [74], each activated selectively based on the input. A learned gating network routes each token to a small subset of these experts—typically one or two—and combines their outputs through learned weights. This selective activation enables MoE models to scale total parameter counts with only sub-linear growth in computational overhead, underpinning state-of-the-art foundation models such as DeepSeek-V3 [47], Gemini [18], Grok [84], gpt-oss [63], and Llama 4 [54].

To mitigate expert imbalance, MoEs commonly use auxiliary load-balancing objectives [16, 66, 91] to encourage more

²Pronounced “movement,” referring to its focus on MoE models and the movement of training state to ensure continued progress despite failures.

³Unless stated otherwise, and following standard practice [58, 67], we assume mixed-precision training: FP32 (32-bit floating point) master weights and optimizer states ensure numerical stability, while FP16 weights are used for forward and backward computation. As we show in §5.7, our techniques are also applicable to low-precision training regimes, including FP8 formats.

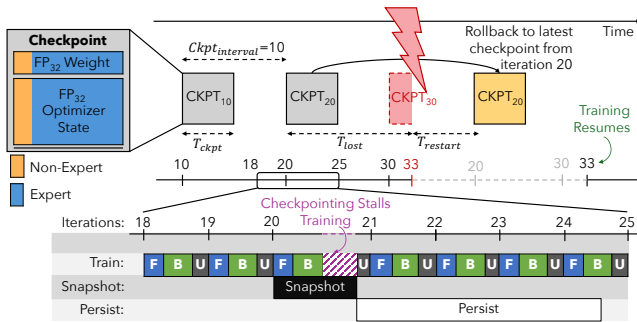


Figure 2: Checkpoint-based fault tolerance in distributed training. A checkpoint is taken every $Ckpt_{interval} = 10$ iterations. On failure, training rolls back to the most recent complete checkpoint (CKPT₂₀) and recomputes lost training progress, and then resumes.

uniform expert activation. In practice, however, activations remains naturally skewed due to input diversity, expert specialization, and training dynamics [24, 25, 89]. Attempts at strict balancing through strong regularization can disrupt expert specialization and degrade model performance [16, 47].

2.2 Distributed Training of MoE Models

State-of-the-art MoE models contain hundreds of billions of parameters trained on datasets with trillions of tokens [2, 16, 29, 75]. Training typically spans weeks and requires thousands of GPUs [3, 36, 54]. For instance, DeepSeek-V3, a 671B-parameter model, trained on 14.8 trillion tokens, consumed 2.7 million Nvidia H800 GPU hours—equivalent to 11 days on a 10,000-GPU cluster [47].

MoE training employs four primary parallelism strategies. *Data parallelism* (DP) splits batches across GPUs, synchronizing parameters with all-reduce operations [19, 73]. *Tensor parallelism* (TP) partitions model layers across GPUs to accelerate large matrix operations (e.g., GEMMs) [75]. *Pipeline parallelism* (PP) divides the model sequentially, processing micro batches in a pipelined fashion [31, 39, 45, 57]. *Expert parallelism* (EP), unique to MoEs, distributes experts across GPUs and routes tokens using all-to-all communication, enabling scalable conditional computation [24, 32, 43, 89].

Frequent Failures in Distributed Training. Modern training clusters comprise thousands of GPUs interconnected by sophisticated networking, storage, and power systems [40]. At this scale, failures are inevitable, arising from hardware faults (e.g., overheating GPUs, SSD degradation), transient network disruptions, software crashes, or service-level issues (e.g., monitoring agents exceeding resource limits) [30, 36, 92]. Alibaba Cloud reported abnormal termination rates of 44% for the top 5% resource-intensive jobs, slowing training by up to 40% [26, 50]. Meta projects MTBF as low as 14 minutes for 130,000-GPU clusters [40], a trend echoed by ByteDance [36], LAION [4], and Google [92].

2.3 Checkpointing Techniques

Checkpointing for Fault-Tolerant Training. Periodic checkpointing remains the predominant method for achieving fault

System	Low Overhead & High Frequency	Fast Recovery	Full Recovery	High ETTR
CheckFreq [56]	✗	✗	✓	✗
Gemini [82]	✗	✗	✓	✗
MoC-System [8]	✗	✓	✗	✗
MoEvent	✓	✓	✓	✓

Table 1: Comparison of periodic checkpointing techniques.

tolerance in distributed training [5, 26, 34, 36, 71, 78, 81]. However, naive checkpointing introduces significant stalls⁴, severely limiting ETTR, particularly for large models. Recent methods like CheckFreq [56] and Gemini [82] mitigate this by overlapping checkpoint operations with computation. CheckFreq introduces a two-phase checkpointing pipeline, dynamically adapting checkpoint intervals based on runtime measurements to balance overhead and recovery costs. Gemini employs in-memory checkpointing, utilizing high-bandwidth CPU memory and strategic placement of checkpoints to speed up recovery. Despite these optimizations, both methods incur substantial runtime overhead when frequently checkpointing large MoE models due to their significantly increased checkpoint sizes, as illustrated in Figure 1a. Consequently, neither can fully overlap large MoE checkpoint snapshots with computation, resulting in long intervals and persistently low ETTR—thus failing to adequately address the runtime–recovery tradeoff (Challenge #1).

More recently, MoC-System [8] proposed Partial Expert Checkpointing (PEC) for MoE models. PEC reduces checkpoint size by snapshotting only a subset of experts each iteration in a round-robin fashion. While initially effective, PEC compromises correctness during recovery: experts lacking recent checkpoints revert to stale states, causing token loss and breaking synchronous training semantics. Although MoC-System attempts to mitigate accuracy degradation by adaptively increasing the number of experts checkpointed after each failure, this approach rapidly devolves into dense checkpointing under frequent failures, nullifying its initial efficiency advantage. As a result, MoC-System struggles to balance correctness and efficiency (Challenge #2).

Other optimizations include storage-level enhancements such as NVMe-driven checkpoint acceleration (Fast-Persist [80]), concurrent data-parallel checkpoint sharing (Megascale [36]) and persistence (PCCheck [76]), parallelism-agnostic checkpoint representations (ByteCheckpoint [78]), and checkpoint size reduction via model-specific quantization (Check-N-Run [15], CPR [50]). Yet, these techniques still require checkpointing the entire model state simultaneously, inevitably causing stalls and limiting checkpoint frequency.

Checkpoint-less Fault-Tolerance. An orthogonal approach to traditional checkpointing exploits redundancy in training setups to minimize or eliminate periodic checkpoints entirely. Just-in-Time [22] leverages redundancy across data-

⁴Checkpoint-induced stall occurs when checkpoint I/O exceeds the forward/backward pass, delaying the optimizer step until checkpoint completion.

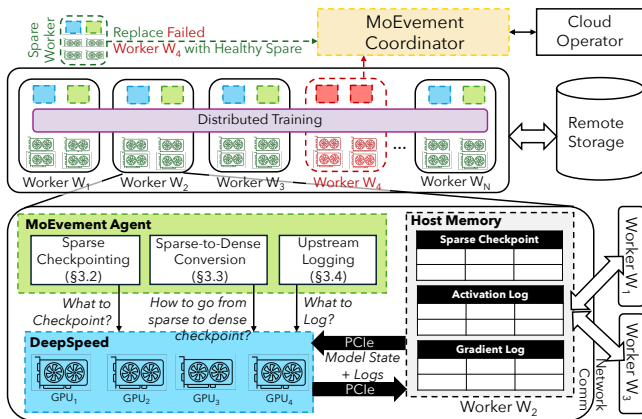


Figure 3: The system architecture of MOEVEMENT

parallel replicas, deferring checkpointing until failures are actually detected. More aggressive checkpoint-less strategies include Bamboo [77], which adds redundant computations to withstand preemptions in cloud spot instances, Oobleck [33], which uses heterogeneous pipelines to recover from failures without requiring spare resources, and ReCycle [17], which dynamically reroutes workloads to functionally redundant data-parallel peers to continue computation after failures.

However, these checkpoint-less methods critically depend on the presence of redundancy, limiting their applicability as memory optimizations such as Fully Sharded Data Parallelism (FSDP) and ZERO-style optimizations [69–71, 90] deliberately eliminate it. This creates a strong need for checkpoint-based techniques in redundancy-constrained training setups.

2.4 How MTBF Affects ETTR

Effective Training Time Ratio (ETTR) depends on two factors: runtime overhead from checkpointing during fault-free operation and recovery overhead from failures. Modeling failures as a Poisson process with rate $\frac{1}{\text{MTBF}}$, ETTR can be approximated by [28]:

$$\text{ETTR} \approx \frac{1}{1 + \frac{T_{\text{ckpt}}}{T_{\text{iter}} \times \text{Ckpt}_{\text{interval}}}} \times \frac{1}{1 + \frac{\mathbb{E}[R]}{\text{MTBF}}}$$

Runtime Overhead
Recovery Overhead

where T_{iter} is the iteration time, $\frac{T_{\text{ckpt}}}{T_{\text{iter}} \times \text{Ckpt}_{\text{interval}}}$ is the runtime overhead from checkpointing every $\text{Ckpt}_{\text{interval}}$ iterations, and $\mathbb{E}[R]$ is the expected recovery time per failure which scales linearly with checkpoint interval, on-average half the interval ($\mathbb{E}[R] \approx \frac{1}{2} \times \text{Ckpt}_{\text{interval}} \times T_{\text{iter}}$) [13]. These relationships imply an inherent trade-off in selecting checkpoint intervals: longer intervals reduce runtime overhead but increase expected re-computation after failures, and vice versa. The *optimal checkpoint interval* balances these competing costs to maximize ETTR, thereby achieving faster end-to-end training. As seen in Fig. 1b, as MTBF decreases, recovery overhead starts to dominate, pushing ETTR downward and shifting optimal checkpoint intervals toward shorter durations.

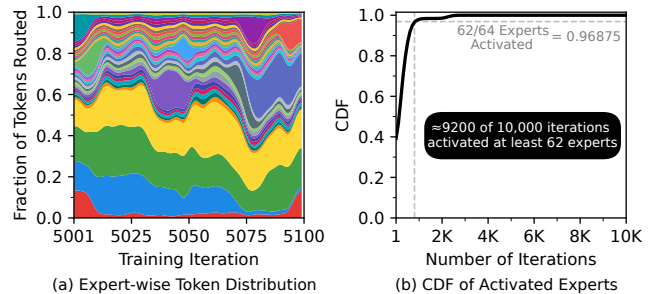


Figure 4: MoE training dynamics in DeepSeek-16.4B/64E [12]. (a) Token distribution (color-coded by expert) is dynamic and skewed. (b) CDF of activated experts shows that nearly all experts are active in most iterations, each receiving non-zero tokens with uneven shares.

3 The MOEVEMENT Approach

3.1 MOEVEMENT Overview

MoEvent is a distributed, in-memory checkpointing system tailored to the expert-parallel architecture of MoE models. It reduces checkpointing overhead by exploiting the dynamic and skewed token-to-expert assignment patterns inherent in MoE training, while preserving synchronous training semantics and ensuring fast and accurate failure recovery.

MoEvent introduces three key ideas. First, *Sparse Checkpointing* §3.2 incrementally captures subsets of operators across multiple iterations instead of checkpointing the full training state at once. By spreading the work over a sparse checkpointing window, it incurs negligible runtime overhead during fault-free execution, enabling high-frequency, continuous checkpointing. Second, *Sparse-to-Dense Checkpoint Conversion* §3.3 resolves the temporal inconsistency of sparse snapshots by incrementally reconstructing a logically consistent dense checkpoint—selectively replaying computations and activating operators as their master weights and optimizer state become available. This technique ensures correctness and recovery semantics of dense checkpointing without incurring its overhead. Third, *Upstream Logging* §3.4 captures input activations flowing forward and gradients propagating backward at each pipeline-stage boundary. During recovery, these logs enable each stage to independently recompute the state of failed workers without rolling back unaffected ones.

