CLOCKWORK.IO

Fault Tolerance Benchmark: Clockwork TorchPass, TorchFT and checkpoint restart

Keeping Distributed Training Running Through Failures

tl;dr: we randomly injected failures during TorchTitan Llama-4 MoE Scout (109B) training to evaluate the fault tolerance performance of standard checkpoint restarts, Clockwork TorchPass and Meta's TorchFT. TorchPass reached 3,000 steps the fastest at 405 minutes, checkpoint restarts took 818 minutes (2X slower) and TorchFT 930 minutes (2.3X slower).

Introduction

In standard distributed training, checkpoints are periodically written to persistent storage and used as the primary recovery mechanism when failures occur. The downside is that all progress between the failure and the last checkpoint is lost and must be re-executed. Additional GPU time is also wasted while failed resources are replaced, checkpoints are restored, and the jobs are restarted.

Clockwork TorchPass and Meta's TorchFT eliminate this overhead using two different approaches:

TorchPass performs a live GPU migration to spare resources upon failure, allowing training to continue from the same step (see TorchPass blog). TorchFT uses replica groups as the failure domain and upon failure, removes an entire replica group allowing training to continue without that replica group until it has been replaced (samples assigned to that replica group during failure are never processed).

Testing Summary

We performed four TorchTitan Llama-4 MoE Scout (109B) training runs on a 64 x Nvidia H200 GPUs (8 nodes x 8 GPUs) cluster as follows:

- **Baseline**: training run with checkpoint frequency every 100 steps and no injected failures.

- **Checkpoint Restart:** training run with checkpoint frequency every 100 steps and injected failures every 2700-4500s (failure rate chosen to approximate that of a 32,000 GPU cluster). Upon failure detection, an automated checkpoint restore and job restart was performed.
- **TorchFT:** training run with checkpoint frequency every 100 steps and injected failures every 2700-4500s. Upon failure TorchFT removed the affected replica group and continued in a degraded training state at the same step until the replica groupwas repaired.
- **TorchPass**: training run with checkpoint frequency every 100 steps and injected failures every 2700-4500s. Upon failure detection, a TorchPass live GPU migration was performed after which training continued at the same step

The results are shown in the chart. Across the 3,000 step training run:

- TorchPass completed the fastest in 405 minutes, checkpoint restarts took 818 minutes (2X slower) and TorchFT 930 minutes (2.3X slower).
- TorchPass and TorchFT completed with zero lost steps (steps that required recomputation) vs 869 lost steps for checkpoint restart.

Methodology

Test Definitions and Failure Injection

The goal of the testing was to compare the effectiveness of TorchPass Live GPU migration vs standard checkpoint restart to recover from training failures. In order to do this, we ran three tests sequentially as described below.

Baseline Testing Without Failure Injection

Test 1 - Baseline

This test involved a standard training run with no injected failures. For consistency with the other tests, asynchronous checkpoints were taken (although not used) every 100 steps. The purpose of the test was to establish baseline values for throughput TPS (tokens per second) and training time.

Fault Tolerance Testing With Failure Injection

During tests 2-4, faults were injected at random intervals every 2700-4500s (normal distribution). For consistency, the same random number generator seed was used across the tests, so that the faults occurred at the same times. The total number of failures was capped at 12. The failure rate was chosen to approximate that of a 32,000 GPU cluster.

Test 2 - Checkpoint Restart With Injected Failures

The checkpoint restart test followed industry-standard best practices for failure recovery: taking periodic checkpoints during training (every 100 steps) and using them for recovery after failures. Failures were injected randomly as described above.

Upon failure, the following occurred:

1. Worker Pod dies after ~30 seconds
2. Surviving ranks hang at NCCL collective and later receive a NCCL timeout
3. All pods crash and are recreated by PyTorchJob controller (Kubeflow v1)
4. Checkpoint is restored from most recent successful periodic checkpoint
5. Training is restarted at point of last periodic checkpoint

Test 3 – TorchPass Live GPU Migration With Injected Failures

The TorchPass test was similar to the baseline test, including the taking of periodic asynchronous checkpoints every 100 steps, except that a mutating webhook was used to inject a lightweight TorchPass component into each worker pod. Failures were injected randomly as described above.

Upon failure, the following occurred:

1. TorchPass intercepts the failure, quiesces training and initiates a graceful migration of training workload from failed worker to a new joiner pod
2. State is transferred to the joiner pod
3. Training restarts using the new joiner pod with zero lost steps

Note that if the migration fails for any reason, TorchPass reverts to using standard checkpoint-based recovery (this did not occur during the test run).

