Purpose-Built Cloud for AI at Scale: Achieving 20% Higher MFU and 10× Reliability on Thousand-GPU Clusters

NVIDIA H100 Performance Benchmarks
A CoreWeave Technical Report

Wes Brown (Distinguished Engineer), David Marx (Senior Engineer), Anthony Mercurio (Engineer), Eta Syra (Engineer), Sanger Steel (Engineer), Rex Wang (Engineer), Deok Filho (Product Manager)

August 2025

Table of Contents

Executive Summary	2
Introduction: The Challenge of Large-Scale AI Training	4
Benchmarking Results	6
Benchmarking Methodology	12
The CoreWeave Approach: Purpose-Built for AI Scale	15

Technical Implementation	19
Best Practices for Large-Scale Training on CoreWeave	19
Conclusion: Reliable, Performant AI Training at Scale with CoreWeave	30
Appendix	34

Executive Summary

The artificial intelligence industry's push toward trillion-parameter models has exposed a critical gap: while GPU availability has increased, the infrastructure capability to maintain stable, efficient training at scale has not kept pace. Industry reports document effective training time ratios as low as 90%[1] and mean time to failure under 8 hours[2] for thousand-GPU clusters, directly impacting development costs and competitive timelines. For organizations investing in AI development, infrastructure-related failures can extend training times from weeks to months, delaying time-to-market in a rapidly evolving competitive landscape.

CoreWeave provides a specialized AI cloud platform meticulously optimized for large-scale, GPU-accelerated workloads, differentiating through bare metal performance, low-latency networking, flexible configurations, and deep AI/ML operational expertise. During a six-week period in May-June 2025, we performed LLM pre-training exercises to provide concrete evidence of CoreWeave's value proposition, demonstrating superior performance, reliability, and stability crucial for large model training.

CoreWeave minimizes downtime, averaging an **Effective Training Time Ratio (ETTR) of 98%** while maximizing computational efficiency, reducing costs, and accelerating time-to-market for large-scale AI. Our experiments demonstrate a **Mean Time To Failure (MTTF) of 3.66 days for a 1,024-GPU job**, representing a **43.7% improvement** on MTTF over a similarly trained industry model[1] when our results are projected up to **16,384 GPUs**.

Our benchmarking shows CoreWeave achieves **Model FLOPS Utilization (MFU) exceeding 50% on NVIDIA Hopper GPUs**. This level of efficiency represents **up to 20% higher**

performance compared to the 35%-45% MFU range typically observed in public foundation model training benchmarks, significantly bridging the "AI Efficiency Gap."[3] Further benchmarking against specific published results showed MFU improvements of 18–28% over published results from leading AI labs. Additionally, collaborative testing using NVIDIA DGX Cloud Benchmarking Recipes confirmed CoreWeave's NVIDIA Hopper GPU infrastructure performs **on par with the NVIDIA reference architecture**.

Achieving this level of performance was only made possible by the specific capabilities unique to CoreWeave Cloud: bare metal performance, robust health checking, automated fleet and node lifecycle management, optimized storage solutions accessed via NVIDIA BlueField DPU-managed network links on dedicated network fabric separate from the NVIDIA Quantum InfiniBand fabric used to communicate updates during training, topology-aware scheduling via CoreWeave's Slurm on Kubernetes (**SUNK**), and integrated detailed observability.

We additionally demonstrated efficiency gains from in-house implementations of modern best practices, including asynchronous checkpointing using tools like **Tensorizer**, massively parallel text processing with the high-speed gpt_bpe tokenizer, and automated recovery facilitated via SUNK.

Key Results

Table 1. Summary of CoreWeave Benchmark Results Compared to Industry Baselines

Metric	CoreWeave Result	Industry Baseline	CoreWeave Uplift
Model FLOPS Utilization (MFU)	51–52% (1024 NVIDIA H100 GPUs)	35–45%[4,5,6]	**+18–28%**
Effective Training Time Ratio (ETTR)	97.5% @ 1024 GPUs	~90% or lower[1]	**~8% gain**
Mean Time to Failure (MTTF)	3.66 days @ 1024 GPUs	~0.33 days[2] @ 1024 GPUs	**10× longer**
Checkpoint Save Time	17s (async, 1024 GPUs)	129s (synchronous baseline, Section 6.2)	**~8× faster**
Checkpoint Load Time	8.8–34.5s (Tensorizer)	25.9–68.3s (torch.distributed, Section 6.2)	**2–3× faster**

Tokenization Throughput	63M tokens/sec (gpt_bpe)	~5–10M tokens/sec (HugginFace Tokenizers)[7]	**6–12× faster**

Overview of performance and reliability metrics from large-scale AI training jobs on CoreWeave's platform, benchmarked against public results from industry leading benchmarks. CoreWeave demonstrated 18–28% higher GPU efficiency (MFU), 10× longer MTTF, and significantly faster checkpointing and tokenization—all contributing to improved cost-efficiency and time-to-market for foundation model training.

1. Introduction: The Challenge of Large-Scale AI Training

The AI industry continues its rapid trajectory towards ever-larger models, demanding unprecedented levels of computational demand and unwavering infrastructure reliability. While hardware advancements provide the necessary raw compute, continuously using thousands of GPUs for extended training runs remains a formidable challenge, pushing the boundaries of infrastructure design and operational management.

Training large language models (LLMs) at scale presents significant infrastructure challenges that extend far beyond simply acquiring sufficient GPU resources. Training models on thousands of GPUs synchronously is immensely complex, with failures significantly impacting Time-To-Market (TTM) and total cost. As we will discuss, CoreWeave's infrastructure is holistically designed to mitigate these risks.

This document concentrates on the benchmarking methodology, infrastructure advantages, and performance results; this is not a comprehensive guide to the practical steps involved in pre-training **Ixchel**, CoreWeave's Llama 3–based 30B model.

1.1. Model Flops Utilization (MFU)

Central to evaluating training efficiency is Model FLOPS Utilization (MFU), which measures the percentage of theoretical peak GPU performance achieved during model training. MFU is calculated as the ratio of observed computational throughput to theoretical hardware capacity, accounting for the actual FLOPS required by the model architecture.

For transformer models, theoretical FLOPS per token approximates 6N for the forward and backward passes, where N represents total parameters. Our 30B parameter model thus requires approximately 180 billion FLOPS per token. With NVIDIA Hopper GPUs delivering

989 TFLOPS theoretical peak for BF16 operations, 50% MFU translates to sustained throughput of ~495 TFLOPS per GPU during active training.

This metric's value lies in its hardware-agnostic nature—it excludes implementation details like activation checkpointing, data loading overhead, and communication latency, enabling fair comparison across different systems and configurations. Industry benchmarks typically report 35–45% MFU[4,5,6] for large-scale distributed training, with efficiency declining as model size and GPU count increase due to growing communication overhead.

1.1. Landscape and Industry Context

Industry publications and reputable technical reports[1] implicitly acknowledge the difficulties inherent in large-scale training, referencing significant training interruptions even within highly resourced environments. This underscores a critical need within the ecosystem: infrastructure platforms that are not just powerful but fundamentally reliable and stable when subjected to the intense, long-duration stresses of training foundation models. Simply having access to GPUs is no longer sufficient; the infrastructure supporting them must be purpose-built for resilience and performance at scale.

1.2. Technical Hurdles

Successfully training large models requires overcoming substantial technical obstacles:

- **Platform stability:** Modern distributed deep learning training techniques synchronize parallel processes across a complex array of components that must work together seamlessly at each step of the process. Job interruptions are often caused by a failure in a single component: ensuring a multi-billion parameter model can be trained in a tractable amount of time demands stable and consistent infrastructure performance. Unpredictable interruptions, hardware degradation, or network slowdowns waste valuable compute resources as well as researcher time and can make a difference when training a model over a period of days, weeks, or months.
- **Data preparation at scale:** The multi-trillion token datasets required by large models present a massive data engineering challenge. Acquiring, cleaning, formatting, and efficiently tokenizing this data is a significant undertaking, often becoming a major bottleneck requiring high-throughput storage and considerable computational resources distinct from the main training cluster.
- **Distributed system scaling:** Coordinating thousands of GPUs for distributed model training necessitates the use of extremely high-bandwidth, low-latency interconnects, such as NVIDIA Quantum-2 InfiniBand. Beyond the fabric itself, managing heat, power, and potential hardware failures across a vast fleet becomes

a critical operational burden. Furthermore, the intense, synchronous communication patterns typical of distributed training (e.g., NCCL all-reduce operations) are highly sensitive to outliers; a single slow GPU or network link can bottleneck the entire cluster. Maintaining consistent performance across every component is essential.

- **Network contention:** Large-scale training generates distinct, high-intensity network traffic patterns. One pattern involves inter-GPU communication for synchronizing model parameters (compute fabric traffic), while another involves reading datasets and writing model checkpoints (storage fabric traffic). Allowing these fundamentally different traffic types to contend for the same network resources inevitably leads to performance degradation and unpredictable bottlenecks for both processes.

2. Benchmarking Results

Our benchmarking efforts across dozens of runs over months, including the initial validation runs and subsequent focused performance comparisons, demonstrated significant efficiency gains achievable on CoreWeave Cloud.

2.1.1 High MFU Achievement

Across multiple runs, CoreWeave consistently demonstrated high hardware utilization. The initial target throughput for the Ixchel run start was estimated at approximately 450 TFLOPS per GPU, translating to a cluster-wide effective throughput nearing 910 PFLOPs.

Subsequent optimizations and measurements across various configurations, including those aligning with external published results, confirmed that the optimized CoreWeave platform consistently achieves **MFU exceeding 50% on NVIDIA H100 GPUs**. This significantly surpasses the 35%-45% MFU typically reported in public benchmarks,[4,5,6] representing up to 20% higher performance.