3.2 Avoiding Stalls with Sparse Snapshots

MoEvent exploits two key insights to achieve low-overhead, high-frequency checkpointing. First, token distribution across experts is dynamic and skewed: nearly all experts are *active* (assigned at least one token) in most steps ($\geq 62/64$ in $\approx 9.2\text{K}/10\text{K}$ iterations), yet token shares fluctuate widely (Fig. 4). MoEvent strategically leverages this imbalance, as detailed in Section 3.5. Second, the iterative nature of training allows reconstructing an operator’s full training state (model weights and optimizer state) from earlier checkpoints by replaying micro batches. Rather than simultaneously checkpointing all operators, MoEvent in-

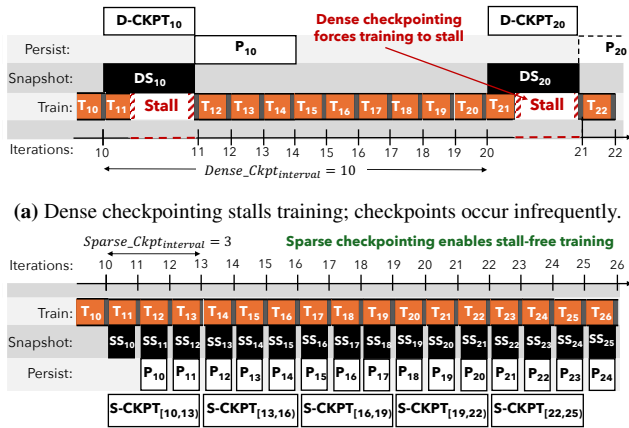


Figure 5: Dense vs. Sparse checkpointing

crementally checkpoints subsets of operators over multiple iterations. At each iteration, only a small subset of operators checkpoint their training state; others checkpoint only their compute weights—83% smaller (2 bytes vs. 12 bytes per parameter in FP16-FP32 mixed-precision training using Adam optimizer [48])—and temporarily enter a *frozen* state⁵. Transitions into and out of this *frozen* state are orchestrated by the sparse-to-dense conversion mechanism detailed in §3.3, cutting per-iteration checkpoint sizes by $\approx 55\%$ (Fig. 6(Inset)), fully overlapping checkpoint I/O with computation, and eliminating stalls (Fig. 5b).

We define a *snapshot* as moving the training states from GPU to local CPU memory during training. Traditional *dense* checkpointing methods snapshot the entire training state, model weights and optimizer state, within a single iteration ($W_{dense} = 1$). If the snapshot is not completed before the subsequent optimizer step, training stalls, which forces long intervals to amortize cost (e.g., $Ckpt_{interval} = 10$ in Fig. 5a).

Sparse checkpointing treats each expert (E_1 – E_4), non-expert (NE), and gating (G) operator as independently snapshotable. As illustrated for a three-layer MoE under FP16-FP32 mixed-precision training (Fig. 6), iteration 11: dense DS_{10} would snapshot all operators at once, whereas sparse SS_{10} snapshots E_1, E_2 's FP32 states and E_3, E_4, NE, G 's FP16 compute weights; iteration 12 (SS_{11}) snapshots E_3, E_4 's FP32 states and FP16 weights for NE, G; iteration 13 (SS_{12}) snapshots NE, G's FP32 states. After three iterations ($W_{sparse} = 3$), each operator has exactly one FP32 snapshot.

Persisting Snapshots. Once a sparse snapshot is moved to local CPU memory, MoEvent asynchronously persists it to remote CPU memory on r peer nodes (similar to Gemini [82]), concurrently with training. A sparse checkpoint is considered persisted once all snapshots within W_{sparse} window are durably replicated. MoEvent always maintains one per-

⁵*Frozen* operators skip weight-gradient computations and optimizer updates, performing only forward and input-gradient computations.

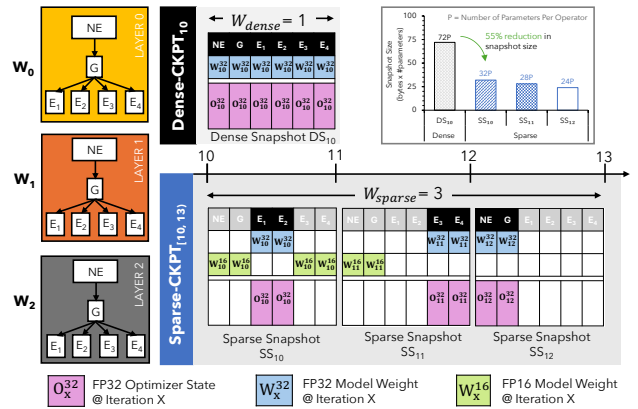


Figure 6: Dense checkpointing snapshots FP32 optimizer state and model weights for all operators in a single iteration. Sparse checkpointing incrementally snapshots FP32 states of different operator subsets over three iterations. **Inset (top-right):** Sparse checkpointing reduces the per-snapshot size by 55% relative to dense.

sisted checkpoint and another in-flight, garbage-collecting the oldest checkpoint after persisting a new one.

3.3 Sparse-to-Dense Checkpoint Conversion

Sparse checkpointing avoids stalls but introduces temporal inconsistencies: operators are snapshotted at different iterations (e.g., in $S\text{-CKPT}_{[10,13]}$ operators E_1, E_2 at iteration 10, E_3, E_4 at iteration 11, and NE, G at iteration 12). If left unresolved, these mismatches can yield incorrect recovery states and degrade model accuracy [15, 50, 65]. MoEvent addresses this by incrementally reconstructing a logically consistent dense checkpoint from multiple sparse snapshots.

We assume standard mixed-precision FP16-FP32 training, where FP16 weights are used for forward and backward computation, and FP32 master weights and optimizer state are updated each optimizer step. The key insight is that an operator's state at iteration t depends entirely on its FP32 state at some earlier iteration s ($s < t$) and the parameter updates applied since. Thus, once an *anchor* snapshot is captured at iteration s , future FP32 states can be reconstructed by replaying micro batches and re-applying updates. Crucially, replaying does not require FP32 states for operators not yet updating parameters; FP16 weights alone suffice to compute necessary input gradients. In MoEvent, operators are classified based on FP32 state availability at snapshot loading. *Active* operators have FP32 weights and optimizer state, performing forward, backward, and optimizer updates. In contrast, *frozen* operators, with only FP16 weights, perform forward and input-gradient computations but skip weight-gradient computations and optimizer updates until a later *anchor* snapshot provides their FP32 state, at which point they become *active* (Fig. 7).

Figure 8 illustrates incremental reconstruction of dense checkpoint at iteration 13 ($D\text{-CKPT}_{13}$) from sparse snapshots (SS_{10} – SS_{12}). Loading snapshot SS_{10} activates operators E_1 and E_2 (FP32 state available), leaving operators E_3, E_4, G , and

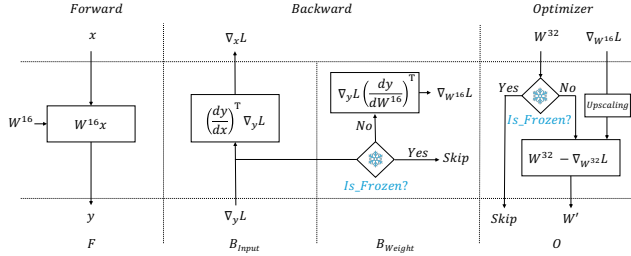


Figure 7: Conditional execution of forward, backward, and optimizer steps based on operator state (*frozen* vs. *active*).

NE frozen with FP16 weights (1). Replaying micro batches of iteration 11 updates the FP32 weights and optimizer state of active operators (E_1, E_2), while frozen operators (E_3, E_4, NE , and G) participate only in forward and input-gradient computation (2). Next, loading snapshot SS_{11} activates operators E_3 and E_4 alongside E_1 and E_2 , while G and NE remain frozen (3). Replaying micro batches of iteration 12 updates parameters and optimizer state for active operators (E_1 – E_4), advancing them to iteration 12, whereas frozen operators (NE and G) continue propagating gradients without updates (4). Finally, loading snapshot SS_{12} activates the remaining operators G and NE , making all operators active (5). Replaying micro batches of iteration 13 updates all operators, achieving a consistent dense checkpoint identical to one captured by traditional dense checkpoint technique (6).

3.4 Upstream Logging for Localized Recovery

Fast recovery from failures is essential to sustaining high training efficiency. In dense checkpointing, a failure at any pipeline stage triggers rollback of all stages, forcing even non-faulty workers to redo computations [36, 56, 82]. This cascaded rollback inflates re-computation and prolongs recovery times, as 1F1B schedules introduce pipeline bubbles while re-priming the pipeline (Figure 9b, left). For example, recovering stage W_1 in a three-stage pipeline requires rolling back W_0, W_1 , and W_2 (Figure 9a, left), a penalty that grows as pipeline depth increases—a common occurrence as large models scale beyond a single node using pipeline parallelism [57, 58].

MoEvent avoids cluster-wide rollback with *Upstream Logging*, a lightweight mechanism that restricts rollback to only the affected data-parallel groups. During training, MoEvent logs intermediate tensors at each pipeline stage boundary: (1) activations passed downstream during forward propagation, and (2) gradients sent upstream during backward propagation. Logs are recorded at the sender stage and stored off GPU in host (CPU) memory. Because logging occurs locally at the sender, it introduces no additional network communication and remain accessible even if upstream or downstream workers fail. Each activation and gradient log is tagged with iteration numbers and micro batch identifiers to allow precise and ordered replay during localized recovery.

When a failure occurs, MoEvent pauses all data-parallel (DP) groups, aborts the current iteration, and replaces the

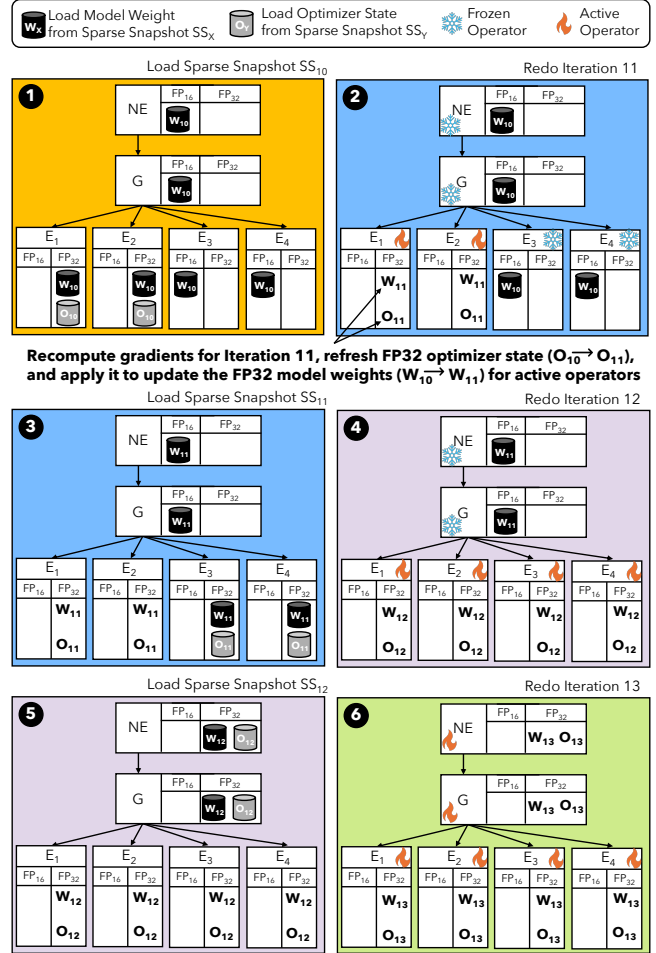
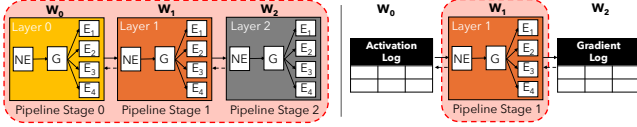


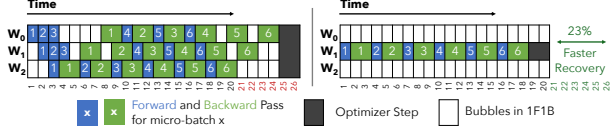
Figure 8: Step-by-step sparse-to-dense checkpoint conversion over iterations 11–13. Operators transition from *frozen* to *active* as their FP32 model weights and optimizer state become available.

failed node with a spare. Only the affected DP group rolls back to its most recent sparse checkpoint—usually just a few iterations old—while other groups remain paused in a consistent state. The failed stage then replays computations locally using logs stored on its upstream and downstream neighbors, reconstructing a consistent dense checkpoint without requiring global re-computation. As shown in Figure 9a (right), when W_1 fails, re-computation stays confined to W_1 , reducing recovery latency by 23% (Figure 9b, right), by avoiding pipeline bubbles during the start-up and cool-down phases. Similarly, when multiple concurrent failures occur, affected DP groups independently perform localized recovery in parallel, without global coordination (additional details in Appendix A).