Test 4 – TorchFT With Injected Failures

The TorchFT test was also similar to the baseline test, including the taking of periodic asynchronous checkpoints every 100 steps, except the torchtitan out-of-the-box fault tolerant training loop was used. Failures were injected randomly as described above.

Important: During an initial run, the test failed to complete successfully due to two torchtitan integration bugs. The first required setting an explicit timeout passthrough to the FTManager (this was undocumented and required source code inspection to track down). The second required synchronous fault tolerant checkpoint staging to prevent save_future deadlocks (this checkpoint lock contention only manifested when TorchFT's per-step staging coincided with Torchtitan's periodic asynchronous checkpoint).

Integration Fixes Required

Issue	Symptom	Fix Applied
Undocumented timeout defaults	ft.Manager() defaulted to 60s for all timeouts; HTTP checkpoint transfer (1.4 TB state) always timed out	Patched *FTManager.__init__* to pass explicit timeout, *quorum_timeout*, *connect_timeout* from config
Shared save_future deadlock	FT per-step staging blocked 500+ seconds waiting for the periodic async checkpoint to finish writing to GCS fuse	Changed *_ft_save()* to use *AsyncMode.DISABLED*, making FT staging synchronous (~1.1s) instead of queuing behind the persistent checkpoint

Upon failure, the following occurred:

1. The failure is detected and the TorchFT Lighthouse recomputes a quorum without the impacted replica group
2. The healthy replica groups reconfigure and training continues with a reduced set of replica groups (samples assigned to the missing replica group are effectively discarded).
3. Since there is no actual failure, the missing replica group reconnects to the Lighthouse in order to join the quorum
4. A live recovery from a healthy peer is performed and after healing, the replica group is synchronized and starts participating again
5. Training continues using all replica groups

Hardware, Software and Training Configuration

All testing used the same environment and training configuration as described in the tables below:

Hardware Stack

Component	Details
GPUs	64 x NVIDIA H200
Nodes	8 nodes x 8 GPUs
Network	RoCEv2 MRDMA (GCP multi-NIC GPUDirect)
Interconnect	NVLink
Cloud	GCP using GKE

Software Stack

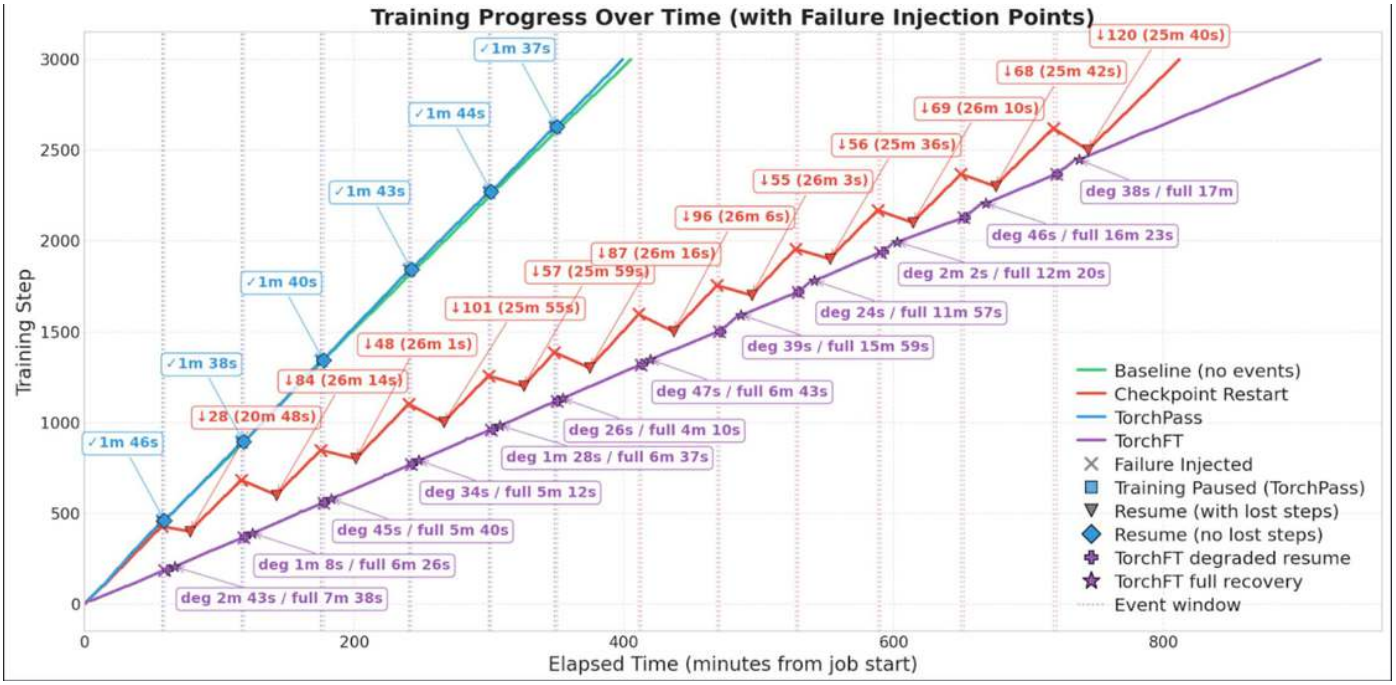
Component	Version
PyTorch	2.10.0 (stable)
TorchTitan	0.2.1
CUDA	12.9
NCCL	2.x
TorchFT	torchft-nightly=2025.12.26
TorchPass	0.2.1
Kubernetes	GKE v1.35
Training Operator	Kubeflow PyTorchJob