2.1.2 Comparative Benchmarks

To provide direct comparison points, we executed training runs aligned with published parameters from leading AI labs:

- *CoreWeave vs. Leader A:* For a 30B parameter model run aligned with hyperparameters used by Leader A (on A100s), CoreWeave achieved **51.9% MFU on 128 NVIDIA H100 GPUs**. This represents a **28% improvement** over the 40.43% MFU reported in their paper. (Note: While GPU types differ, MFU as a percentage of theoretical FLOPs allows for meaningful comparison.)[8]

- *CoreWeave vs. Leader B:* For a 30B parameter model run aligned with hyperparameters used by Leader B for their MPT-30B model (on NVIDIA H100 GPUs), CoreWeave achieved **49.2% MFU on 128 NVIDIA H100 GPUs**. This represents an **18% improvement** over the 41.85% MFU reported by Leader B for their run.[4,9]

Table 2. Comparative MFU Benchmarking Against Public Training Runs

Origin Lab	# GPUs	Seqlen	Origin MFU	CoreWeave MFU (H100-80gb)
Leader A (NVIDIA A100-80gb)[8]	128	8192	40.43%	51.9%
Leader B (NVIDIA H100-80gb)[4,9]	128	2,048	41.85%	49.2%

CoreWeave's H100 GPU MFU results compared to published MFU values from two industry experts, Leader A8 and Leader B.9 Despite architectural and hardware differences, CoreWeave consistently outperformed public baselines by 18–28%, demonstrating superior GPU efficiency and infrastructure tuning for large-model training.

2.1.3 NVIDIA DGX Cloud Benchmark Equivalence

In collaboration with NVIDIA, we utilized NVIDIA **DGX Cloud Benchmarking Recipes** to evaluate CoreWeave's platform for NVIDIA H100 GPUs across a suite of training and fine-tuning applications. Preliminary results demonstrated that CoreWeave's infrastructure achieved performance **on par with NVIDIA's reference architecture** across all tested workloads, for both BF16 and FP8 precision. This equivalence underscores the quality and optimization level of CoreWeave's environment.

2.2 Reliability Analysis

2.2.1 MTTF Analysis

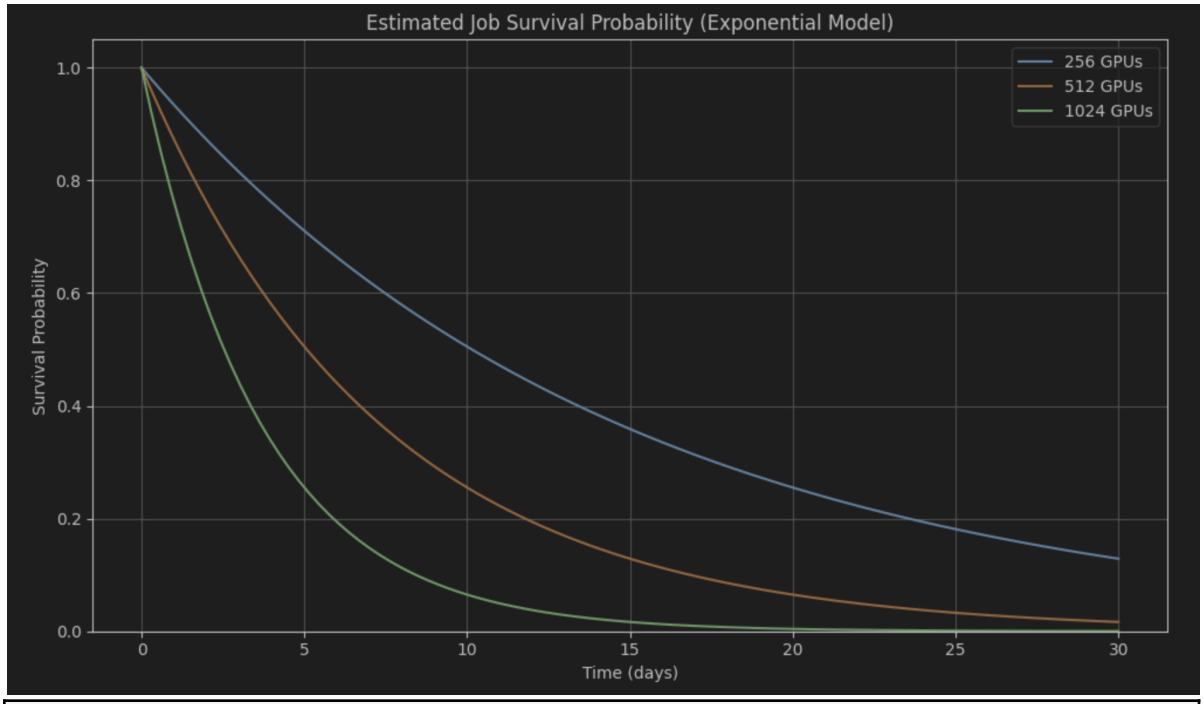
Beyond raw MFU, the focus on infrastructure stability and automated recovery directly impacts overall throughput. Features like proactive health monitoring and automated node replacement contribute to achieving high goodput rates (productive time vs. total time), **measured as high as 99%** in some contexts (1,024 or more GPUs), by minimizing disruptions. The automated job resubmission via SUNK proved effective in handling non-zero exit codes and NCCL timeouts, significantly reducing recovery time compared to manual intervention, directly benefiting ETTR and MTTF metrics.

The failure analysis process involved correlating job exit codes and SUNK logs with Weights & Biases (W&B) training metrics and Grafana infrastructure dashboards for systematic root cause identification. This allowed us to track interruptions, quantify recovery times, and calculate the overall impact on training efficiency (ETTR) and reliability (MTTF).

Conventional distributed deep learning training workloads operate in a “lock-step” paradigm, such that if one hardware component fails, it brings down the whole job. Consequently, the likelihood of job interruption within a given time window scales with the number of hardware components involved in the job.

We performed experiments on clusters ranging from 512 to 1,024 GPUs. To estimate the per-GPU failure rate, we fit a right-censored univariate exponential survival model (See appendix for details). This model allowed us to leverage information from both job failures and jobs that did not end from an unexpected interruption. For the purpose of the model, we assume interruptions are generally caused by hardware failures and that these failures are independent and identically distributed: we then normalize the data to estimate the per-GPU failure rate by multiplying job durations by number of GPUs.

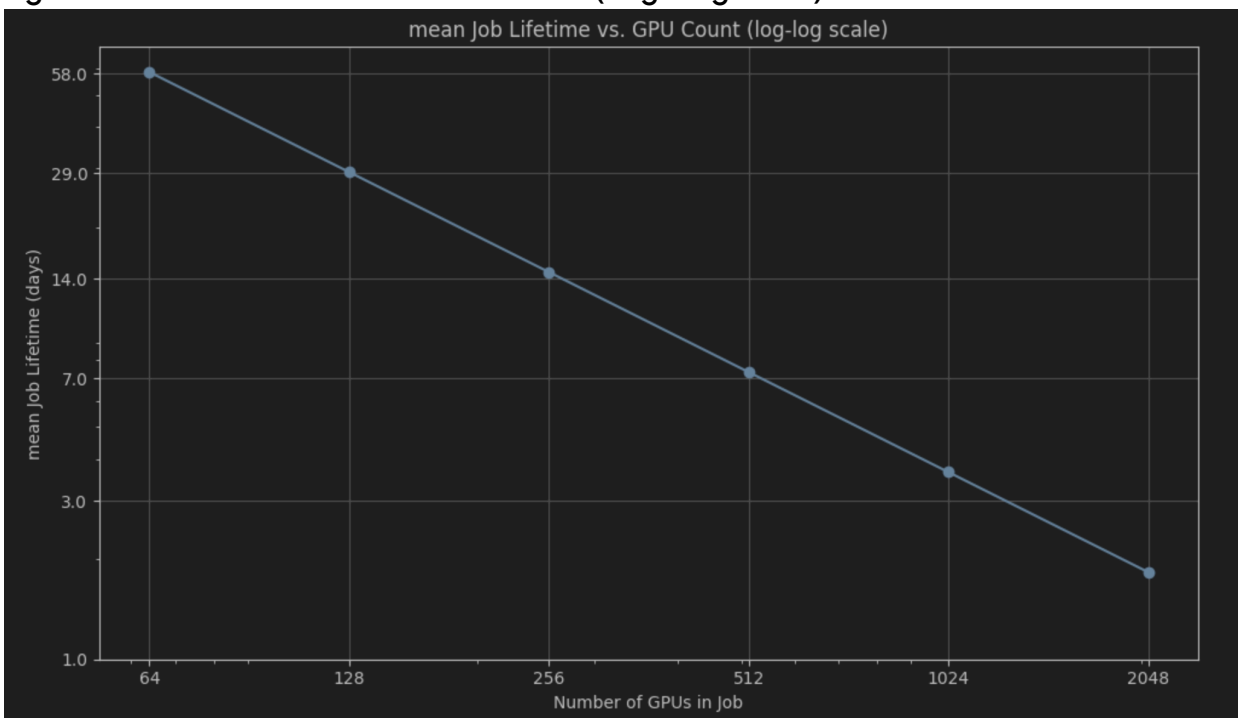
Figure 1. Estimated Job Survival Probability for Large-Scale Training Jobs



Survival curves for jobs running on 256, 512, and 1,024 GPUs, derived from an exponential failure model fitted to observed job durations and interruptions. Larger jobs experience reduced survival times due to increased component failure likelihood.

From our experiments, we estimate a per-GPU failure rate ($n\lambda$) of 3,748.25 days/failure. We then use the fitted rate to estimate the mean job lifetime—i.e. expected time to failure (MTTF=E[TTF])—for jobs of varying scales by dividing $n\lambda$ by the number of GPUs in the job by (i.e. $E[TTF] = \lambda = n\lambda / n$).