Stale Log Cleanup. Logged tensors from prior sparse checkpoints become obsolete once a new sparse checkpoint is persisted, typically every W_{sparse} iterations. To avoid unbounded accumulation, MoEvent proactively garbage collects stale logs from host (CPU) memory. For the MoE models evaluated in §5.6, these logged tensors occupy less than 2% of available host (CPU) memory (see Table 6).



(a) Recomputation scope *without* (left) and *with* (right) Upstream Logging



(b) 1F1B schedule *without* (left) and *with* (right) Upstream Logging

Figure 9: Upstream Logging narrows recomputation scope to the affected worker (W_1), achieving 23% faster recovery.

3.5 Sparse Checkpointing Policy

To implement sparse checkpointing effectively, MoEvent jointly optimizes two key parameters: the checkpoint window size (W_{sparse}) and the order in which operators are checkpointed. Together, these choices determine the checkpointing overhead and the recovery overhead.

Determining W_{sparse} . In MoEvent, the training state is captured incrementally over W_{sparse} iterations using *sparse checkpointing*, which snapshots only a subset of operators in each iteration. To maintain low-overhead checkpointing, MoEvent selects the smallest W_{sparse} whose snapshot fits within an iteration, determined by the `FindWindowSize()` function in Algorithm 1. The function starts with all operators marked as *active* for checkpointing and gradually transitions some to *frozen*, recalculating the snapshot size at each step. For *active* operators, it records FP32 master weights and optimizer state; for *frozen* operators, it stores FP16 model weights. The process terminates once the estimated snapshot time fits within the iteration time, ensuring checkpointing does not force training to stall.

Determining Operator Ordering. Expert popularity is measured by the frequency with which an expert is activated. For expert j in layer l , the activation count is:

$$\mathcal{A}_j^l = \sum_{x_i \in \mathcal{D}} \mathbf{1}[\gamma(e_j^l, x_i) = 1]$$

where \mathcal{D} is the training dataset and $\gamma(e_j^l, x_i)$ indicates whether expert j in layer l is activated for token x_i . Popularity distributions are naturally skewed—some experts are activated more frequently than others, and are commonly referred to as *popular experts* [24, 60]. MoEvent opportunistically exploits this imbalance by sorting experts in ascending order of popularity using the `OrderOperators()` function, deferring the checkpointing of popular experts to later iterations within the sparse checkpointing window. Deferring popular experts keeps them *frozen* longer during sparse-to-dense conversion, avoiding weight-gradient and optimizer updates and lowering recomputation cost by $\approx 33\%$ compared to *active* experts. Alternative ordering strategies are described in Appendix B.

Algorithm 1 Sparse Checkpoint Scheduling

```

1 # O: List of operators in MoE model
2 # T_Iter: Iteration Time
3 # S_Compute: Size of compute weights per operator
  ↪ used in Forward/Backward Pass
4 # S_Master: Size of master weights per operator
5 # S_Optim: Size of optimizer state per operator
6 # B_PCIe: Effective GPU-to-CPU PCIe bandwidth
7 def FindWindowSize():
8     O_Total, T_Iter, S_Compute, S_Master, S_Optim,
  ↪ B_PCIe = Profiler(O) #get profiled stats
9     O_Active = O_Total #all operators active
10    while O_Active > 2:
11        O_Frozen = O_Total - O_Active
12        ckpt_size = (S_Master + S_Optim) * O_Active
13                  + S_Compute * O_Frozen
14        if ckpt_size / B_PCIe <= T_Iter:
15            break #checkpoint fits within iter time
16        O_Active -= 1 #try fewer active operators
17    W_Sparse = ceil(O_Total / O_Active)
18    return W_Sparse, O_Active
19 def GenerateSchedule(O, W_Sparse, O_Active):
20    O_Ordered = OrderOperators(O) #popularity sort
21    schedule = []
22    for i in range(W_Sparse):
23        start = i * O_Active
24        end = min(start + O_Active, len(O))
25        schedule.append({
26            'active': O_Ordered[start:end],
27            'frozen': O_Ordered[end:]
28        })
29    return schedule
30 def SparseCheckpointSchedule():
31    W_Sparse, O_Active = FindWindowSize()
32    return GenerateSchedule(O, W_Sparse, O_Active)

```

Expert popularity can evolve over the course of training [25, 89]. MoEvent reorders operators when activation frequencies change by over 10% for at least 25% of experts, balancing responsiveness with schedule stability (evaluated in §5.6). We further analyze the impact of expert popularity skewness on MoEvent’s performance in Appendix D, and discuss generalizing our checkpointing policy to dense models in Appendix E.

Runtime Analysis. The overall complexity of Algorithm 1 is $\mathcal{O}(|O| \log |O|)$, where $|O|$ is the number of operators. The `FindWindowSize()` function runs in $\mathcal{O}(|O|)$ time, with the dominant cost coming from sorting operators by popularity during schedule generation. In practice, the algorithm runs on the CPU, completing in (≈ 0.1 sec), and executes asynchronously without interrupting GPU computation.

3.6 Recovery Guarantees under Sparse Checkpointing

When a failure occurs, training resumes from the most recent checkpoint. In existing techniques, a dense checkpoint is taken every $\text{Ckpt}_{\text{interval}}$ iterations. Assuming failures occur uniformly at random, let R denote the recovery time following a failure and T_{iter} the duration of a single iteration. For existing checkpointing techniques, R is bounded by the checkpoint interval, and the expected recovery time $\mathbb{E}[R]$ is, on-average,

half the checkpoint interval [13, 56]:

$$0 \leq R \leq \text{Ckpt}_{\text{interval}} \times T_{\text{iter}} \quad \text{and} \quad \mathbb{E}[R] \approx \frac{1}{2} \times \text{Ckpt}_{\text{interval}} \times T_{\text{iter}}$$

In contrast, MoEevent recovers in two phases: (1) replaying W_{sparse} iterations to reconstruct a dense checkpoint from the most recent sparse checkpoint, and (2) re-executing up to W_{sparse} additional iterations from that restored state. This yields the following bounds R and $\mathbb{E}[R]$:

$$0 \leq R \leq 2 \times W_{\text{sparse}} \times T_{\text{iter}} \quad \text{and} \quad \mathbb{E}[R] \approx \frac{3}{2} \times W_{\text{sparse}} \times T_{\text{iter}}$$

Empirically, we find $W_{\text{sparse}} \ll \text{Ckpt}_{\text{interval}}$, and demonstrate in §5.2 that MoEevent checkpoints up to $26\times$ more often than dense checkpointing techniques like Gemini and CheckFreq.

4 Implementation

We implement MoEevent on top of DeepSpeed v0.16 [71], adding $\approx 2K$ lines of Python code.

Sparse Snapshot. We modify DeepSpeed’s checkpoint saving routine to operate at per-operator granularity, treating each expert, non-expert, and gating operator as independently snapshottable. For *active* operators, we record master weights and optimizer states; for *frozen* operators, only compute weights. GPU-to-CPU transfers use pinned host buffers, allocated once at the beginning of the training job and reused throughout, with asynchronous `cudaMemcpyAsync` calls on a dedicated CUDA stream to overlap device-to-host I/O with forward and backward passes. Once in host memory, snapshots are replicated to r peer nodes (default $r = 2$) via `torch.distributed` point-to-point (`send/recv`), with replication running asynchronously alongside training.

Sparse-to-Dense Conversion. We integrate sparse-to-dense conversion into DeepSpeed’s checkpoint loading routine. On load, each operator is marked *active* if their master weights and optimizer states are available, otherwise marked *frozen*. *Active* operators run forward, backward, and optimizer update steps, while *frozen* operators perform only forward and input-gradient computations. We extend DeepSpeed with state-aware execution paths, wrapping gradient computation and optimizer updates in state checks, using `torch.no_grad()` to skip autograd tracking for *frozen* operators. During recovery, the replay engine loads sparse snapshots in schedule order, replays the corresponding micro batches, and transitions operators to *active* state once their full state is restored from the sparse snapshot. This continues until all operators are marked *active*, yielding a dense checkpoint.

Upstream Logging. We extend DeepSpeed’s PipelineModule to log a copy of activations and gradients in pinned host buffers, tagged with iteration and micro batch ID. DeepSpeed retains these tensors in GPU memory while transmitting them to the next pipeline stage; we leverage this period to issue an asynchronous `cudaMemcpyAsync` on a dedicated CUDA stream, overlapping GPU-to-CPU

Model	#Layers	Gate	#Experts Per Layer	#Activated Per Token	Total Params	Active Params
MoE-LLaVa [46]	32	Top-2	4	2	2.9B	2B
GPT-MoE [68]	12	Top-6	32	6	7.3B	1.6B
QWen-MoE [86]	24	Top-8	64	8	14.3B	2.7B
DeepSeek-MoE [12]	28	Top-8	64	2(shared) + 8	16.4B	3.7B

Table 2: Specifications of models used for evaluation.

transfers with ongoing training. Logged tensors remain in host memory until their corresponding sparse checkpoint is either consumed during recovery or garbage-collected.

5 Evaluation

5.1 Experimental Setup

Cluster. Unless otherwise noted, all of our experiments were conducted on a GPU cluster with 12 Standard_NC96ads_A100_v4 nodes, each of which has a 64-core AMD EPYC 7V13 CPU, 880 GB of RAM, and eight Nvidia A100 80 GB GPUs on Azure Cloud. GPUs on the same node are connected via a 600 GB/s NVLink interconnect, and nodes are connected via 80 Gbps inter-node interconnect across 8 NICs. Azure Blob Storage is used as the remote persistent storage and the aggregated bandwidth to it is 40Gbps. All servers run 64-bit Ubuntu 22.04 with CUDA library v12.8.

Baselines. We compare MoEevent against Gemini [82], a state-of-the-art in-memory checkpointing system, CheckFreq [56], a disk-based checkpointing alternative, and MoC-System [8], a recent MoE-specific checkpointing technique. To assess potential interference from these checkpointing systems, we also measured the throughput of DeepSpeed-MoE [71] with checkpointing disabled, utilizing a 1F1B interleaved schedule and ZERO Stage-1 optimizations [69].

Models. We evaluate all systems using the four representative MoE models shown in Table 2: MoE-LLaVa [46], GPT-MoE [62], QWen-MoE [86], and DeepSeek-MoE [12]. GPT-MoE, QWen-MoE, and DeepSeek-MoE are trained on RedPajama dataset [83] with a batch size of 512, micro-batch size of 32, and sequence length of 2048 tokens using parallelization strategies (PP, DP, EP) degrees of (3, 4, 8), (6, 2, 8), and (12, 1, 8) respectively. MoE-LLaVa is trained on the ImageNet-1K dataset [11] using (PP, DP, EP) degrees of (6, 2, 8).