Training Configuration

Attribute	Details
Model	TorchTitan Llama-4 MoE Scout (109B)
Parallelism – Baseline, Checkpoint Restart, TorchPass	PP=4, DP=2, FSDP=4, TP=2, EP=4, ETP=2
Parallelism – TorchFT	PP=4, FSDP=4, TP=2, EP=4, ETP=2 per RG TorchFT replaces data_parallel_replicate_degree=2 with a TorchFT replicate group of 2
Training steps	3,000
Batch Size	local_batch_size=8, seq_len=8192
NCCL Timeout	init=300s, train=300s
Checkpoint Interval	Every 100 steps (asynchronous)
Optimizer	AdamW, lr=4e-3

Results

All four test variants (Baseline, Checkpoint Restart, TorchPass and TorchFT) successfully completed 3,000 training steps. The following chart provides an overall summary of training progress for the three tests:



Notes on the chart labels used above:

- **On the TorchPass plot line**: the blue labels with a "✓" represent the amount of time taken by the live GPU migration.
- **On the checkpoint plot line:** the red labels with " ↓ " indicate the number of lost steps and the numbers in parentheses represent the amount of idle time while restoring the checkpoint and restarting the training.
- **On the TorchFT plot line**: the purple "deg" indicates the time TorchFT took before it resumed the training in a degraded state (without the impacted replica group), "full" indicates how long training continued in a degraded state until a full recovery was completed.

Total Wall Clock Run Time and Effective Steps per Minute

As shown in the table below, the baseline training completed 3,000 steps in 411.3 minutes. TorchPass was 6.7 mins faster, checkpoint restart was 817.5 minutes slower and TorchFT was 903 minutes slower.

TorchPass achieved the highest number of effective training steps per minute at 7.4 (compared to 7.3 for baseline, 3.7 for checkpoint restart and 3.2 for TorchFT). Effective training steps per minute takes into account the impact of overheads including step recomputation, migration and checkpoint restore.

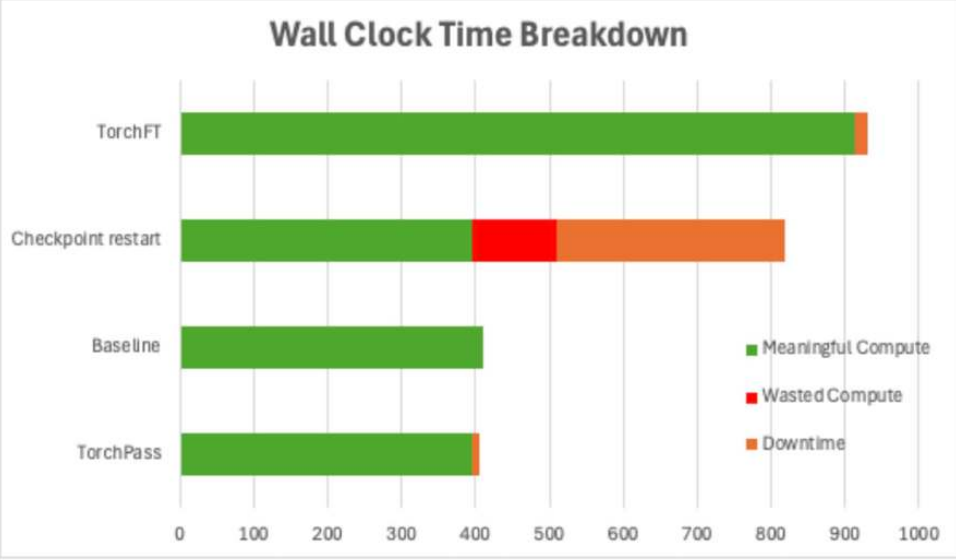
Test	Total Run Time (Wall Clock)	Time vs Baseline	Effective Steps/Min
Baseline	411.3 min	-	7.3
TorchPass	404.6 min	6.7 min faster	7.4
Checkpoint Restart	817.5 min	406.2 min slower	3.7
TorchFT	930.0 min	518.70 min slower	3.2

Note: The TorchPass improvement over the baseline is largely attributed to run-to-run variance in cluster performance in the cloud.