Figure 2. Mean Job Lifetime vs. GPU Count (Log-Log Scale)



Projected mean time to failure (MTTF) across varying GPU cluster sizes, based on an exponential survival model fitted to real training job data. The log-log curve highlights the nonlinear decrease in job lifetime as GPU count increases, emphasizing the operational fragility of ultra-large training jobs.

Our observed job reliability is a ~**10×** improvement relative to the rates reported in a reputable industry paper (which included interruptions from job preemption, which was not a factor for our experiment),[2] and a projected **43.7%** improvement relative to the 7.8 unplanned interruptions/day at 16K GPUs observed by the authors of a reputable industry study(419 unplanned interruptions over a 54 days window).[1]

Table 3. Projected Job Reliability at Varying GPU Scales Compared to Industry Baselines

n_gpus	n_nodes	MTTF Industry Benchmark[1,2] (days)	E[TTF] CoreWeave (days)	E[Failures per day]
1			3,748.25	0.0003
2			1,874.13	0.0005
4			937.06	0.0011
8	1	47.70	468.53	0.0021
16	2		234.27	0.0043
32	4		117.13	0.0085
64	8		58.57	0.0171
128	16		29.28	0.0341
256	32		14.64	0.0683
512	64		7.32	0.1366
1,024	128	0.33	3.66	0.2732
2,048	256		1.83	0.5464
4,096	512		0.92	1.0928
8,192	1,024		0.46	2.1856
16,384	2,048	0.075	0.23	4.3711

Mean Time to Failure (MTTF) estimates for training jobs using between 1 and 16,384 GPUs, derived from a survival model fit to CoreWeave's real training workloads on 512–1,024 GPU clusters. Results are benchmarked against published baselines

from an industry-leading reliability study2 and paper,1 showing up to 10× longer job lifetimes and a 43.7% lower failure rate at 16K GPUs.

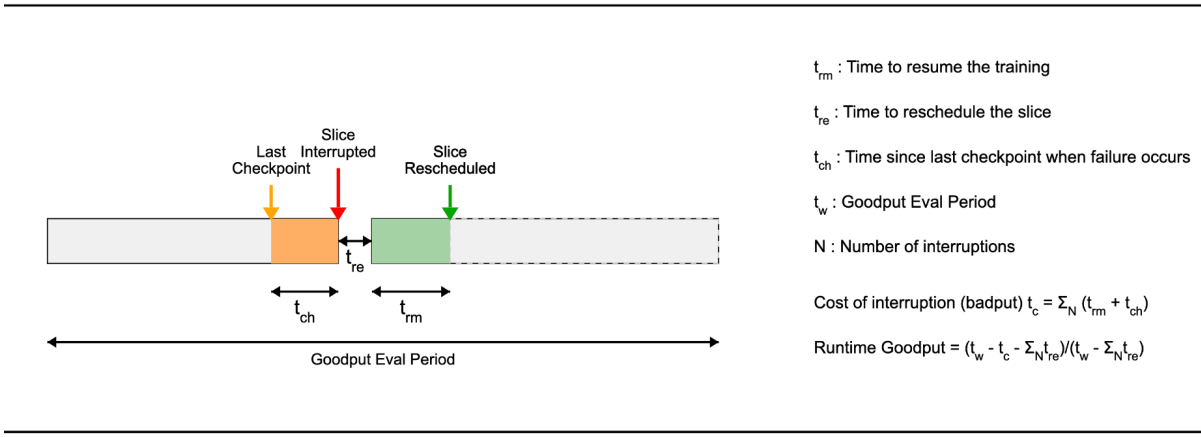
2.2.2 Goodput

A variety of software events are necessary components of model training. These include:

- Job initialization
- Loading checkpoints and initializing models
- Saving checkpoints
- Forward/backward training steps

To call the "Active Goodput" of a job is the fraction of the job duration that was comprised of training steps. The "Runtime Goodput" of a job (as previously described by Google[7]) discounts the duplicated effort from steps made after the last checkpoint, aka "badput". Assuming a failure can occur randomly anywhere in the inter-checkpoint interval, the expected badput for any job is half the inter-checkpoint interval. Our experiments generally targeted a checkpointing cadence of around 1–2 hours, i.e. risk appetite calibration considered only the wall time lost rather than optimizing checkpointing cadence specifically to maximize GPU clock time.

Figure 3. Breakdown of Job Runtime for Goodput and ETTR Calculation



Schematic illustrating how a training job's total runtime is divided into productive (goodput) and non-productive (badput)

segments. Components include active training, checkpointing, job initialization, failure recovery, and resume overhead. This breakdown supports the computation of Effective Training Time Ratio (ETTR). Based on original image from source: Google — Introducing ML Productivity Goodput.10

Table 4. Measured Runtime Goodput for Large Training Jobs

# GPUs	# Nodes	Average Runtime Goodput %
512	64	99.53%
1,024	128	99.72%

Runtime Goodput measurements for 512- and 1,024-GPU jobs, where productive time includes checkpointing and recovery overhead. Despite large scale, jobs achieved over 99.5% Runtime Goodput, highlighting the efficiency of CoreWeave's infrastructure even under generous accounting metrics. Source: Definition from Google.10

2.2.3 Goodput Ratio (ETTR)

Google's Runtime Goodput considers checkpointing time as part of the goodput period. We consider that badput, as does an industry leader's Effective Training Time Ratio (ETTR) metric—"the ratio of *productive runtime* to the *available wallclock time* of a *job run*"—which additionally takes into account re-queueing time.[1]

For the purpose of calculating ETTR, we adopt FAIR's more pessimistic approach and define goodput as the time spent on training iterations, excluding checkpointing, loading the model and optimizer state, tearing down the job after failure, etc. We then define lost training progress as the interval to the start of the resume job as reported by Slurm plus the time taken from the start of the slurm job to the first training step (to account for the overhead from restarting) plus half of the inter-checkpoint interval (to account for "wasted steps" lost when loading the resume job from the most recent available checkpoint).

We penalized this goodput metric by subtracting the lost training progress, and divided by the total slurm job duration to compute the ratio of productive training time to available wallclock time, i.e. ETTR. We computed the per-"# GPUs" ETTR by summing the respective components across jobs for a given number of GPUs, and then computing a single ETTR for that category.

Table 5. Effective Training Time Ratio (ETTR) for Large-Scale Jobs

# GPUs	# Nodes	ETTR %

512	64	98.69
1,024	128	97.48

ETTR measurements for 512– and 1,024–GPU training jobs, based on an industry leader's definition of productive time.2 This stricter metric excludes checkpointing and recovery overhead, providing a conservative estimate of usable training time. Even under this definition, jobs maintained ETTR values above 97%, highlighting the platform's ability to minimize disruption during long training runs.

3. Benchmarking Methodology

To objectively validate the benefits of CoreWeave's infrastructure and operational practices for large-scale AI training, we designed and executed a comprehensive benchmarking project. Our methodology focused on measuring key infrastructure-level performance and reliability metrics during a realistic, production-quality pre-training task.

3.1. Benchmarking Goals and Narrative

The primary goal of this benchmark was not to train a state-of-the-art model but rather to rigorously evaluate the capabilities of the underlying CoreWeave Cloud platform in supporting such demanding workloads. We focused specifically on quantifying:

- **Reliability and stability:** Measured primarily through MTTF and the effectiveness of automated recovery mechanisms.
- **Training efficiency:** Measured via the ETTR, which accounts for time lost due to interruptions, checkpointing overhead, and recovery compared to the ideal theoretical training time.
- **Hardware utilization:** Measured using MFU to understand how effectively the GPU compute resources were used during the active training phases.
- **Checkpoint performance:** Measured through both checkpoint save time (comparing synchronous vs. asynchronous methods) and checkpoint load time (comparing torch.distributed vs. Tensorizer), quantifying the overhead of model persistence operations.
- **Data pipeline efficiency:** Measured via tokenization throughput using the gpt_bpe tokenizer, demonstrating the ability to prepare trillion-token datasets without bottlenecking training.
- **Total non-training time & breakdown:** Time spent on initialization, data loading waits, checkpoint saving/loading, and idle time during interruptions.

To minimize subjectivity often associated with model selection or dataset novelty, we opted for well-understood components: a Llama-style architecture, the publicly available

Dolma dataset (augmented), and the widely used Megatron-LM training framework. This allows the results to focus squarely on the impact of the infrastructure and CoreWeave's operational best practices.

3.2 Experimental Setup

Our benchmark culminated in a production-scale pre-training run, internally codenamed **"Ixchel,"** utilizing the following configuration:

- **Model:**
  - **Parameters:** 30 Billion
    - This 30B scale was chosen as large enough to necessitate distributed training across a significant cluster, thereby exercising the infrastructure's capabilities while remaining manageable and potentially transferable to different hardware (for example, NVIDIA L40 GPUs using FP8 for inference).
  - **Precision:** BF16
  - **Key Hyperparameters:** (Llama 3 style model)
    - Sequence Length: 8,192
    - Hidden Size: 6,144
    - Number of Layers: 60
    - Number of Attention Heads: 48
    - FFN Hidden Size: 21,504
- **Dataset:** The foundation was a **3.4 trillion token dataset**, curated primarily from the public Dolma collection and augmented with additional sources such as Project Gutenberg and an internal non-public dataset. Overlapping context data augmentation was applied during dataset preparation, creating a final dataset of **9.7 trillion tokens** worth of samples to feed the model during training. The motivation underlying this augmentation was to mitigate positional biases and truncated contexts.
- **Tokenizer:** We chose the **Nerdstash v2** tokenizer[11] for our vocabulary not only because it is readily supported in our dataset pipeline (Section 2.4) but also because it enabled us to tokenize our entire pretraining dataset using only a 16-bit data type per token. This decision alone allows us to create tokenized datasets that are half the size of datasets stored in a 32-bit data type, as the vocabulary length of Nerdstash 2 does not exceed 2^{16}.
- **Framework:** A CoreWeave fork of NVIDIA's Megatron-LM framework[12] served as the foundation for training, allowing us to pair megatron's pre-existing support for modern algorithms and integrations (e.g. 3D model parallelism,[12] Flash Attention,[13] Transformer Engine[14]) with additional adaptations and optimizations implemented

by the CoreWeave team (e.g. custom data loaders, Tensorizer for checkpointing, fault tolerance).