5.2 Training Efficiency Under Controlled Failures

We evaluate MoEevent under controlled failures during training of the four MoE models listed in Table 2, using a range of failure rates by varying MTBF from 2 hours down to 10 minutes over 12-hour training runs. Table 3 summarizes our results, addressing three questions:

What checkpoint intervals are achievable, and at what overhead? Checkpoint interval—the number of iterations between successive checkpoints—directly impacts checkpoint overhead and recovery performance, while checkpoint window denotes iterations over which a checkpoint is spread, ensuring every operator is snapshotted at least once. For dense checkpointing methods (CheckFreq and Gemini), this win-

Model	Checkpointing Interval (iterations)				Avg. Per-Iteration Checkpointing Overhead (seconds, overhead % in parentheses)			MTBF	Total Recovery Time (seconds)				Effective Training Time Ratio (ETTR)				
	CheckFreq	Gemini	MoC	MoEvent	CheckFreq	Gemini	MoC		MoEvent	CheckFreq	Gemini	MoC	MoEvent				
														CheckFreq	Gemini	MoC	MoEvent
MoE-LLaVa [46]	57	1	1	$(W_{\text{sparse}}=3)$	0.03 (2%)	0.02 (1%)	0.14 (8%)	0.01 (1%)	2H	1265	938	10	11	0.945	0.959	0.922	0.981
									1H	2530	1551	20	24	0.914	0.939	0.375	0.979
									30M	5059	2203	41	57	0.853	0.897	0.328	0.975
									20M	7589	2570	61	74	0.792	0.852	0.231	0.971
									10M	11178	4161	98	117	0.712	0.764	0.184	0.964
GPT-MoE [68]	78	1	1	$(W_{\text{sparse}}=3)$	0.03 (1%)	0.03 (1%)	0.18 (9%)	0.03 (1%)	2H	1424	1121	13	18	0.937	0.947	0.919	0.979
									1H	2848	1718	25	28	0.903	0.923	0.384	0.976
									30M	4486	2160	50	78	0.834	0.868	0.326	0.971
									20M	8543	3259	76	105	0.765	0.815	0.220	0.966
									10M	13086	4627	132	161	0.687	0.722	0.179	0.959
QWen-MoE [86]	113	1	1	$(W_{\text{sparse}}=5)$	0.05 (2%)	0.04 (2%)	0.19 (9%)	0.04 (2%)	2H	1170	961	15	17	0.927	0.934	0.927	0.981
									1H	2340	1402	30	33	0.898	0.915	0.380	0.977
									30M	4680	2114	60	83	0.841	0.868	0.325	0.970
									20M	7020	2331	90	140	0.783	0.819	0.205	0.963
									10M	10040	3495	146	207	0.703	0.717	0.162	0.957
DeepSeek-MoE [12]	124	1	1	$(W_{\text{sparse}}=6)$	0.08 (3%)	0.07 (2%)	0.22 (8%)	0.06 (2%)	2H	992	800	17	22	0.902	0.910	0.933	0.975
									1H	1984	1218	35	43	0.877	0.894	0.349	0.970
									30M	3967	1879	70	104	0.827	0.853	0.324	0.965
									20M	5951	1983	104	167	0.777	0.814	0.168	0.956
									10M	9402	3236	176	241	0.672	0.728	0.134	0.951

Table 3: Comparison of checkpointing techniques across representative MoE models under varying Mean Time Between Failures (MTBF).

down is always 1. For CheckFreq, we configure its policy module to select intervals that cap runtime overhead at $\leq 3\%$, resulting in intervals of 57–124 iterations across evaluated models and overhead of ≤ 0.08 secs/iter. Gemini, in contrast, employs an oracle policy: intervals are selected offline individually for each MTBF to maximize ETTR. This idealized, hindsight-informed selection provides an upper bound on its achievable performance. At MTBF=2H, Gemini selects shorter intervals (46–92 iterations) with lower overhead ($\leq 2\%$) than CheckFreq, benefiting from the use of high-bandwidth CPU memory rather than disk. However, as MTBF decreases, Gemini must shorten intervals further (e.g., 17–31 iterations at MTBF=10M) to minimize recomputation, causing per-iteration overhead to increase to 11–15%.

MoC aggressively checkpoints every iteration by partially snapshotting experts, resulting in an effectively unbounded checkpoint window, risking token loss as some experts remain uncheckpointed indefinitely. MoC maintains modest overhead under infrequent failures, but overhead spikes dramatically as failures become frequent and the token-loss budget is exhausted (e.g., $\approx 4.7\times$ slowdown at MTBF=10M for DeepSeek-MoE). In contrast, MoEvent checkpoints every iteration by incrementally snapshotting subsets of operators over a checkpoint window W_{sparse} , selected by Algorithm 1, ensuring all experts are checkpointed per window, eliminating token loss (§3.3). This evenly distributes checkpoint operation across multiple iterations, maintaining low overhead ($\leq 2\%$ slowdown) across all models and failure scenarios.

How quickly can we recover from failures? Total recovery time—the cumulative time spent recovering from all failures—depends on both checkpoint intervals and failure frequency. CheckFreq and Gemini incur extended recovery times (9402–13086 secs for CheckFreq, 3236–4627 secs for Gemini at MTBF=10M) due to global rollbacks and recomputation of half their checkpoint intervals, on average, after each failure. MoC’s per-iteration checkpointing strategy limits recomputation and thus total recovery time (98–176 secs at

MTBF=10M). However, this aggressive strategy comes at the cost of high runtime overhead—up to 470% at MTBF=10M. In comparison, MoEvent restricts rollbacks exclusively to the affected data-parallel group and limits recomputation to $\lceil \frac{3}{2} \times W_{\text{sparse}} \rceil$ iterations, achieving recovery times of 117–241 secs at MTBF=10M—only marginally higher than MoC due to the sparse-to-dense conversion during recovery. Crucially, MoEvent preserves all tokens and synchronous training semantics like CheckFreq and Gemini, yet recovers up to $31\times$ and $18\times$ faster, respectively.

What training efficiency can be achieved under failures? ETTR—the fraction of wall-clock time spent on useful training computation—encapsulates the combined effects of checkpoint overhead and recovery downtime. CheckFreq and Gemini achieve high ETTR (0.90–0.96) at MTBF=2H due to infrequent failures, but their prolonged recovery periods degrade ETTR under frequent failures. CheckFreq drops to 0.672, Gemini to 0.728 at MTBF=10M. MoC starts strong at low failure rates, but its high checkpoint overhead degrades ETTR under frequent failures, falling to just 0.134 for DeepSeek-MoE at MTBF=10M. MoEvent consistently maintains high ETTR (0.951–0.964 at MTBF=10M) by effectively combining high frequency checkpointing with low runtime overhead. This approach allows MoEvent to match MoC’s fast recovery, CheckFreq and Gemini’s low overhead without inheriting their respective penalties.

Ultimately, this superior training efficiency translates to tangible end-to-end performance improvements: at MTBF=10M, MoEvent completes training up to $1.4\times$ faster than CheckFreq, $1.3\times$ faster than Gemini, and $7.1\times$ faster than MoC.

5.3 Training Efficiency Under Dynamic Failures

To further evaluate MoEvent under realistic conditions, we replay a 6-hour failure trace collected from Google Cloud Platform (GCP) instances, as used in recent works [17, 33, 77]. Figure 10a shows the 24 failure events in this period, corresponding to an average MTBF of ≈ 19 minutes.

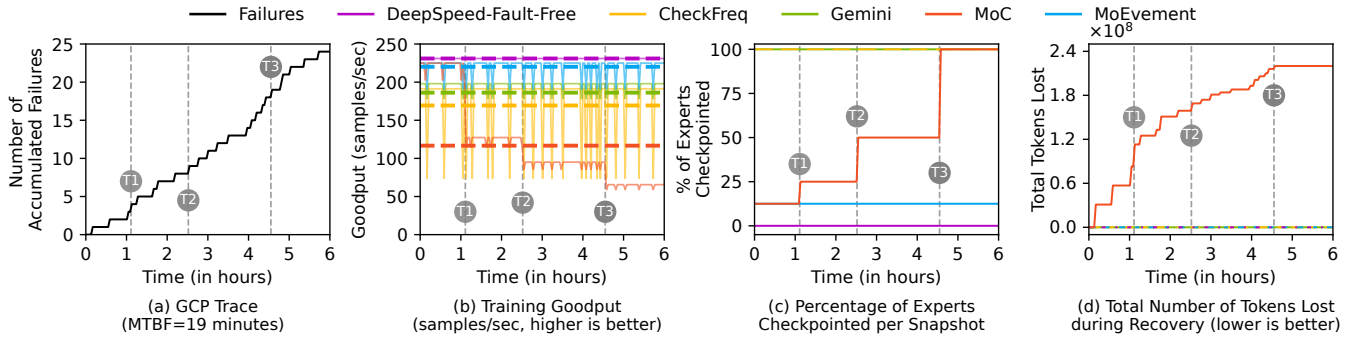


Figure 10: Training DeepSeek-MoE under a 6-hour real-world failure trace. MoEvEvent sustains the highest goodput throughout, while MoC’s goodput declines as its lost-token budget is exhausted, prompting more experts to be checkpointed per snapshot. CheckFreq and Gemini achieve stable but lower goodput due to long checkpoint intervals. In 10b, dashed lines show the average goodput over the 6-hour trace.

Model	System	1H	30M	10M
QWen-MoE	Gemini	-0.14%	-1.47%	+1.02%
	MoEvEvent	+0.86%	-0.73%	+0.30%
DeepSeek-MoE	Gemini	-0.87%	-0.56%	-0.19%
	MoEvEvent	-0.59%	+1.22%	+0.25%

Table 4: Difference between simulated and measured ETTR.

Figure 10b compares goodput—useful throughput (samples/sec) excluding recomputed samples after failures—for each checkpointing method over the replayed trace. CheckFreq and Gemini exhibit consistently lower goodput due to frequent recomputation caused by their long checkpoint intervals (124 and 92 iterations, respectively). MoC initially achieves high goodput by checkpointing a smaller set of experts, as seen in Fig. 10c. However, as lost tokens from recovery with partial snapshots accumulate (Fig. 10d), MoC must progressively checkpoint a larger fraction of experts (from 12.5% at T1 to 100% at T3 in Fig. 10c) to preserve accuracy, which reduces its goodput by 71% over time.

In contrast, MoEvEvent maintains the highest goodput throughout the trace, delivering 1.25 \times , 1.15 \times , and 1.98 \times higher goodput than CheckFreq, Gemini, and MoC-System, respectively. This demonstrates MoEvEvent’s effective balance of low checkpoint overhead, fast localized recovery, and no lost tokens under dynamic failure conditions.

5.4 MoEvEvent Scalability

Simulator. Due to the unavailability of a cluster with thousands of GPUs, we built a simulator to estimate Effective Training Time Ratio (ETTR) for arbitrary model and cluster configurations, given a specified MTBF and checkpointing technique (detailed in Appendix C). It leverages real-world profiled statistics for each pipeline operation, capturing both computation and communication costs. We validated accuracy against measurements on Azure for two MoE models under diverse failure scenarios. Table 4 shows a maximum deviation of 1.47%, primarily due to minor runtime variations in NCCL collectives, which minimally affect MoEvEvent’s relative performance and scalability trends.

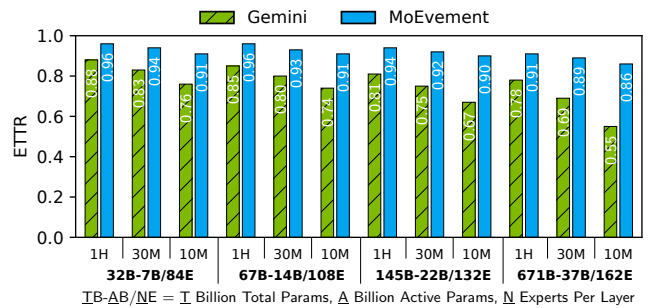


Figure 11: Simulated ETTR of MoEvEvent as model size increases.