Wall Clock Time Breakdown

The following chart and table further breaks down the wall clock into three buckets:

- **Meaningful Compute** – this is the time spent performing forward training operations. Note that this metric does not speak to the efficiency of the compute (that is better represented by something similar to the Effective Steps/Min metric used above).
- **Wasted Compute** – this is the time spent on compute work that is lost and must be rerun. This only applies to the checkpoint restart test, where each time a failure occurs, the training must be restarted from the point of the last successful checkpoint.
- **Downtime** – this is the time where no forward training operations are performed. In the case of TorchPass it is due to the time taken by the live GPU migrations, for checkpoint restart it is the time taken restoring the checkpoint and restarting training and for TorchFt it is the time taken between the failure and when the training resumes in a degraded state.



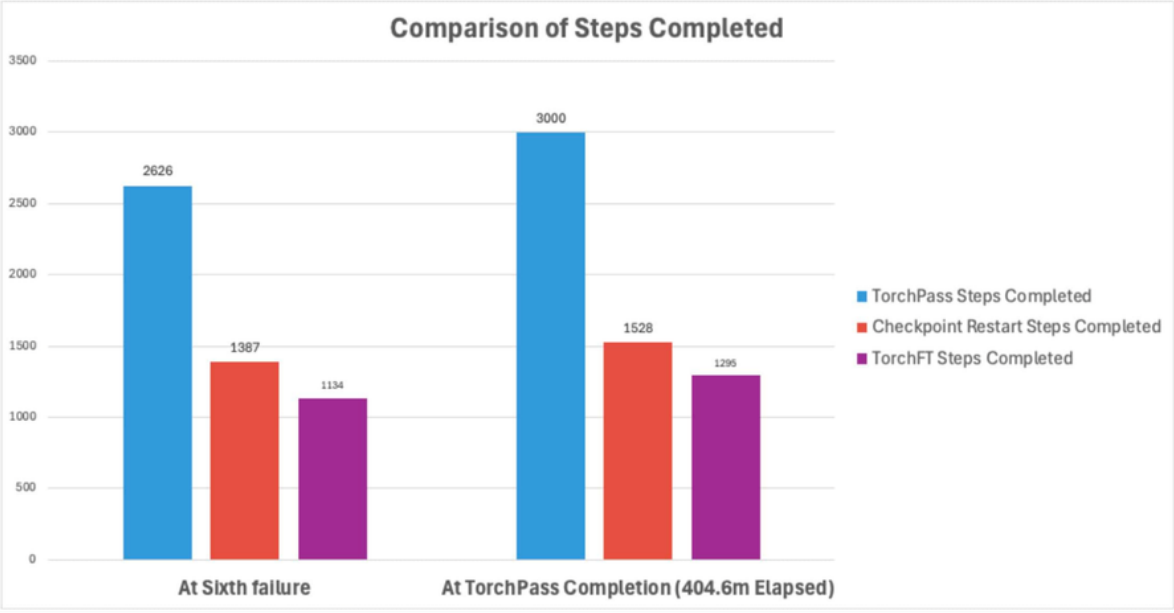
Wall Clock Time Breakdown

Test	Total Run Time (Wall Clock)	Meaningful Compute	Wasted Compute	Downtime
TorchPass	404.6 min	394	0	10
Baseline	411.3 min	411	0	0
Checkpoint Restart	817.5 min	396	114	307
TorchFT	930.0 min	913	0	17

Apples-to-Apples Failure Point Comparison at the 6th Failure

As described above, faults were injected every 2700-4500 seconds at random intervals (the same generator seed was used so that faults were injected at the same times across tests). The total number of failures was capped at 12 for all tests. This resulted in 12 faults being injected during the checkpoint restart and TorchFT tests, and 6 faults during the TorchPass test (since the TorchPass test completed in approximately half the time).

In order to provide an apples-to-apples comparison, the chart and tables below show a comparison of training steps completed at the sixth failure (just prior to TorchPass completion) and at the time that TorchPass completed the test.