- **Parallelism strategy:** To efficiently distribute the 30B model across nodes, we employed a hybrid parallelism strategy combining:
    - Tensor Parallelism (TP = 4) within a node
    - Pipeline Parallelism (PP = 1, effectively disabled) was found to be optimal for our setup
    - Sequence Parallelism (SP) enabled to distribute certain activations across TP ranks[6]
    - Data Parallelism (DP = 32 for the 128-GPU validation runs) across nodes
- **Parallelism optimizations:** We utilized Megatron's distributed optimizer state and overlapped communications to help increase our overall throughput by reducing the volume of network communication needed for training.
- **Compute infrastructure:** Experiments were conducted on clusters of up to 1,024 NVIDIA H100 GPUs (64 nodes) connected via NVIDIA Quantum-2 InfiniBand, with workloads orchestrated by SUNK on CoreWeave Kubernetes Service (CKS).

3.3 Instrumentation

Comprehensive monitoring was crucial for capturing the necessary data. We utilized:

- **W&B:** For real-time tracking of training-specific metrics, including loss curves, perplexity, gradient norms, timing breakdowns (forward/backward pass, optimizer steps), and calculated MFU.
- **Grafana:** For visualizing infrastructure and hardware metrics collected via node exporters and DCGM, including GPU utilization, power draw, temperature, memory usage, and network traffic (both InfiniBand and Ethernet/DPU).
- **SUNK/Slurm Logs:** For recording job start/stop times, exit codes, node allocations, and requeue events.

While alerting specifically on performance *slowdowns* (not just hard failures) was identified as valuable future work, the existing instrumentation provided deep visibility into job execution and failure modes.

4. The CoreWeave Approach: Purpose-Built for AI Scale

CoreWeave addresses these multifaceted challenges through a specialized, vertically integrated cloud platform designed explicitly for the demands of high-performance computing and large-scale AI workloads. Our approach incorporates several key

differentiators aimed at maximizing reliability, performance, and operational efficiency for our clients.

4.1. Proactive Infrastructure Health and Lifecycle Management

A cornerstone of our reliability strategy is a proactive stance on infrastructure health. We employ continuous, granular health checks that actively monitor the status of critical hardware components, including GPUs, network interfaces (both InfiniBand and Ethernet/DPU), memory modules, and system thermals. Unlike basic pass/fail checks, our monitoring aims to detect subtle degradation or anomalous behavior before it leads to outright failure. Nodes failing these rigorous checks or exhibiting concerning trends are automatically flagged and removed from the scheduling pool by our infrastructure management systems. This automated lifecycle management ensures that client workloads are dispatched only to a healthy and performant hardware fleet, significantly reducing the risk of hardware-induced job failures or difficult-to-diagnose performance issues. The positive impact of this proactive health monitoring and automated node management is quantified in our failure analysis results presented later in this paper (Section 4.4).

4.2. Expert, Integrated Support

Complementing our automated systems is CoreWeave's unique, deeply-engaged support model. We provide clients with direct access to our experienced engineering teams—the same engineers who design, build, and operate our infrastructure—often through shared Slack channels. This facilitates rapid, expert assistance for troubleshooting unexpected issues and proactively optimizing workloads for peak performance on the CoreWeave platform. Having direct lines of communication to specialists knowledgeable about the interplay between hardware, networking, storage, orchestration, and common ML frameworks functions as a powerful operational advantage. It accelerates problem resolution, fosters knowledge transfer, and enables a level of collaborative optimization far beyond the typical break-and-fix support paradigms found in other cloud environments.

4.3. Optimized and Flexible Infrastructure Components

Underpinning our operational model is an infrastructure stack where each component is selected and configured for demanding AI workloads:

- **Networking architecture:** We recognize that network performance is paramount and employ a multi-fabric design.
    - *Compute fabric:* High-bandwidth, low-latency **NVIDIA Quantum-2 InfiniBand** fabrics serve as the standard interconnect for our large GPU clusters. This fabric is reserved exclusively for the intense, latency-sensitive

inter-GPU communication characteristic of distributed training (e.g., NCCL collectives), ensuring maximum performance for synchronizing model gradients and parameters across nodes, and using **NVIDIA NVLink™** for GPUs within the same node.

- *Storage connectivity and traffic separation:* To handle storage I/O efficiently and without interfering with the compute fabric, CoreWeave utilizes **NVIDIA BlueField-3 Data Processing Units (DPUs)**. These DPUs manage connectivity to our distributed file storage systems, providing high-speed (100 Gib/s per DPU) network links specifically for storage access, complete with robust tenant isolation. This architecture deliberately segregates storage traffic onto a separate physical network path from the NVIDIA Quantum InfiniBand compute fabric. This separation is crucial as it prevents the compute fabric from becoming congested by potentially large storage operations during training (like checkpoint writes or dataset reads) and ensures storage I/O performance is not impacted by compute traffic. Eliminating this network contention is fundamental to maintaining predictable performance and maximizing GPU utilization.

- **Compute resources:** We provide **bare metal access** to underlying compute resources (CPUs and GPUs). This eliminates the performance overhead, jitter, and potential compatibility issues sometimes associated with hypervisor-based virtualization, granting users maximum performance potential and direct control over the hardware environment.
- **Storage solutions:** We offer a range of storage options designed for different performance tiers and access patterns. This includes high-performance Network File Systems (NFS) suitable for many use cases, alongside highly scalable parallel file systems like **VAST Data** for workloads demanding maximum I/O throughput and scalability. Access to these systems is provided via the dedicated BlueField DPU-managed storage network links. This flexibility allows users to select the optimal storage backend for different parts of their workflow, such as utilizing VAST's performance characteristics for highly efficient asynchronous checkpointing strategies, as explored later in this benchmark (Section 4.2). For our experiments, training code was deployed to a “home” mounted volume, and training artifacts such as checkpoints and logs were written to a larger “data” mount.
- **Job scheduling and orchestration:** CoreWeave utilizes **SUNK (Slurm on Kubernetes)**, a robust and scalable job orchestration system that integrates tightly with our infrastructure and Kubernetes control plane. SUNK enables sophisticated scheduling policies, including **topology-aware placement** (critical for minimizing communication hops on large, multi-switch InfiniBand fabrics), and provides the essential foundation for automated job lifecycle management, including the failure

detection and automated resubmission mechanisms central to our reliability strategy.

- **Serverless supercomputers via CKS:** Our compute cluster is a serverless abstraction defined via a config that requests a specific target hardware allocation. This setup mitigates hardware issues by immediately evicting and replacing in the abstract cluster. The logical serverless cluster is configured to use hardware constrained to the same data center (prioritizing the same rack and leaf group when available), ensuring that we are able to minimize the contribution of physical limitations to communication delay between GPU nodes. When nodes are idling, automated health checks are performed regularly to ensure the readiness of the cluster, and nodes failing these checks are immediately evicted and replaced via the serverless Kubernetes mechanism.

4.4. Enabling Optimized Workflows

The flexibility and raw performance inherent in CoreWeave Cloud actively enable users to implement highly optimized workflows that might be difficult or impossible in more restrictive environments. One particularly illustrative example of how CoreWeave's infrastructure provides exceptional flexibility in data handling is detailed below:

- **Data pipeline efficiency:** We leveraged CoreWeave's high-throughput storage access and readily available compute resources from outside of our main training cluster to implement a highly efficient pipeline to prepare our pretraining dataset. We have used a custom tool named gpt_bpe,[16] a highly optimized implementation of the BPE tokenization method that allows us to tokenize our pretraining dataset with the **Nerdstash v2** tokenizer. The tokenizer had achieved processing speeds of up to **63 million tokens per second**, reducing the time required to prepare our trillion-token scale dataset as much as 6× compared to HuggingFace Tokenizers.[7] Furthermore, the tokenizer readily supports applying **overlapping context data augmentation** techniques which amplifies our effective pretraining dataset size from 3.4 trillion tokens to **9.7 trillion tokens**. We have also implemented an optimized dataloader within the training framework (Megatron-LM), which allows us to use our own dataset format. This ability to customize and optimize the entire pipeline—from raw data curation, high-speed tokenization, augmentation, to efficient loading—significantly accelerated development iterations and contributed to overall training efficiency.
- **Job startup efficiency:** Jobs were executed in docker containers, utilizing NVIDIA pyxis+enroot. To accelerate job start-up time, the container image was saved locally to the data mount in compressed SquashFS format. Additionally, checkpoints were saved in the highly performant **Tensorizer** format to minimize model load time (see discussion in sec 6.2).

4.5. Integrated Observability and Configuration

Effective management and optimization of large-scale training require deep visibility. Our platform features an integrated observability stack combining infrastructure, hardware, and GPU metrics (collected via node exporters and DCGM, visualized in **Grafana**) with real-time training metrics captured directly from ML frameworks (logged to and visualized in **Weights & Biases**). This unified view allows users and CoreWeave support teams to easily correlate system behavior (e.g., network latency spikes, GPU power fluctuations, memory usage) with training performance dynamics (e.g., drops in MFU, slower gradient updates, loss curve anomalies), facilitating rapid root cause analysis and informed performance tuning. Compared to the often rigid instance types of traditional clouds, CoreWeave offers significantly greater **configurability** and **observability** across compute instances, storage volumes, and network settings, allowing resource allocation to be tailored precisely to workload needs.