How effectively does MoEvEvent scale to large models?

Using the simulator, we evaluate DeepSeek-MoE models with 32B, 67B, 145B, and 671B total parameters across clusters configured as follows: (512 GPUs, 16 stages per pipeline, 4 pipelines), (1536 GPUs, 24 stages per pipeline, 8 pipelines), (4096 GPUs, 32 stages per pipeline, 16 pipelines), and (16384 GPUs, 64 stages per pipeline, 32 pipelines). All models employ 8-way expert parallelism, corresponding to the NVLink domain size. Figure 11 presents the ETTR achieved by Gemini and MoEvEvent across varying MTBF.

MoEvEvent consistently sustains high ETTR across increasingly large models and clusters, even as failure rates increase. At a 1-hour MTBF, MoEvEvent achieves ETTRs above 0.91 across all scales. As MTBF drops to 30 minutes and 10 minutes, ETTR declines gradually, but MoEvEvent remains significantly more resilient than Gemini. For instance, on the 671B model, MoEvEvent achieves ETTR of 0.86 under 10-minute MTBF, compared to Gemini’s 0.55; delivering 1.55 \times faster training. This performance gap stems from fundamental differences in design. Gemini checkpoints less frequently and must roll back all machines—faulty and healthy—on failure. As pipeline depth and cluster size grow, the cost of global rollback becomes increasingly severe, compounding runtime disruption. In contrast, MoEvEvent’s localized recovery avoids full-cluster rollbacks and supports frequent checkpointing. This design allows MoEvEvent to scale more gracefully under high failure rates, maintaining high training efficiency even at industrial scale.

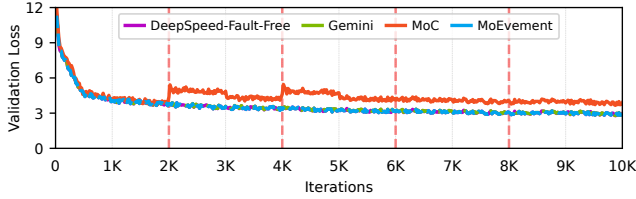


Figure 12: Validation loss for DeepSeek-MoE during 10K iterations of training, with failures injected at 2K, 4K, 6K, 8K iterations.

Task	#Shot	DeepSpeed Fault-Free	Gemini	MoC	MoEvemnt
PIQA [6]	0-shot	72.4	72.5	61.2	72.4
HellaSwag [88]	0-shot	68.9	68.8	53.1	68.8
TriviaQA [38]	5-shot	54.8	54.6	37.5	55.1
NaturalQuestions [41]	5-shot	15.3	15.1	6.3	15.3

Table 5: Downstream evaluation on commonsense reasoning and knowledge-intensive tasks. We compare fault-free DeepSpeed (baseline) against Gemini, MoC, and MoEvemnt for DeepSeek-MoE. **Best** and **worst** results highlighted in green and red, respectively.

5.5 Impact of Failures on Model Accuracy

To evaluate how checkpointing and recovery affect model quality, we train DeepSeek-MoE for 10,000 iterations, injecting faults at iterations 2K, 4K, 6K, and 8K (marked by red vertical lines in Fig. 12). The validation loss trajectories for Gemini, MoC, MoEvemnt, and a fault-free DeepSpeed baseline reveal that MoC exhibits validation loss spikes after the first two failures due to token loss from partial recovery. After iteration 4K, MoC begins checkpointing all experts to prevent further degradation, avoiding new spikes but never regaining fault-free performance. Gemini and MoEvemnt closely track the fault-free DeepSpeed baseline, demonstrating that both effectively preserve model quality despite repeated failures.

We further assess downstream performance (on a 0–100 scale, higher is better) on PIQA [6] and HellaSwag [88] (commonsense reasoning) and TriviaQA [38] and NaturalQuestions [41] (knowledge-intensive QA) as shown in Table 5. MoEvemnt achieves comparable accuracy to fault-free DeepSpeed and Gemini baselines across all tasks, confirming no impact on downstream model quality. MoC, however, consistently under performs, highlighting the negative consequences of partial recovery and reinforcing the importance of preserving synchronous training semantics.

5.6 MOEVEMENT Performance Breakdown

What is each MoEvemnt technique’s contribution to performance? We evaluate the incremental impact of *sparse checkpointing*, *skipping B_{weight} for frozen operators*, *popularity based reordering*, and *upstream logging* on training efficiency (ETTR) of MoEvemnt. Figure 13 reports ETTR for four models as each technique is added; green annotations show the relative improvement at each step.

Sparse checkpointing establishes the baseline, yielding ETTRs of 0.57 (MoE-LLaVa), 0.43 (GPT-MoE), 0.37 (QWen-

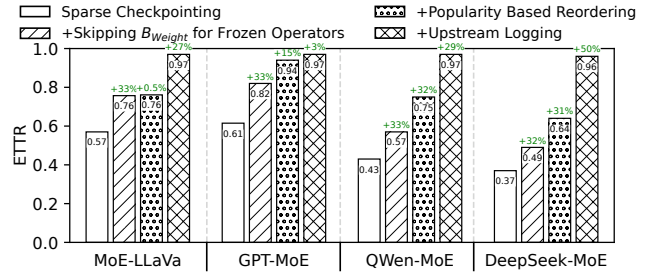


Figure 13: Incremental impact of MoEvemnt’s techniques across models. Each addition reduces recovery overhead, improving ETTR.

Model	Gemini		MoEvemnt		Increase over Gemini (%)
	GPU	CPU	GPU	CPU (X + Y)	
MoE-LLaVa	0	75.4	0	83.1 (81.2 + 1.9)	+10.1%
GPT-MoE	0	189.8	0	209.4 (205.2 + 4.2)	+10.3%
QWen-MoE	0	371.6	0	433.2 (420.8 + 12.4)	+16.5%
DeepSeek-MoE	0	426.4	0	499.8 (478.7 + 21.1)	+17.2%

Table 6: Memory footprint (in GB) of Gemini and MoEvemnt on GPU and CPU. For MoEvemnt, CPU memory footprint is decomposed as $X + Y$, with X representing the sparse checkpoint size (in GB) and Y representing the activation and gradient log size (in GB).

MoE), and 0.49 (DeepSeek-MoE). *Skipping B_{weight} and optimizer updates for frozen operators*—those lacking FP32 state—reduces recovery cost during sparse-to-dense conversion by $\approx 33\%$, as their role is limited to forward and input-gradient propagation. *Popularity-based reordering* defers the checkpointing of frequently activated experts, keeping them frozen longer and extending the compute savings. Its impact grows with expert count: models like QWen-MoE and DeepSeek-MoE (64 experts) see ETTR improve by 32%, GPT-MoE (32 experts) by 15%, while MoE-LLaVa (4 experts) sees no change. With more experts, the checkpointing order has greater influence over how long operators remain frozen, making reordering more effective. Finally, *upstream logging* confines recovery to the failed stage’s data-parallel group by logging activations and gradients in host memory. This reduces rollback scope and eliminates pipeline bubbles, raising ETTR to ≈ 0.97 across all models. The largest gain (+50%) is seen in DeepSeek-MoE due to its 12-stage pipeline, while the smallest improvement is in GPT-MoE, whose shallow pipeline limits the benefit of localized recovery.

How much extra memory does MoEvemnt use? MoEvemnt adds *no* GPU memory overhead; all additional state lives in CPU memory, consistent with Gemini. The extra CPU memory usage arises from two sources: (i) FP16 compute weights in sparse checkpoint (X) for *frozen* operators awaiting their full FP32 training state, and (ii) activation and gradient logs (Y) for localized recovery. Across the evaluated models in Table 6, the increase is at most 17.2% relative to Gemini’s dense checkpoint (and similarly for CheckFreq, omitted for brevity). This modest increase, $\leq 2\%$ of the available 10TB CPU memory, enables MoEvemnt to sustain high ETTR without compromising synchronous training semantics.

Training Configuration			Checkpointing Interval (iterations)				Avg. Per-Iteration Checkpointing Overhead (seconds, overhead % in parentheses)				MTBF	Total Recovery Time (seconds)				Effective Training Time Ratio (ETTR)				
Compute Weight	Master Weight	Optimizer State	CheckFreq	Gemini	MoC	MoEvent	CheckFreq	Gemini	MoC	MoEvent		CheckFreq	Gemini	MoC	MoEvent	CheckFreq	Gemini	MoC	MoEvent	
FP16	FP16	FP16 + FP16 [87]	77	71	1	1	(W _{sparse} = 3)	0.11 (3%)	1.16 (39%)	2.91 (97%)	0.06 (2%)	1H	2036	1854	79	86	0.918	0.925	0.613	0.984
				48				0.16 (6%)	0.876			0.886	0.459	0.979						
				27				0.29 (10%)	3.51 (117%)			10M	10648	4324	246	268	0.706	0.885	0.425	0.973
FP8	FP32	FP32 + FP32 [55]	227	182	1	1	(W _{sparse} = 6)	0.09 (4%)	3.53 (154%)	8.89 (386%)	0.05 (2%)	1H	5162	3591	48	59	0.853	0.878	0.357	0.952
				94				0.17 (8%)	0.748			0.839	0.196	0.946						
				46				0.34 (15%)	10.70 (465%)			10M	28574	5446	185	232	0.302	0.718	0.115	0.941
FP8	FP16	FP32 + FP32 [52]	205	157	1	1	(W _{sparse} = 4)	0.08 (4%)	2.88 (137%)	7.25 (345%)	0.04 (2%)	1H	3763	2846	45	53	0.881	0.893	0.368	0.966
				79				0.16 (8%)	0.793			0.852	0.226	0.958						
				43				0.30 (14%)	8.73 (416%)			10M	22581	4756	151	172	0.439	0.738	0.163	0.953
FP8	FP16	FP8 + FP16 [64]	94	68	1	1	(W _{sparse} = 3)	0.10 (5%)	1.16 (61%)	2.92 (154%)	0.04 (2%)	1H	1554	1108	52	57	0.916	0.923	0.557	0.976
				61				0.11 (6%)	0.889			0.894	0.349	0.968						
				29				0.22 (12%)	3.52 (185%)			10M	9322	2934	196	213	0.739	0.806	0.321	0.965
FP8	FP8	FP8 + FP16 [64]	78	59	1	1	(W _{sparse} = 3)	0.09 (5%)	0.87 (51%)	2.19 (129%)	0.03 (1%)	1H	1153	876	40	44	0.937	0.943	0.584	0.979
				52				0.18 (10%)	0.905			0.907	0.381	0.976						
				24				0.31 (17%)	2.63 (155%)			10M	6921	2100	162	178	0.752	0.816	0.326	0.971

Table 7: Comparison of checkpointing techniques across low precision training configurations for DeepSeek-MoE over a 12-hour run.

5.7 Low-Precision Training Beyond FP16 & FP32

Low-precision training has emerged as a promising approach for scaling next-generation models by accelerating iterations, reducing memory footprint, and lowering communication overheads [42]. Recent GPUs such as Nvidia H100 introduce native support for FP8 datatypes [61], making low-precision training increasingly practical at scale. To evaluate checkpointing performance in this regime, we use a private cluster with 16 nodes, each equipped with a 104-core Intel Xeon processor, 2.1 TB of RAM, and eight Nvidia H100 80 GB GPUs. Within a node, GPUs are connected by 900 GB/s NVLink, and across nodes by 200 Gbps InfiniBand. We evaluate all four systems across five low-precision configurations proposed in prior work [52, 55, 64, 87], training DeepSeek-MoE (Table 2) with 8-way pipeline parallelism, 2-way data parallelism, and 8-way expert parallelism, using a batch size of 512 (micro-batch size 32) and a sequence length of 2048 tokens.