4.6. The Vertical Integration Advantage

Finally, CoreWeave's ability to deliver these capabilities reliably stems from our **vertical integration**. By owning, managing, and optimizing the entire technology stack—from physical data center deployment and hardware selection, through network fabric design and configuration, operating system builds, the orchestration layer (SUNK), storage solutions, and up to the support model—we ensure all components work harmoniously. This tight integration enables us to rapidly deploy platform-wide optimizations, swiftly resolve complex cross-layer issues that might stump siloed teams elsewhere, and provide performance tuning advice uniquely tailored to our specific, high-performance environment.

5. Technical Implementation

5.1 Developer Experience

Our team used or developed a number of tools to make the development, execution, logging, and monitoring of training runs easier. The team used PyCharm as their primary IDE. Coding was generally performed locally on a given engineer or researcher's laptop, with PyCharm configured to deploy changes to their home directory on the slurm cluster via SSH. This ensured a secure, version-controlled source-of-truth while permitting frictionless development and experimentation.

Jobs were configured and submitted through the development of a dispatch script that made use of YAML files for templating sbatch, srun, and fault tolerance daemon processes,

allowing the modular configuration of compute, training, logging, checkpoint monitoring, and environment parameters that all fill substitutions in the aforementioned template scripts. Essentially, once these config files were configured, the srun, sbatch, and fault tolerance scripts were generated on the fly.

Debugging is a notable issue with Slurm, due to the inherent issues traditional debuggers (like pydevd and debugpy) have in seamlessly connecting to a debug server remotely. Work was done to modify a PyCharm remote interpreter to be a wrapper of a normal `python` binary to include, on execution, the reverse tunneling of a debug server on a compute node over to the login node. To fully enable PyCharm's debugger for Python code running on compute nodes, we configured the remote interpreter with SSH port forwarding. This setup routed the debugging connection directly back to PyCharm's local debugging client. This is far more desirable than more primitive debuggers like `pdb`, which are not thread-aware.

6. Best Practices for Large-Scale Training on CoreWeave

Based on the experiences and findings from this benchmarking project, we have distilled several best practices for maximizing efficiency and reliability when undertaking large-scale model training on CoreWeave Cloud.

6.1. Data Preparation and Loading Strategy

Efficiently handling massive datasets is paramount, as I/O and preprocessing can easily become bottlenecks that starve valuable GPU resources. Our approach involved several key steps enabled by CoreWeave Cloud:

- **Dataset curation:** The focus of this guide is technical considerations, but it merits noting that the quality of the model is strictly bounded by the quality of the data used to train it. The goal is to "model" the generating distribution of the training data: This is only achievable if the data reflects the distribution we are interested in modeling.
- **Base tokenization:** Training data processing was effected upstream of the training exercise. We were able to leverage CoreWeave's high-performance compute and storage for batch processing on infrastructure independent of the main training cluster. Our entire pretraining dataset was subjected to tokenization, data augmentation, and construction of fully prepared training samples (8192 token contexts) prior to starting the training run, rather than reading in raw text during training and performing tokenization, augmentation, and sample construction on

the fly while the GPUs are hungrily waiting to eat those tokens. This effectively amortizes compute that would otherwise slow or potentially even bottleneck the training process.

- **Optimized tokenization tooling:** Utilize high-performance tokenization tools. Our project employed a custom BPE tokenizer written in Golang called gpt_bpe,[16] which supports the Nerdstash v2 tokenizer that we have intended to target. This tool's speed (63M tokens/sec) was critical for processing the 3.4T token base dataset efficiently. Additionally, Nerdstash v2 allowed us to store our tokenized pretraining datasets in a 16–bit data format due to the limited vocabulary size being no greater than 2^{16}. Saving tokens in a 16–bit data format, rather than 32–bit, allows us to cut down our tokenized dataset's storage footprint in half.
- **Data augmentation for training:** We have applied **context overlapping** during the pre-tokenization phase to create a diverse amount of training examples from our base dataset, which effectively expands the amount of contextual information presented to the model during training from 3.4T to 9.7T tokens after the pre-tokenization phase.
- **Efficient data loading:** Standard data loading mechanisms within training frameworks may not always be optimal. Implement or adapt **custom data loaders** designed to work efficiently with the augmented data stream and storage backend (e.g., streaming effectively from NFS or VAST). Our approach involved bypassing default Megatron-LM loaders to ensure consistent, high-throughput data delivery to the GPUs with our custom dataset format with the intent of maximizing training efficiency.

6.2. Checkpointing and Model Loading Optimization

Frequent and efficient checkpointing is non-negotiable for resilience in long-running training jobs, but naive implementations can severely impact performance.

- **Asynchronous checkpointing:** Standard synchronous checkpointing (saving directly from training processes to shared storage like NFS) halts training computation during the save operation, directly reducing ETTR. Adopt **asynchronous checkpointing** strategies. CoreWeave recommends using tools such as **Tensorizer**, which was integrated into our benchmark. The optimized approach involved:
    - Training processes quickly dump their state (model weights, optimizer state) using Tensorizer.
    - Training resumes almost immediately.
    - A separate background process asynchronously writes the checkpoint files to durable, high-performance shared storage, such as the **VAST Data**

parallel filesystem, leveraging the dedicated DPU-managed storage network.

- **Performance impact:** Our benchmarks confirmed the value of this approach. Using Tensorizer for optimized asynchronous checkpointing to VAST resulted in approximately **2× faster model load times** compared to baseline methods and reduced the overhead impacting training throughput during the save process by roughly **1.5×**.
- **Frequency and verification:** Balance checkpoint frequency with performance; more frequent checkpointing improves recovery point objective but adds overhead. Implement automated verification checks on saved checkpoints to ensure integrity.

6.2.1 Basic Checkpointing

It's imperative to insert real numbers into this situation so we can develop heuristics to optimize for efficiency. For this, the most important data point to understand is how long a single checkpoint takes.

These numbers will necessarily vary for different model sizes, storage backends, and world sizes. We explored this with our 30B model (approx. 390 GiB per checkpoint) using a VAST networked filesystem as a storage backend. There are several units this can be framed in, so we choose time lost in seconds. For example, consider two identical training scenarios—one performs a checkpoint, the other does not. We measure the checkpoint overhead as the additional seconds spent compared to the scenario without checkpointing. Checkpoint overhead can also be expressed in terms of impact to goodput or MFU as well.

In our tests, the average time cost of a single checkpoint using Megatron's built-in checkpointing machinery based on the `torch.distributed` framework was 106.6 seconds across 512 workers, or 128.7 seconds across 1,024 workers.

6.2.2 Asynchronous Checkpointing

With normal checkpointing, all training must grind to a halt to save a snapshot of the current model weights and optimizer state to persistent storage before training is allowed to resume. This approach can be improved by leveraging concurrency. The application of concurrency here is not entirely trivial: We do not want weights to change partway through a checkpoint. We need an immutable snapshot of the model state from one defined point in time to work with. The solution: Rather than stop training, we can quickly take a snapshot of the weights detached from the trainer's working copy and write that snapshot to disk in a separate process or thread in the background asynchronously, allowing the trainer to

resume separately as soon as a stable copy is made. As I/O is much slower than a memory copy, this will decrease the time cost of a checkpoint substantially.

Asynchronous checkpointing is a known pattern and has an existing stock implementation in Megatron based on multi-processing.

In our tests, the average time cost of a single checkpoint using Megatron's built-in asynchronous checkpointing machinery was 112.3 seconds across 512 workers, or 134.3 seconds across 1,024 workers—awkwardly, slightly slower than a basic synchronous checkpoint. At a checkpointing cadence of one checkpoint every 30 minutes, on the lower end, this would represent 1,665.7 seconds of training time out of 1,800 seconds of wall-clock time, reducing overall performance to 92.5% of what it could be with no checkpointing costs.

There is nuance to measuring time cost for asynchronous checkpointing. Although the ideal implementation should quickly pass control back to the trainer and incur no further costs while running in the background, this is not what occurs in practice. Resource contention between the background checkpointing process and foreground trainer process may slow down training until the checkpoint is complete. Contended resources include CPU time, network bandwidth, and various other system-level resources.

Our measurement for time cost thus accounts for both time spent blocking (the synchronous time taken to copy a snapshot of the model state and launch a background checkpointing task) and the background performance hit (how many seconds slower the trainer runs while the checkpoint is ongoing in the background). The latter metric is calculated with the following logic: If the average time taken to progress N training steps with no background checkpointing activity is T1, and the time taken to progress N steps with background checkpointing activity is T2, then T2 – T1 represents the time cost of the background checkpoint, assuming N is set sufficiently high that it covers a span of time longer than the background checkpoint.

As an example, if the average time taken to progress by 10 training steps is 20 seconds with no interference, and the time taken to progress by 10 training steps with a background checkpoint ongoing for 3 of those steps is 25 seconds, then the background performance hit for this checkpoint is considered to be 5 seconds.

The average amount of blocking time taken by Megatron's built-in asynchronous checkpointing was 99.0 seconds across 512 workers and 121.4 seconds across 1,024 workers, while the average amount of time lost to the background performance hit was 13.3 seconds across 512 workers and 12.9 seconds across 1,024 workers.

6.2.3 Optimized Asynchronous Checkpointing

Using CoreWeave's Tensorizer library for serializing and deserializing weights alongside an optimized asynchronous checkpointing flow, we were able to substantially reduce the performance impact of checkpointing, allowing for greater robustness from more frequent checkpointing and more efficient training.