The five configurations vary precision used for computations, master weights, and optimizer state. The choice of precision impacts two key variables: iteration time and snapshot size. Switching compute from FP16 to FP8 shortens iterations, shrinking the window to overlap snapshot I/O. Lowering the precision of training state (e.g., FP8 master weights and FP8+FP16 optimizer state) reduces the snapshot size by as much as 66%. Dense baselines are limited by the tighter of these two constraints: shorter iterations force longer intervals, while smaller snapshots from lower-precision states enable shorter intervals. For CheckFreq, we configure its policy module to choose intervals that maintain runtime overhead below 3%. For Gemini, we apply an oracle policy: for each MTBF, we sweep intervals offline and select the one maximizing ETTR; this hindsight-informed choice requires knowledge unavailable at runtime, thus upper-bounding Gemini’s achievable performance. MoC and MoEvent both checkpoint every iteration, albeit selectively.

As shown in Table 7, CheckFreq’s shortest feasible interval (78 iterations) occurs when training state uses FP8 master weights and FP8+FP16 optimizer states. When using FP32 optimizer state with FP16 or FP32 model weights, CheckFreq is forced to lengthen intervals significantly, to 205 and 227 iterations respectively. Gemini follows a similar trend

at MTBF=1H, increasing its interval from 59 to 182 as state precision rises; under frequent failures its oracle policy trades higher runtime overhead for shorter rollbacks, checkpointing every 24-94 iterations at 6-13% per-iteration overhead. MoC’s overhead grows with failure frequency because it exhausts its lost-token budget sooner and is forced to increase experts checkpointed per iteration—eventually checkpointing all experts—and incurring 39–465% per-iteration overhead. In contrast, MoEvent maintains 1–2% overhead across all configurations with small sparse windows.

These interval choices directly shape recovery costs and ETTR. At higher precision (FP32), dense baselines suffer from long intervals and high recovery costs. At MTBF=1H, CheckFreq’s recovery time ranges from 1153 to 5162 secs, and Gemini’s from 876 to 3591 secs. At MTBF=10M, these costs increase to 6921–28574 secs (CheckFreq) and 2100–5446 secs (Gemini), even when Gemini selects intervals using an oracle. As precision decreases (FP8 master weights with FP8+FP16 optimizer state), dense baselines improve: Gemini’s ETTR at MTBF=10M rises from 0.718 to 0.816. However, CheckFreq and Gemini still sacrifice 10–25% ETTR under frequent failures. MoC consistently underperforms across all precisions, with ETTR dropping from 0.36–0.61 at MTBF=1H to 0.12–0.43 at MTBF=10M. In contrast, MoEvent maintains low and stable overhead (1–2%), consistently achieving ETTR of 0.94–0.98 across all precisions and MTBFs. Consequently, MoEvent provides up to an 8× improvement in end-to-end training time.

6 Conclusion

We presented MoEvent, a distributed, in-memory checkpointing system designed for efficient and reliable MoE model training. MoEvent introduces three key techniques—*sparse checkpointing*, *sparse-to-dense checkpoint conversion*, and *lightweight upstream activation and gradient logging*—which together break the runtime–recovery trade-off, mitigate the correctness–efficiency tension, and eliminate excessive global recomputation in prior approaches. Our evaluation shows that MoEvent sustains $ETTR \geq 0.94$ even at MTBFs as low as 10 minutes, achieving up to an 8× end-to-end training speedup without compromising model accuracy.

Acknowledgements

We are grateful to the anonymous reviewers and to our shepherd, Dan Li, whose comments have greatly helped improve this paper. This research was partly supported by the Stanford Platform Lab and its affiliates, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

- [1] Amey Agrawal, Sameer Reddy, Satwik Bhattamishra, Venkata Prabhakara Sarath Nookala, Vidushi Vashishth, Kexin Rong, and Alexey Tumanov. Inshrinkerator: Compressing Deep Learning Training Checkpoints via Dynamic Quantization. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 1012–1031, New York, NY, USA, 2024. Association for Computing Machinery.
- [2] Meta AI. Meta Llama 3. <https://ai.meta.com/blog/meta-llama-3/>, 2024.
- [3] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. Pathways: Asynchronous Distributed Dataflow for ML. In *Proceedings of Machine Learning and Systems*, 2022.
- [4] Romain Beaumont. Large Scale OpenCLIP: L/14, H/14 and G/14 Trained on LAION-2B. <https://laion.ai/blog/large-openclip/>, 2022.
- [5] BigScience Workshop. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. *arXiv e-prints*, page arXiv:2211.05100, November 2022.
- [6] Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. PIQA: Reasoning about Physical Commonsense in Natural Language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7432–7439, 2020.
- [7] Weilin Cai, Juyong Jiang, Fan Wang, Jing Tang, Sunghun Kim, and Jiayi Huang. A Survey on Mixture of Experts in Large Language Models. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–20, 2025.
- [8] Weilin Cai, Le Qin, and Jiayi Huang. 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*, ASPLOS '25, page 655–671, New York, NY, USA, 2025. Association for Computing Machinery.
- [9] BLOOM Chronicles. BLOOM Chronicles. <https://github.com/bigscience-workshop/bigscience/blob/master/train/tr11-176B-ml/chronicles.md>, 2022.
- [10] Ben Cottier, Robi Rahman, Loredana Fattorini, Nestor Maslej, Tamay Besiroglu, and David Owen. The Rising Costs of Training Frontier AI Models. *arXiv preprint arXiv:2405.21015*, 2024.
- [11] Justin Cui, Ruochen Wang, Si Si, and Cho-Jui Hsieh. Scaling Up Dataset Distillation to ImageNet-1K with Constant Memory. In *International Conference on Machine Learning*, pages 6565–6590. PMLR, 2023.
- [12] Damai Dai, Chengqi Deng, Chenggang Zhao, RX Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Yu Wu, et al. DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models. *arXiv preprint arXiv:2401.06066*, 2024.
- [13] John T Daly. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future generation computer systems*, 22(3):303–312, 2006.
- [14] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. GLaM: Efficient scaling of language models with mixture-of-experts. In *International conference on machine learning*, pages 5547–5569. PMLR, 2022.
- [15] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, Renton, WA, April 2022. USENIX Association.
- [16] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *J. Mach. Learn. Res.*, 23(1), January 2022.
- [17] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. ReCycle: Resilient Training of Large DNNs using Pipeline Adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 211–228, New York, NY, USA, 2024. Association for Computing Machinery.
- [18] Google. A new era of intelligence with Gemini 3. <https://blog.google/products/gemini/gemini-3/>, 2025.

- [19] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv e-prints*, page arXiv:1706.02677, June 2017.
- [20] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948*, 2025.
- [21] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in Large Scale Systems: Long-term Measurement, Analysis, and Implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.
- [22] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. Just-In-Time Checkpointing: Low Cost Error Recovery from Deep Learning Training Failures. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 1110–1125, New York, NY, USA, 2024. Association for Computing Machinery.
- [23] Ethan He, Abhinav Khattar, Ryan Prenger, Vijay Korthikanti, Zijie Yan, Tong Liu, Shiqing Fan, Ashwath Aithal, Mohammad Shoeybi, and Bryan Catanzaro. Upcycling Large Language Models into Mixture of Experts. *arXiv preprint arXiv:2410.07524*, 2024.
- [24] Jiaao He, Jiezhong Qiu, Aohan Zeng, Zhilin Yang, Jidong Zhai, and Jie Tang. FastMoE: A Fast Mixture-of-Expert Training System. *arXiv preprint arXiv:2103.13262*, 2021.
- [25] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. FasterMoE: Modeling and Optimizing Training of Large-Scale Dynamic Pre-Trained Models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '22, page 120–134, New York, NY, USA, 2022. Association for Computing Machinery.
- [26] Tao He, Xue Li, Zhibin Wang, Kun Qian, Jingbo Xu, Wenyuan Yu, and Jingren Zhou. Unicron: Economizing Self-Healing LLM Training at Scale. *arXiv e-prints*, page arXiv:2401.00134, December 2023.
- [27] Xu Owen He. Mixture of A Million Experts. *arXiv preprint arXiv:2407.04153*, 2024.
- [28] Thomas Herault and Yves Robert. *Fault-Tolerance Techniques for High-Performance Computing*. Springer Publishing Company, Incorporated, 1st edition, 2015.
- [29] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack W. Rae, and Laurent Sifre. Training Compute-Optimal Large Language Models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.
- [30] 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. Characterization of Large Language Model Development in the Datacenter. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*, NSDI'24, USA, 2024. USENIX Association.
- [31] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *Advances in neural information processing systems*, 32, 2019.
- [32] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, HoYuen Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. Tutel: Adaptive Mixture-of-Experts at Scale. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 269–287. Curran, 2023.
- [33] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 382–395, New York, NY, USA, 2023. Association for Computing Machinery.
- [34] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association.
- [35] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas,

- Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, L lio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Th ophile Gervet, Thibaut Lavril, Thomas Wang, Timoth e Lacroix, and William El Sayed. Mixtral of Experts. *arXiv preprint arXiv:2401.04088*, 2024.
- [36] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, Santa Clara, CA, April 2024. USENIX Association.
- [37] Norman Lloyd Johnson, Samuel Kotz, and Narayanaswamy Balakrishnan. *Continuous Multivariate Distributions*, volume 7. Wiley New York, 1972.
- [38] Mandar Joshi, Eunsol Choi, Daniel Weld, and Luke Zettlemoyer. TriviaQA: A Large Scale Distantly Supervised Challenge Dataset for Reading Comprehension. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1601–1611, Vancouver, Canada, July 2017. Association for Computational Linguistics.
- [39] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. BPIPE: Memory-Balanced Pipeline Parallelism for Training Large Language Models. In *Proceedings of the 40th International Conference on Machine Learning, ICML 23*. JMLR.org, 2023.
- [40] Apostolos Kokolis, Michael Kuchnik, John Hoffman, Adithya Kumar, Parth Malani, Faye Ma, Zachary DeVito, Shubho Sengupta, Kalyan Saladi, and Carole-Jean Wu. Revisiting Reliability in Large-Scale Machine Learning Research Clusters. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1259–1274. IEEE, 2025.
- [41] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural Questions: A Benchmark for Question Answering Research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.
- [42] Joonhyung Lee, Jeongin Bae, Byeongwook Kim, Se Jung Kwon, and Dongsoo Lee. To FP8 and Back Again: Quantifying Reduced Precision Effects on LLM Training Stability, 2025.
- [43] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *The Ninth International Conference on Learning Representations*, 2021.
- [44] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.*, 13(12):3005–3018, aug 2020.
- [45] Shigang Li and Torsten Hoefler. Chimera: Efficiently Training Large-Scale Neural Networks with Bidirectional Pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC ’21*, New York, NY, USA, 2021. Association for Computing Machinery.
- [46] Bin Lin, Zhenyu Tang, Yang Ye, Jiaxi Cui, Bin Zhu, Peng Jin, Jinfa Huang, Junwu Zhang, Yatian Pang, Munan Ning, Jiebo Luo, and Li Yuan. MoE-LLaVA: Mixture of Experts for Large Vision-Language Models. *arXiv preprint arXiv:2401.15947*, 2024.
- [47] Aixiu Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437*, 2024.
- [48] Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. *arXiv e-prints*, page arXiv:1711.05101, November 2017.
- [49] Jan Ludziejewski, Jakub Krajewski, Kamil Adamczewski, Maciej Pi ro, Micha  Krutul, Szymon Antoniak, Kamil Ciebiera, Krystian Kr l, Tomasz Odrzyg dz, Piotr Sankowski, Marek Cygan, and Sebastian Jaszczur. Scaling Laws for Fine-Grained Mixture of Experts. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 33270–33288. PMLR, 21–27 Jul 2024.
- [50] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyang Yang, Mike Rabbat, Brandon Lucia, and Carole-Jean Wu. CPR: Understanding and Improving Failure

Tolerant Training for Deep Learning Recommendation with Partial Recovery. *Proceedings of Machine Learning and Systems*, 3:637–651, 2021.