The first improvement was to minimize time spent directly blocking training while taking a snapshot of the model state. Taking a snapshot is essentially writing a copy of the data currently in GPU VRAM to buffers in CPU RAM; to optimize this process, we kept persistent buffers the size of the model state in CPU RAM to copy data into and reused these buffers for each checkpoint, minimizing slow RAM allocations on all but the first checkpoint. We also use pinned CPU buffers (i.e. page-locked host memory registered for device-to-host DMA transfers via `cudaHostRegister`), which allowed much faster device-to-host transfers handled entirely by the DMA engine. We then launched the background asynchronous checkpoint as a thread rather than a process, eliminating any IPC overhead in communicating data to be saved. Tensorizer is written in Python, but is designed to minimally block the GIL during serialization so that multithreading is viable.

The second improvement was to minimize the background performance hit. By default, Tensorizer makes heavy use of concurrency and will stress a system's resources to finish saving a checkpoint as fast as possible; this incurs a background performance hit as it starves the foreground training activity of resources. However, as Tensorizer's concurrency is configurable, we simply instructed it to use less parallelism and save one tensor at a time, which reversed its usual behavior and made it very resource-light at the cost of a longer background checkpoint write time. Since the training process wasn't blocked on the background checkpoint write time, it was fine for that to take longer.

With the optimized approach, checkpointing every 30 minutes (1,800 seconds) only consumed about 10.3 and 17.1 seconds per checkpoint across 512 and 1,024 workers respectively, leaving at least 1,782.9 seconds for actual training. This represents an 8× reduction in checkpointing overhead compared to the baseline method, boosting effective training performance to 99.1%.

The average amount of blocking time taken by our asynchronous checkpointing was equal to the time cost. The time cost of the background performance hit was so small it could not be discerned from random noise in iteration time.

Table 5. Checkpoint Save Time by Method and Cluster Size

Method	Checkpointing	Cluster size	Blocking Write Time	Mean Computation Time Lost Per	Effective Training @ 30m checkpoint

				Write	cadence
Tensorizer	Synchronous	512	-	106.6s	83.1%
		1,024	-	128.7s	86.0%
Megatron		512	-	110.5s	83.7%
		1,024	-	118.7s	84.8%
Tensorizer	Asynchronous	512	**10.3s**	**10.3s**	**99.1%**
		1,024	**17.1s**	**17.1s**	**99.1%**
Megatron		512	99.0s	112.3s	92.5%
		1,024	121.4s	134.3s	92.5%

Average checkpoint save times (in seconds) for synchronous and asynchronous methods across 512– and 1,024–GPU clusters. Asynchronous checkpointing—enabled via Tensorizer and background threading—reduced save times by nearly 8×, significantly improving training throughput and fault recovery efficiency. Also significantly, the mean computational impact on the duration of the training run is measured.

6.2.4 Loading

The counterpart to checkpointing performance is loading performance. Although saving a checkpoint is a more frequent occurrence than loading from a checkpoint during training, load times are still significant in resilience and error recovery, as they contribute to how quickly a training job can restart after an error.

With the stock checkpointing and loading strategy based on `torch.distributed`, the minimum time it takes to load our 30B parameter, 390 GiB model from networked storage onto 512 worker processes is 25.9 seconds with a hot cache (i.e. after loading from the same checkpoint several times in quick succession), while the slowest recorded time was 68.3 seconds with no caching (i.e. loading from a checkpoint that had never been read before).

6.2.5 Optimized Loading

In addition to providing greatly improved checkpoint save times, Tensorizer also improves load times. While our particular application of Tensorizer in the Megatron codebase doesn't take full advantage of Tensorizer's full suite of available optimizations for streaming tensors

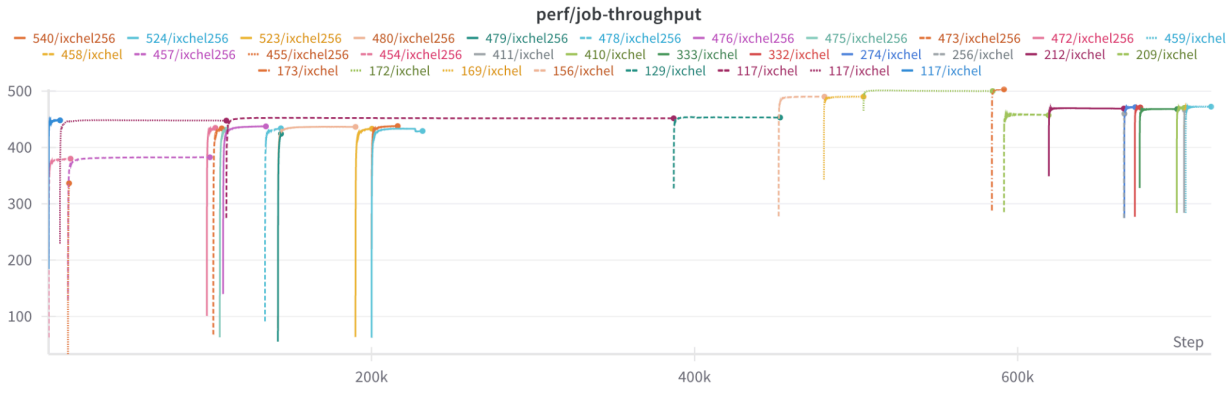
directly to the GPU, it still manages to improve the loading speed in the same testing configuration as before to 8.8 seconds with a hot cache, with a slowest recorded time of 34.5 seconds with no caching.

Tensorizer's speedup over the baseline method (Megatron-native checkpointing) ranges from 2.0×–2.9×. The fastest times for each method may be more representative of the potential of each method with a generic fast storage backend, as slowest recorded load times with no caching are highly dependent on the speed of a specific storage backend at a particular moment in time.

6.3. Automated Failure Handling and Recovery

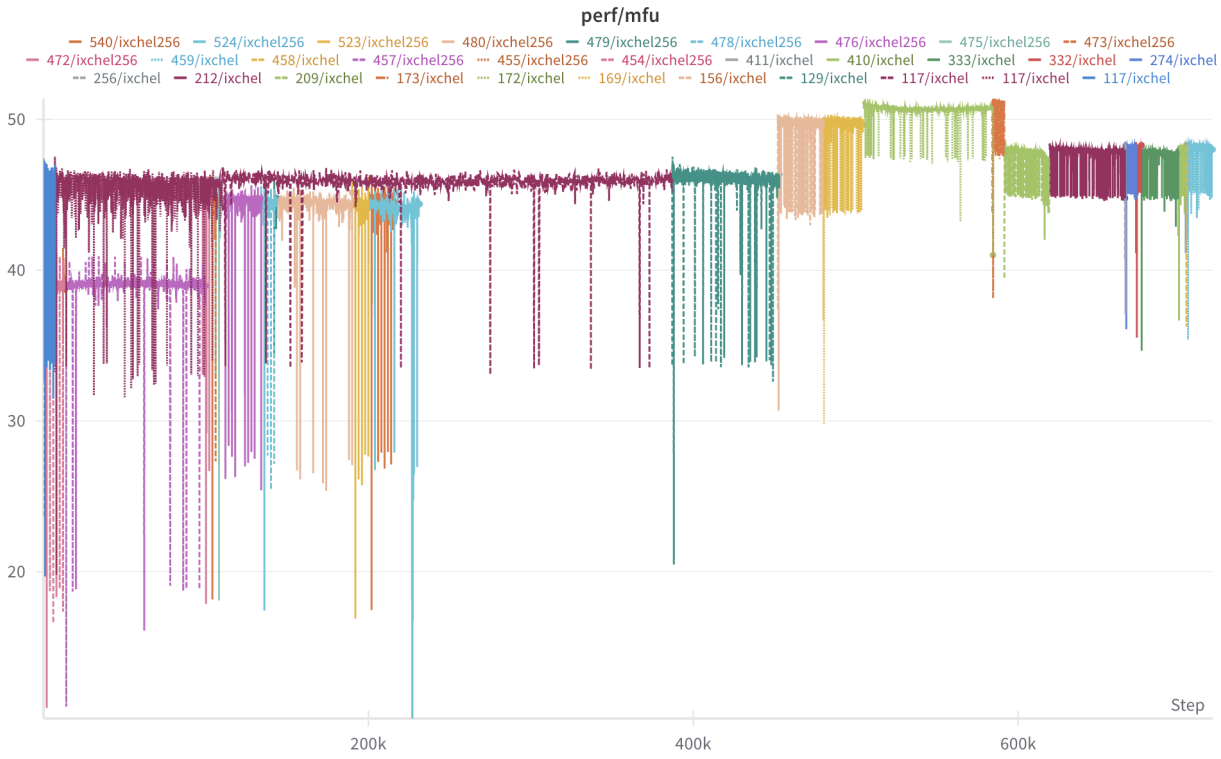
Effective failure handling and recovery minimize downtime and manual intervention by automatically detecting job failures, accurately handling termination signals, and swiftly restarting stalled or crashed processes. This section outlines essential considerations for automated recovery, specifically addressing Slurm signal handling and distributed process deadlock prevention.

Figure 4. Job Throughput Over Time with Automated Recovery



Throughput (e.g., tokens/sec or TFLOPs) plotted across training steps for multiple jobs during the Ixchel pre-training benchmark. Each line represents a unique job ID. Sharp drops indicate job interruptions, followed by automatic recovery and resumption to near-peak throughput—highlighting the effectiveness of CoreWeave's automated job resubmission and fault recovery systems.

Figure 5. MFU Across Training Jobs



MFU plotted over training steps for multiple Ixchel pre-training jobs. Most jobs maintain consistently high utilization above 45%, with brief drops corresponding to failures or job restarts. MFU rapidly recovers after each interruption, underscoring the system's ability to sustain computational efficiency even in the face of transient faults.