- [51] Avinash Maurya, Robert Underwood, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. DataStates-LLM: Lazy Asynchronous Checkpointing for Large Language Models. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC '24*, page 227–239, New York, NY, USA, 2024. Association for Computing Machinery.
- [52] Naveen Mellempudi, Sudarshan Srinivasan, Dipankar Das, and Bharat Kaul. Mixed Precision Training With 8-bit Floating Point. *arXiv preprint arXiv:1905.12334*, 2019.
- [53] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. Convergence Analysis of Distributed Stochastic Gradient Descent with Shuffling. *Neurocomputing*, 337:46–57, 2019.
- [54] Meta. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. <https://ai.meta.com/blog/llama-4-multimodal-intelligence/>, 2025.
- [55] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. FP8 Formats for Deep Learning. *arXiv preprint arXiv:2209.05433*, 2022.
- [56] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216. USENIX Association, February 2021.
- [57] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.
- [58] 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. 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, SC '21*, New York, NY, USA, 2021. Association for Computing Machinery.
- [59] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, Krishnakumar Nair, Isabel Gao, Bor-Yiing Su, Jiyang Yang, and Mikhail Smelyanskiy. Deep Learning Training in Facebook Data Centers: Design of Scale-up and Scale-out Systems. *arXiv e-prints*, page arXiv:2003.09518, March 2020.
- [60] Xiaonan Nie, Xupeng Miao, Zilong Wang, Zichao Yang, Jilong Xue, Lingxiao Ma, Gang Cao, and Bin Cui. Flex-MoE: Scaling Large-scale Sparse Pre-trained Model Training via Dynamic Device Placement. *Proc. ACM Manag. Data*, 1(1), May 2023.
- [61] Nvidia. NVIDIA H100 Tensor Core GPU. <https://www.nvidia.com/en-us/data-center/h100/>, 2024.
- [62] OpenAI. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774*, 2024.
- [63] OpenAI. Introducing gpt-oss. <https://openai.com/index/introducing-gpt-oss/>, 2025.
- [64] Houwen Peng, Kan Wu, Yixuan Wei, Guoshuai Zhao, Yuxiang Yang, Ze Liu, Yifan Xiong, Ziyue Yang, Bolin Ni, Jingcheng Hu, Ruihang Li, Miaosen Zhang, Chen Li, Jia Ning, Ruizhe Wang, Zheng Zhang, Shuguang Liu, Joe Chau, Han Hu, and Peng Cheng. FP8-LM: Training FP8 Large Language Models, 2023.
- [65] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric Xing. Fault Tolerance in Iterative-Convergent Machine Learning. In *International Conference on Machine Learning*, pages 5220–5230. PMLR, 2019.
- [66] Zihan Qiu, Zeyu Huang, Bo Zheng, Kaiyue Wen, Zekun Wang, Rui Men, Ivan Titov, Dayiheng Liu, Jingren Zhou, and Junyang Lin. Demons in the Detail: On Implementing Load Balancing Loss for Training Specialized Mixture-of-Expert Models. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5005–5018, Vienna, Austria, July 2025. Association for Computational Linguistics.
- [67] Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling Language Models: Methods, Analysis &

- Insights from Training Gopher. *arXiv preprint arXiv:2112.11446*, 2021.
- [68] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale. In *International conference on machine learning*, pages 18332–18346. PMLR, 2022.
- [69] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 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, 2020.
- [70] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21*, New York, NY, USA, 2021. Association for Computing Machinery.
- [71] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD '20*, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery.
- [72] Stephen A Rhoades. The Herfindahl-Hirschman Index. *Federal Reserve Bulletin*, 79:188, 1993.
- [73] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv e-prints*, page arXiv:1802.05799, February 2018.
- [74] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *International Conference on Learning Representations*, 2017.
- [75] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR*, abs/1909.08053, 2019.
- [76] Foteini Strati, Michal Friedman, and Ana Klimovic. PC-check: Persistent Concurrent Checkpointing for ML. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS '25*, page 811–827, New York, NY, USA, 2025. Association for Computing Machinery.
- [77] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, Boston, MA, April 2023. USENIX Association.
- [78] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, Xin Liu, and Chuan Wu. ByteCheckpoint: A Unified Checkpointing System for Large Foundation Model Development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 559–578, Philadelphia, PA, April 2025. USENIX Association.
- [79] An Wang, Xingwu Sun, Ruobing Xie, Shuai Li, Jiaqi Zhu, Zhen Yang, Pinxue Zhao, J. N. Han, Zhanhui Kang, Di Wang, Naoaki Okazaki, and Chengzhong Xu. HMoE: Heterogeneous mixture of experts for language modeling. *arXiv preprint arXiv:2408.10681*, 2024.
- [80] Guanhua Wang, Olatunji Ruwase, Bing Xie, and Yuxiong He. FastPersist: Accelerating Model Checkpointing in Deep Learning. *arXiv preprint arXiv:2406.13768*, 2024.
- [81] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. Lineage Stash: Fault Tolerance Off the Critical Path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, page 338–352, New York, NY, USA, 2019. Association for Computing Machinery.
- [82] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP '23*, page 364–381, New York, NY, USA, 2023. Association for Computing Machinery.
- [83] Maurice Weber, Daniel Y Fu, Quentin Gregory Anthony, Yonatan Oren, Shane Adams, Anton Alexandrov, Xiaozhong Lyu, Huu Nguyen, Xiaozhe Yao, Virginia Adams, Ben Athiwaratkun, Rahul Chalamala, Kezhen Chen, Max Ryabinin, Tri Dao, Percy Liang, Christopher Re, Irina Rish, and Ce Zhang. RedPajama: An Open Dataset for Training Large Language Models. In *The*

Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track, 2024.

- [84] xAI. Grok-4. <https://x.ai/news/grok-4>, 2025.
- [85] Shen Yan, Xingyan Bin, Sijun Zhang, Yisen Wang, and Zhouchen Lin. TC-MoE: Augmenting Mixture of Experts with Ternary Expert Choice. In *The Thirteenth International Conference on Learning Representations*, 2025.
- [86] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2.5 Technical Report. *arXiv preprint arXiv:2412.15115*, 2024.
- [87] Tao Yu, Gaurav Gupta, Karthick Gopalswamy, Amith Mamidala, Hao Zhou, Jeffrey Huynh, Youngsuk Park, Ron Diamant, Anoop Deoras, and Luke Huan. Collage: Light-weight low-precision strategy for LLM training. *arXiv preprint arXiv:2405.03637*, 2024.
- [88] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a Machine Really Finish Your Sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.
- [89] Mingshu Zhai, Jiaao He, Zixuan Ma, Zan Zong, Runqing Zhang, and Jidong Zhai. SmartMoE: Efficiently Training Sparsely-Activated Models through Combining Offline and Online Parallelization. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 961–975, Boston, MA, July 2023. USENIX Association.
- [90] 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. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, August 2023.
- [91] Barret Zoph, Irwan Bello, Sameer Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam Shazeer, and William Fedus. ST-MoE: Designing Stable and Transferable Sparse Expert Models. *arXiv preprint arXiv:2202.08906*, 2022.
- [92] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, Steve Lacy, Hang Wang, Aaron Wisner, Chris Lewis, and Henri Bahini. Resiliency at Scale: Managing Google’s TPUv4 Machine Learning Supercomputer. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 761–774, Santa Clara, CA, April 2024. USENIX Association.

A Recovery from Concurrent Failures

Concurrent failures during distributed training can occur in two scenarios: *multiple simultaneous failures*, where two or more workers fail concurrently, and *cascading failures*, where additional failures occur during ongoing recovery operations. **Multiple Simultaneous Failures.** Upon detecting a failure, MoEvent pauses all workers and aborts the current iteration. Failed workers are promptly replaced with healthy spares. Only the data-parallel (DP) groups containing the failed workers roll back to their most recent sparse checkpoints, typically captured within the last W_{sparse} iterations. Unaffected workers remain paused to maintain global consistency. If the failed workers form a contiguous pipeline segment (Fig. 14 (right)), MoEvent initiates a *joint localized recovery*. During this joint recovery, boundary stages adjacent to the affected segment supply logged activations and gradients, enabling the impacted DP groups to collaboratively replay computations and convert their sparse checkpoints into a dense checkpoint. Conversely, if failures involve nonadjacent workers, each affected DP group independently and concurrently executes its own localized recovery. In both cases, overall recovery time is determined by the slowest individual or joint recovery, after which training resumes across all workers.

Cascading Failures. MoEvent dynamically adapts the recovery scope in response to cascading failures. If a subsequent failure occurs in a worker that is adjacent to, or already part of, an ongoing recovery, MoEvent expands the recovery scope to include this newly failed worker, forming an enlarged contiguous segment, and restarts *joint localized recovery* for the affected DP groups. During recovery, each DP group rolls back to the most recent persisted sparse checkpoint common to that region, and replays computations by utilizing activation and gradient logs from healthy boundary neighbors. If the new failure is disjoint from ongoing recoveries, existing recoveries proceed without interruption, and the new failure triggers a separate, independent recovery.

B Alternative Operator Ordering Schemes

While MoEvent defaults to ordering experts by ascending popularity (i.e., activation count) within each sparse checkpoint window, the `OrderOperators()` interface supports alternative ordering schemes to accommodate different model and workload characteristics:

Soft-Count Popularity. Instead of using binary (hard) activation counts, popularity can be estimated more smoothly by aggregating gating probabilities. Let $\mathcal{P}_j^l(x_i)$ represent the gating probability assigned to expert j in layer l for token x_i , where \mathcal{D} is the training dataset. Then, the soft-count popularity is defined as:

$$\mathcal{A}_j^l = \sum_{x_i \in \mathcal{D}} \mathcal{P}_j^l(x_i)$$

Time-Decayed Popularity. To adaptively track popularity over changing activation patterns during training, we use an exponential moving average over recent mini-batches. Let \mathcal{B}_t

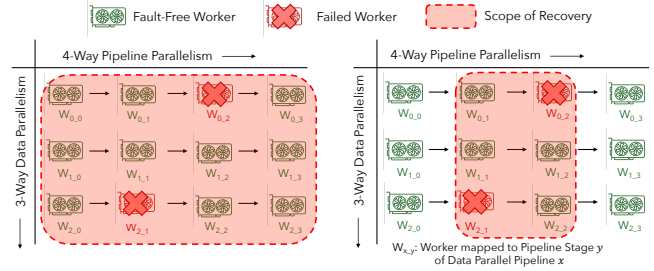


Figure 14: Scope of recovery from multiple simultaneous failures *without* (left) and *with* (right) localized recovery.

be the set of tokens in mini-batch t , and let $\mathbf{1}[\gamma(e_j^l, x_i) = 1]$ indicate whether expert j in layer l is activated for token x_i , where $\gamma(e_j^l, x_i)$ is a binary indicator function, and α controls the rate of decay. The time-decayed popularity is:

$$\mathcal{A}_j^l(t) = \alpha \times \mathcal{A}_j^l(t-1) + (1-\alpha) \times \sum_{x_i \in \mathcal{B}_t} \mathbf{1}[\gamma(e_j^l, x_i) = 1]$$

Capacity-Aware Ordering. For heterogeneous experts [79, 85] (e.g., with different capacity factors), ordering can be weighted by utilization relative to capacity, prioritizing those with lower relative utilization. If each expert j in layer l has a capacity factor C_j^l (maximum tokens it can process per batch), we define a capacity-normalized popularity score:

$$\hat{\mathcal{A}}_j^l = \frac{\mathcal{A}_j^l}{C_j^l}$$

C Simulator for Estimating ETTR

We implement a simulator that estimates iteration time and Effective Training Time Ratio (ETTR) given a model architecture, cluster configuration, parallelization plan, and Mean Time Between Failures (MTBF). We validate its accuracy against measurements collected from Azure using two MoE models under diverse failure scenarios (Table 4). Validation results show a maximum deviation of 1.47%, primarily due to runtime variability in NCCL operations. These small variations have negligible impact on observed relative performance trends, demonstrating the simulator’s effectiveness in evaluating distributed training performance without requiring direct access to large-scale GPU clusters. We describe its implementation below:

Inputs. Our simulator requires four primary inputs: (1) a training job specification, including the model architecture, `global_batch_size`, optimizer configuration, and hyperparameters such as `micro_batch_size`; (2) a cluster configuration specifying GPU type and count, GPU allocation per node, inter-node bandwidth, topology, and hardware heterogeneity; and (3) a parallelization plan describing pipeline stage partitioning, tensor- and data-parallel degrees, and MoE expert placement or sharding, where applicable; and (4) a Mean Time Between Failures (MTBF) to characterize failure rates and corresponding recovery overhead.