6.3.1. Automated Job Resubmission

Automated job resubmission involves continuously polling job states using Slurm's sacct and automatically restarting jobs upon detecting recoverable failures. Decisions to resubmit depend on the observed job states and associated exit codes:

- **No action** is required for jobs in the `RUNNING` or `PENDING` states.
- **Automatic resubmission** occurs for recoverable failure states identified by Slurm (e.g., `NODE_FAIL`, NCCL communication timeout signals).

- **Manual intervention** is required for irrecoverable failure states (e.g., `OUT_OF_MEMORY`, `FAILED` due to invalid training configurations).

Failures identified by non-zero exit codes, framework-specific error signals, or NCCL communication timeouts indicating process hangs should all be configured to trigger automated recovery. Testing has shown that automated recovery is approximately 3× faster than manual intervention, making it essential for maximizing training goodput.

To prevent excessive retries due to persistent code bugs or recurring issues, explicitly set a reasonable `MAX_RETRIES` limit within your job scheduling policies. Additionally, ensure the restart logic reliably resumes training from the latest valid checkpoint.

Our approach for this white paper uses full job resubmissions (`sbatch`) to ensure modularity and clearly defined handling of distinct failure states. However, using Slurm's native requeue functionality (`scontrol requeue`) is another viable approach, particularly advantageous when maintaining a consistent job ID across restarts is desirable for long-running production jobs. Both approaches share similar error-handling logic and checkpoint-based recovery strategies.

Observability for Root Cause Analysis

After automated recovery, leverage integrated observability tools (e.g., Weights & Biases, Grafana, Slurm logs) to rapidly diagnose root causes of both transient and persistent failures. Correlating training metrics (e.g., loss spikes, drops in Model Flops Utilization (MFU)) with infrastructure metrics (GPU ECC errors, network latency issues, node temperature warnings) greatly facilitates targeted troubleshooting.

Demonstrating Resilience

The effectiveness of automated recovery should be illustrated clearly through training progress visualizations. Graphs such as MFU or loss curves over time should demonstrate brief interruptions followed by rapid, automatic resumption.

6.3.2 Signal Handling and Deadlock Prevention in Distributed Training

Automated recovery must account for the nuances of signal handling within distributed training environments, where incorrect timing or stalled processes can result in difficult-to-recover deadlocks. Two key scenarios require careful attention:

- Slurm's signaling to the Python training processes.
- Stalled Python processes holding critical locks, such as Python's Global Interpreter Lock (GIL).

These two scenarios are interconnected because a mishandled termination signal (from Slurm) can lead directly to stalled processes, which subsequently may hold locks and prevent effective recovery.

Proper Timing of Slurm and Distributed Training Timeouts

Slurm alone does not reliably terminate distributed training jobs cleanly. Thus, explicitly coordinating timeouts between Slurm and PyTorch distributed parameters is essential.

Configure these parameters carefully to avoid deadlocks:

- Megatron's `--distributed-timeout-minutes`: wraps around torch.distributed's process group timeout, ensuring collective operations fail cleanly after a defined waiting period.
- Slurm's job timeout (`--time`): must be set longer than `--distributed-timeout-minutes` to avoid premature termination.
- Environment variable: set `TORCH_NCCL_BLOCKING_WAIT=1` to help PyTorch distributed groups handle communication stalls explicitly.

When Slurm's `--time` parameter is shorter than `--distributed-timeout-minutes`, Slurm prematurely sends a `SIGTERM` to the Python training processes. Python defers handling this signal until control returns from C++ extensions (such as LibTorch). If the signal arrives during a C++ execution, Python does not handle it immediately. As a result, the terminating process may fail to notify its distributed training group, causing other processes to indefinitely await a signal that never arrives, leading to a deadlock.

Handling Stalled Processes with Watchdog and Heartbeat Monitors

Even with proper signal handling, processes can still stall indefinitely if a Python thread holds the Global Interpreter Lock (GIL), preventing standard watchdog threads from intervening. To handle these edge cases, leverage PyTorch distributed's built-in watchdog and heartbeat mechanisms:

- **Watchdog thread**: PyTorch distributed includes a watchdog thread that attempts to terminate stalled processes after detecting communication inactivity. However, the watchdog requires acquiring the GIL to terminate Python processes. If the stalled process already holds the GIL, the watchdog thread itself may deadlock.
- **Heartbeat monitor (failsafe)**: To mitigate the watchdog's limitation, configure the heartbeat monitor as a second-level failsafe. Set environment variables:
    - `TORCH_NCCL_ENABLE_MONITORING=1`
    - `TORCH_NCCL_HEARTBEAT_TIMEOUT_SECONDS` (recommended values typically around 30–60 seconds depending on your training workload)

When enabled, this heartbeat monitor tracks the watchdog thread. If the watchdog thread itself stalls (due to GIL issues), the heartbeat monitor terminates both the watchdog and the training processes directly, ensuring a reliable escape from deadlocks.

6.3.3 Reliable Job Termination with Slurm

Even when distributed processes correctly exit due to internal timeouts or heartbeat monitors, Slurm may fail to recognize the termination and produce an appropriate job exit code. To guarantee consistent and predictable job behavior, explicitly set: `--kill-on-bad-exit=1` in your Slurm `srun` command.
This ensures Slurm terminates the entire job step if any individual task encounters a failure, thereby maintaining clear and actionable exit states.

6.4. Monitoring and Observability

Deep visibility is essential for both reactive troubleshooting and proactive optimization.

- **Correlate training and infrastructure metrics:** Actively use the combined view offered by tools like W&B and Grafana. Understanding the interplay is crucial: Did a drop in MFU correlate with increased network latency on the storage fabric during a checkpoint or perhaps with thermal throttling on a specific GPU? This correlated view accelerates debugging.
- **Monitor communication collectives:** For advanced users, monitor the performance of NCCL communication collectives (like AllReduce). Framework-level logging or specialized tools can reveal imbalances or bottlenecks within the InfiniBand compute fabric that might not be obvious from system-level metrics alone.
- **Future directions:** Explore deeper tracing capabilities (e.g., system-level tracing, PyTorch Profiler) to analyze fine-grained performance characteristics, such as dataloading bottlenecks or specific kernel execution times. Implement alerting based on performance degradation (e.g., sustained MFU below a threshold, increased communication overhead) rather than just hard failures.

7. Conclusion: Reliable, Performant AI Training at Scale with CoreWeave

Training state-of-the-art artificial intelligence models at the scale demanded by modern LLMs presents formidable challenges that transcend raw compute power. Success requires an infrastructure platform meticulously designed for stability, performance, and operational efficiency under extreme load, coupled with the flexibility to implement sophisticated optimization techniques across the entire workflow.

This white paper has detailed CoreWeave's approach to addressing these challenges, validated through rigorous benchmarking involving the pre-training of a 30-billion parameter model. Our results demonstrate that CoreWeave's specialized cloud, built upon bare metal performance, thoughtfully architected networking (including dedicated NVIDIA Quantum InfiniBand compute fabrics and NVIDIA BlueField DPU-managed storage connectivity), and robust infrastructure management, provides the necessary environment for reliable and highly performant large-scale training.

We have shown that through the implementation of operational best practices and platform-specific optimizations, significant improvements in training efficiency and reliability are consistently achievable. Key findings include:

- **Leading performance:** CoreWeave achieves **Model FLOPS Utilization (MFU) exceeding 50%** on NVIDIA H100 GPUs, a figure representing **up to 20% higher performance** than typical industry benchmarks. Direct comparisons showed MFU improvements of 18–28% over specific published results, and performance was validated to be **on par with the NVIDIA reference architecture**.
- **Infrastructure matters:** Proactive health monitoring, automated node lifecycle management, and purpose-built network architectures are critical for minimizing interruptions and maximizing uptime, contributing to **goodput rates as high as or higher than 96%**.
- **Optimized workflows drive efficiency:** The platform's flexibility enables crucial optimizations, including high-speed custom tokenization (`gpt_bpe`, Nerdstash v2), sophisticated data augmentation, and highly efficient asynchronous checkpointing (Tensorizer with VAST).
- **Automation enhances resilience:** Automated failure detection and job resubmission via SUNK significantly improve the Effective Training Time Ratio (ETTR) by reducing recovery times.
- **Observability Enables Insight:** Integrated monitoring across infrastructure (Grafana) and training frameworks (Weights & Biases) provides essential visibility for rapid root cause analysis and tuning.

These quantified advantages translate directly to tangible benefits for organizations undertaking large-scale AI initiatives: Faster time-to-market for model deployment, reduced total cost of training through minimized wasted compute, and increased predictability for complex research and development efforts. For organizations pushing the boundaries of artificial intelligence, CoreWeave offers a demonstrably superior platform, combining leading performance with the architectural foresight, operational rigor, flexibility, and expert support required to succeed at scale.