Profiler. We derive computational and communication costs from empirical profiling of real training runs. The profiler captures compute and memory requirements by executing the training job on a single GPU node for each GPU node type available in the resource pool. To minimize profiling overhead, repeated layers are represented by a single instance (e.g., one transformer layer for large language models). We employ PyTorch hooks in DeepSpeed to measure the forward pass, backward pass, and parameter update times for varying micro batch sizes and tensor parallel degrees, using CUDA events to ensure accurate GPU timing measurements. Additionally, the profiler records parameters count, output activation sizes, and intermediate memory consumption per layer using PyTorch’s CUDA memory allocator. Profiling overhead is minimal, typically completing in a few minutes.

For cluster-level profiling, the simulator gathers network bandwidth data between pairs of machine types. Since network bandwidth varies with message size, we measure bandwidth using PyTorch collectives with the NCCL backend across multiple message sizes, fitting a polynomial function to derive bandwidth coefficients for each node-type pair. NCCL collective operations use an affine model:

$$T_{\text{NCCL}}(m, p) = \alpha(p) + \beta(p) \times m$$

where m is message size, and p is group size. This captures NCCL-specific algorithmic behavior and network characteristics.

Estimating Iteration-time. Each iteration runs forward and backward passes over the global batch and then applies an optimizer update. We partition the global batch B into M micro batches and execute them with pipeline parallelism using an interleaved 1F1B schedule that proceeds through warm-up, steady state, and cool-down phases. Let S be the number of pipeline stages, and let t_s denote the measured per-micro-batch time for stage s (forward, backward, and local communication). For a single data-parallel pipeline, the time spent on the forward and backward passes is:

$$T_{\text{pipeline}} = (M + S - 1) \times \max_{s \in [1, S]} (t_s)$$

To estimate iteration time, we combine pipeline execution time with global synchronization overhead from all-reduce and optimizer update time:

$$T_{\text{iter}} = \max_{i \in [1, PP]} (T_{\text{pipeline}}^i) + T_{\text{sync}} + T_{\text{update}}$$

where \max is taken over PP data-parallel pipelines to account for potential stragglers due to network variability. We estimate T_{sync} from empirically measured intra- and inter-node bandwidths, and we incorporate observed overlap between computation and communication rather than assuming full serialization. T_{update} is empirically profiled and reflects the time needed to compute and apply parameter updates.

Estimating ETTR. ETTR quantifies training efficiency under failures by combining checkpointing overhead with recovery overhead. Modeling failures as a Poisson process with rate $\frac{1}{\text{MTBF}}$, ETTR is estimated as:

$$\text{ETTR} = \frac{1}{1 + \frac{T_{\text{ckpt}}}{T_{\text{iter}} \times \text{Ckpt}_{\text{interval}}}} \times \frac{1}{1 + \frac{\mathbb{E}[R]}{\text{MTBF}}}$$

Here, $\frac{T_{\text{ckpt}}}{T_{\text{iter}} \times \text{Ckpt}_{\text{interval}}}$ represents the runtime overhead incurred by checkpointing every $\text{Ckpt}_{\text{interval}}$ iterations, and $\mathbb{E}[R] \approx \frac{1}{2} \times \text{Ckpt}_{\text{interval}} \times T_{\text{iter}}$ is the expected recovery time per failure under a uniform failure distribution within a checkpoint interval. This formulation explicitly captures the trade-off involved in checkpoint interval selection: longer intervals lower runtime overhead but increase potential recomputation after failures, whereas shorter intervals reduce recovery overhead but incur higher runtime checkpointing costs.

D Effect of Varying Expert Popularity

Mixture-of-Experts (MoE) models frequently exhibit skewed activation patterns, with some experts consistently receiving more tokens despite the use of auxiliary load-balancing loss, and this skew often varies over time and across layers [24, 25, 60, 89]. To analyze how varying degrees of skewness in expert popularity affect MoEvent’s performance, we quantify skewness using the normalized Herfindahl–Hirschman Index (HHI) [72], defined as follows:

$$\text{HHI} = \sum_{i=1}^E p_i^2 \quad \text{and} \quad S = \frac{\text{HHI} - \frac{1}{E}}{1 - \frac{1}{E}}$$

where $p = (p_1, \dots, p_E)$ represents the fraction of tokens assigned to each expert, with $\sum p_i = 1$, and E is the number of experts ($E \geq 2$). The skewness metric S ranges from 0 for perfectly uniform popularity (each expert receives an equal share of tokens) to 1 for maximum skewness (one expert receives all tokens). We systematically vary intermediate levels of skewness by sampling expert token distributions p from a symmetric Dirichlet distribution [37] parameterized by α . Large values of α yield nearly uniform distributions, while small values yield highly skewed distributions. The expected HHI ($\mathbb{E}[\text{HHI}]$) and skewness ($\mathbb{E}[S]$) can be computed as:

$$\mathbb{E}[\text{HHI}] = \frac{\alpha + 1}{\alpha \times E + 1} \quad \text{and} \quad \mathbb{E}[S] = \frac{\mathbb{E}[\text{HHI}] - \frac{1}{E}}{1 - \frac{1}{E}}$$

We report the Effective Training Time Ratio (ETTR) for CheckFreq, Gemini, MoC, and MoEvent using DeepSeek-MoE with an MTBF of 10 minutes across varying skewness levels over a 12-hour long training run. All experiments use 96 A100 GPUs on Azure, employing standard FP16-FP32 mixed-precision [58, 67] training with the AdamW [48] optimizer (additional details about the test cluster can be found in §5.1). The target skewness levels $S \in \{0.25, 0.50, 0.75, 0.99\}$ correspond to $\alpha \approx \{0.0469, 0.0156, 0.0052, 0.000158\}$.

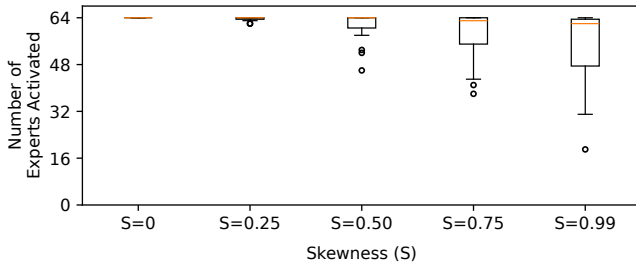


Figure 15: Box plot showing the number of experts activated (assigned at least one token) per iteration across varying expert popularity skewness. Despite skewness concentrating tokens among fewer experts, most experts remain active. Boxes extend from the first quartile (Q1) to the third quartile (Q3), with a line at the median.

Figure 16 shows that $S = 0$ (uniform popularity) represents the worst-case scenario for MoEvent since uniform distribution of tokens across experts provides no clear popular subset of experts to defer, thereby limiting the benefits of popularity-based expert reordering. As S increases to 0.25, 0.50, 0.75, and 0.99, MoEvent’s advantage over prior methods widens: higher skewness lets MoEvent defer the most popular experts longer within each sparse window, which reduces recomputation during sparse-to-dense conversion and lowers recovery overhead. Importantly, despite increased skewness concentrating tokens among fewer experts, Fig. 15 demonstrates that majority of experts remain active, i.e., they are assigned at least one token per iteration. This necessitates checkpointing all experts to avoid token loss upon failure, highlighting a crucial design consideration explicitly addressed by MoEvent. Specifically, MoEvent leverages popularity-based reordering to defer the most frequently activated experts, yet crucially ensures every expert, regardless of popularity, is checkpointed within each sparse window. This design preserves training consistency, completely eliminating any risk of token loss and thus maintaining equivalence to fault-free training semantics.

MoEvent’s design principle of zero token loss contrasts sharply with MoC, which inherently risks token loss due to its partial-checkpointing strategy. MoC checkpoints only a subset of experts per iteration in a round-robin fashion, making its ETTR sensitive to skewed expert popularity distributions. With increased skewness, a small subset of popular experts processes the majority of tokens. If a failure occurs after a popular expert has not been checkpointed for roughly $\frac{E}{K}$ iterations (where E is the total number of experts, and K is the number checkpointed per iteration), the resulting burst of token loss can rapidly exhaust MoC’s token-loss budget. To avoid exceeding this budget, MoC is forced to checkpoint more experts per iteration, directly increasing per-iteration runtime overhead due to checkpoint-induced stalls and thereby lowering its ETTR. Empirically, as skewness S increases from 0 (uniform popularity) to 0.99 (highly skewed popularity), MoC’s ETTR drops from 0.17 to 0.13 (Fig. 16).

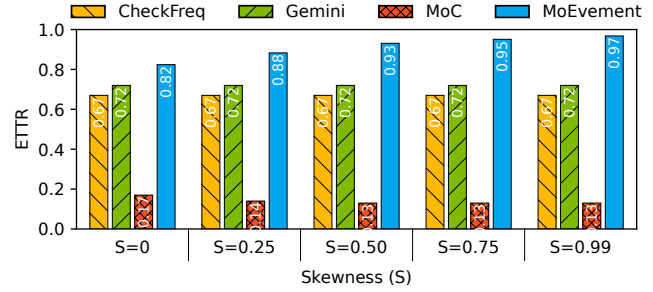


Figure 16: Impact of Expert Popularity Skewness on ETTR across CheckFreq, Gemini, MoC, and MoEvent using DeepSeek-MoE with varying levels of expert popularity skewness. Higher skewness enhances MoEvent’s advantage, as its popularity-based expert reordering reduces recovery overhead.

CheckFreq and Gemini avoid popularity-based differentiation by checkpointing all experts every $Ckpt_{interval}$ iterations. Consequently, their performance remains insensitive to variations in expert popularity skewness, resulting in stable but consistently lower ETTR (0.67 for CheckFreq and 0.72 for Gemini) compared to MoEvent. Overall, MoEvent outperforms prior approaches across all levels of skewness, achieving up to a $7.5\times$ reduction in end-to-end training time.

E Generalizing MoEvent to Dense Models

Sparse checkpointing in MoEvent leverages architectural properties unique to MoE models—each expert can be treated as an independent checkpointable unit, and the distribution of tokens across experts is both dynamic and inherently skewed (Fig. 4). Generalizing sparse checkpointing techniques to dense transformer models represents a compelling direction for future research. Dense transformer architectures typically consist of monolithic feed-forward layers, which cannot be readily partitioned at a finer granularity akin to MoE experts. Nevertheless, each transformer layer itself can be considered an independently checkpointable unit. Sparse checkpointing could therefore be adapted to dense models by incrementally checkpointing subsets of consecutive layers across multiple iterations. Given the inherent directional flow of activations (forward) and gradients (backward), selecting subsets of layers starting from the output (back layers) and progressing toward the input (front layers) can strategically reduce recomputation costs during recovery.

Moreover, the localized recovery mechanism in MoEvent, facilitated by upstream logging (§3.4), is directly applicable to dense models. Localized recovery can confine recomputation exclusively to affected data-parallel groups, thereby eliminating pipeline bubbles that typically occur during warm-up and cool-down phases of recovery. This advantage may become increasingly pronounced in training configurations with deep pipelines, which are common in large-scale transformer-based models. Quantifying and optimizing these benefits within dense model architectures represents an exciting and promising area for future work.