References

1. The Llama 3 Herd of Models. (2024). *arXiv preprint* arXiv:2407.21783. https://arxiv.org/abs/2407.21783

2. Revisiting Reliability in Large-Scale Machine Learning Research Clusters. (2024). *arXiv preprint* arXiv:2410.21680v1. https://arxiv.org/abs/2410.21680v1

3. CoreWeave. (2025). CoreWeave leads the charge in AI infrastructure efficiency, with up to 20% higher GPU cluster performance than alternative solutions. *CoreWeave Blog*. https://www.coreweave.com/blog/coreweave-leads-the-charge-in-ai-infrastructure-efficiency-with-up-to-20–higher-gpu-cluster-performance-than-alternative-solutions

4. Databricks. (2024). MPT-30B: Raising the bar for open-source foundation models. *Databricks Blog*. https://www.databricks.com/blog/mpt-30b

5. Rajbhandari, S., Rasley, J., Ruwase, O., & He, Y. (2022). Reducing activation recomputation in large transformer models. *arXiv preprint* arXiv:2205.05198. https://arxiv.org/pdf/2205.05198

6. Chowdhery, A., et al. (2022). PaLM: Scaling language modeling with Pathways. *arXiv preprint* arXiv:2204.02311. https://arxiv.org/pdf/2204.02311

7. Hugging Face. (n.d.). *Tokenizers* [Software]. GitHub. https://github.com/huggingface/tokenizers

8. Chen, D., et al. (2023). Efficient parallelization layouts for large-scale distributed model training. *arXiv preprint* arXiv:2311.05610. https://arxiv.org/abs/2311.05610

9. MosaicML. (n.d.). LLM Foundry benchmarking results: MPT 30B. In *LLM Foundry* (script documentation). https://github.com/mosaicml/llm-foundry/tree/main/scripts/train/benchmarking#h100-80gb-bf16–large-scale--128–gpus

10. Google Cloud. (n.d.). Introducing ML productivity goodput: A metric to measure AI system efficiency. *Google Cloud Blog*. https://cloud.google.com/blog/products/ai-machine-learning/goodput-metric-as-measure-of-ml-productivity

11. NovelAI. (n.d.). *nerdstash-tokenizer-v2* [Model]. Hugging Face. https://huggingface.co/NovelAI/nerdstash-tokenizer-v2

12. Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., & Catanzaro, B. (2019). Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint* arXiv:1909.08053. https://arxiv.org/abs/1909.08053

13. Dao, T., et al. (2023). FlashAttention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint* arXiv:2307.08691. https://arxiv.org/abs/2307.08691

14. NVIDIA. (n.d.). *Transformer Engine* [Software]. GitHub. https://github.com/NVIDIA/TransformerEngine

15. NVIDIA. (n.d.). Parallelisms. In *NVIDIA NeMo Toolkit Features*. https://docs.nvidia.com/nemo-framework/user-guide/latest/nemotoolkit/features/parallelisms.html

16. Brown, W. (n.d.). *gpt-bpe* [Software]. GitHub. https://github.com/wbrown/gpt_bpe

17. Davidson-Pilon, C. (n.d.). *Lifelines* [Software]. https://lifelines.readthedocs.io/en/latest/index.html

Appendix

A1. Exponential Survival Model for E[TTF]

Given a job with n GPUs, each GPU has independent exponential failure time with rate λ:

$$T_i \sim Exp(\lambda)$$

A job fails when any of its GPUs fail, i.e. the time to failure for a job (TTF_{job}) with n GPUs is given by the earliest failure of any of its GPUs:

$$TTF_{job} = min(T_1,..., T_n) \sim Exp(n\lambda)$$

The log likelihood of a failure event f occurring at time t is given by:

$$log\ f(t) = log(n\lambda) - n\lambda t$$

And survival event $\mathbf{S}$ at time t given by:

$$log\ S(t) = - n\lambda t$$

Giving the full model:

$$logL(\lambda) = \sum[d_i\ log(n_i\ \lambda) - n_i\ \lambda\ t_i]$$

Where d_i is the "censoring" indicator variable, which takes the value 1 if a failure was observed and 0 otherwise.

The model was fit via the `lifelines` library.[17]

```
import pandas as pd
import matplotlib.pyplot as plt
from io import StringIO



# Input data
df = pd.read_csv("data.csv")
df.rename(columns={
   "ended with unplanned interruption": "event",
```

```
   "Number of GPUs": "n",
   "total job (active) duration (sec)": "duration"
}, inplace=True)

# seconds -> days
df["duration_days"] = df["duration"] / 86400

# Fit using duration * n to model per-GPU failure
df["duration_scaled"] = df["duration_days"] * df["n"]
ef = ExponentialFitter()
ef.fit(df["duration_scaled"], event_observed=df["event"])

t = np.linspace(0, 30, 100)  # time in days

recs = []
plt.figure(figsize=(10, 6))
for n in [256, 512, 1024]:
   n_lambda_hat = ef.lambda_ # fitted value is nλ
   inv_lambda_hat = n / n_lambda_hat # 1/λ for given n
   survival_prob = np.exp(- t * inv_lambda_hat )
   recs.append({'n': n, 'survival_prob': survival_prob, 't':t})
   plt.plot(t, survival_prob, label=f"{n} GPUs")
```

An equivalent fit via Maximum Likelihood Estimation for the parameter λ using `scipy`:

```
# Univariate Exponential Survival Model, MLE

import pandas as pd
import numpy as np
from scipy.optimize import minimize

df = pd.read_csv("data.csv")
df["n"] = df["Number of GPUs"].astype(int)
df["t"] = pd.to_numeric(df['total_duration'])
df["event"] = (df["num_interruptions"]
                                                .fillna(0)
                                                .astype(int)
```

```
                                                .apply(lambda x: 1 if x > 0 else
0))

def neg_log_likelihood(log_lambda, n, t, event):
    lambda_ = np.exp(log_lambda)
    logL = event * (np.log(n * lambda_)) - n * lambda_ * t
    return -np.sum(logL)

res = minimize(
    neg_log_likelihood,
    x0=np.log(1e-8),
    args=(df["n"].values, df["t"].values, df["event"].values),
    method='L-BFGS-B'
)

lambda_hat = np.exp(res.x[0])
```

A.2 Reference Implementation of re-queueing script

See sec. 6.3.1

```
#!/usr/bin/env bash

MAX_RETRIES="{{ cfg.misc.MAX_RETRIES }}"
[ "$MAX_RETRIES" -ge 0 ] || MAX_RETRIES=0

RETRIES_FILE="{{ cfg.misc.CHECKPOINT_DIR }}/retries_$(date +"%T" | sed 's/:/_/g').txt"
SBATCH_FILE="{{ sbatch_file }}"

mkdir -p {{ cfg.misc.CHECKPOINT_DIR }}


# Initialize retries from the file or set to 0 if the file does not exist
if [ -f "$RETRIES_FILE" ]; then
  retries="$(cat "$RETRIES_FILE")"
  [ "$retries" -ge 0 ] || retries=0
else
  retries=0
fi

LOG_RETRIES() {
  echo "${1:?}" > "$RETRIES_FILE"
}

PARSE_JOB_ID() {
  # Match against the expected output format of sbatch or print "ERROR"
```

```
  head -1 \
  | sed -E 's/^Submitted batch job ([[:digit:]]+)$/\1/; t; Q1' \
  | grep '[[:digit:]]' \
  || echo 'ERROR'
}

(
  set -o pipefail
  exec 3>&1
  TEE_STDOUT() {
    # Force tee to use the same open file description
    # Using /dev/fd/3 opens a new one when stdout is a file
    tee >(cat >&3)
  }

  while [ "$retries" -le "$MAX_RETRIES" ]; do

    if [ "$retries" -ne 0 ]; then
      printf 'Job failed; restarting. %d attempts left.\n' "$(( MAX_RETRIES - retries
))"
    fi
    LOG_RETRIES "$((++retries))"

    # Submit the job and output the job ID
    # "sbatch --wait" will match the exit code of the batch job itself
    JOB_ID="$(sbatch --wait -- "$SBATCH_FILE" | PARSE_JOB_ID | TEE_STDOUT)" && {
      printf 'Job %d finished\n' "$JOB_ID"
      break
    }

    if [ "$JOB_ID" = 'ERROR' ]; then
      break
    fi

    JOB_STATE="$(sacct -j "$JOB_ID" --format=State --noheader -XP)" || {
      printf 'Failed to query state of job %s\n' "$JOB_ID"
      break
    }
    # Exhaustive listing of job states: https://slurm.schedmd.com/job_state_codes.html
    # There are 12 to account for. Other than the ones designated for retrying,
    # their handling differs exclusively in diagnostics printed before exiting.
    # The output of "sacct -P" will sometimes include extra information,
    # like "CANCELLED by 1234," so these case statements all match by prefix.
    case "$JOB_STATE" in
      FAILED*|NODE_FAIL*)
        false ;;  # Retry on these states
      COMPLETED*)
```

```
        # This shouldn't be possible since the loop should exit
        # earlier than this on success, but handle it just in case
        printf 'Job %d finished\n' "$JOB_ID"
        break ;;
      TIMEOUT*)
        printf 'Job %d stopped (%s); not retrying\n' "$JOB_ID" "$JOB_STATE"
        break ;;
      CANCELLED*)
        # If the job was intentionally cancelled, don't restart
        printf 'Job %d %s\n' "$JOB_ID" "$JOB_STATE"
        break ;;
      BOOT_FAIL*|DEADLINE*|OUT_OF_MEMORY*)
        printf 'Job %d stopped in an unrecoverable state (%s); cancelling retry
logic\n' \
          "$JOB_ID" "$JOB_STATE"
        break ;;
      PREEMPTED*|SUSPENDED*)
        # Proper handling is unclear in this scenario, but assuming this is because of
        # limited cluster resources, it would be a bad idea to muscle in another job
        # in response to these events, so exit instead
        printf 'Job %d %s; cancelling retry logic\n' "$JOB_ID" "$JOB_STATE"
        break ;;
      # Cases after this point indicate something weird going on with this script
itself
      PENDING*|RUNNING*)
        # Something is wrong with the timing of "sbatch --wait" exiting, or we got the
wrong job ID somehow
        printf 'Error: Job %d was reported as having finished, but is %s; cancelling
retry logic\n' \
          "$JOB_ID" "$JOB_STATE" >&2
        break ;;
      '')
        printf 'Error: Could not retrieve job state\n' >&2
        break ;;
      *)
        printf 'Error: Unrecognized job state: "%s"\n' "$JOB_STATE" >&2
        break ;;
    esac

  done || {
    echo "Job failed after exceeding $MAX_RETRIES attempts. Exiting."
    exit 1
  }
)
```