At the time of the sixth failure event:

Test	Steps Completed	vs TorchPass
TorchPass	2,626	-
Checkpoint Restart	1,387	1,239 steps behind
TorchFT	1,134	1,492 steps behind

At the time of TorchPass test completion:

Test	Steps Completed	vs TorchPass
TorchPass	3,000	-
Checkpoint Restart	1,528	1,472 steps behind
TorchFT	1,295	1,705 steps behind

Training Steps Lost

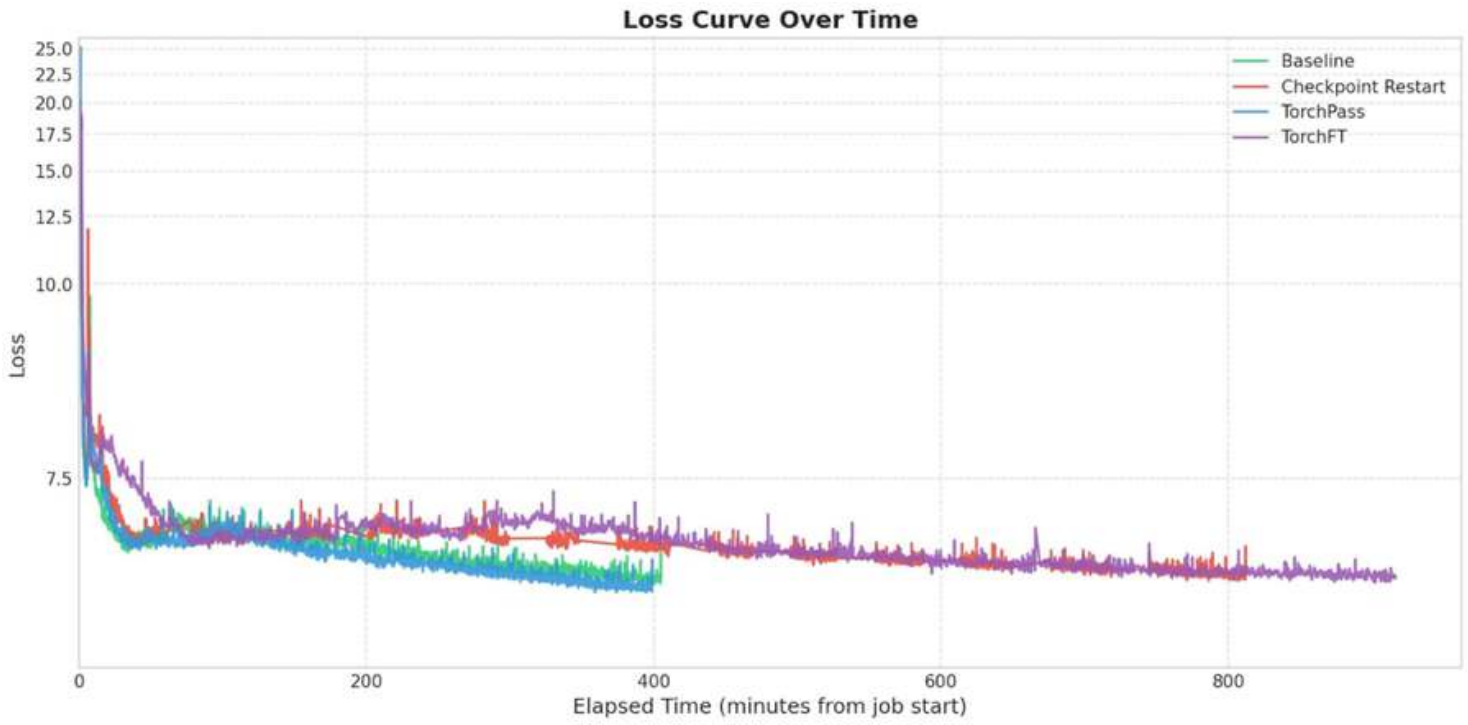
A total of 869 steps were lost and needed to be recomputed during the checkpoint restart test as shown in the chart below. The variance for each failure is due to the random interval timing of the failure injections.

No training steps were lost for the TorchPass and TorchFT tests.

Training Efficacy

All four tests successfully completed 3,000 steps and achieved final loss within expected run-to-run variance (variances attributed to stochastic noise including non-deterministic CUDA operations: atomicAdd in reductions and all-reduce ordering).

Training Efficacy



Loss at Step 3,000:

Test	Final Loss at Step 3,000
Baseline	6.248
Checkpoint Restart	6.296
TorchPass	6.150
TorchFT	6.238

Explanation of Performance

TorchPass achieved performance close to the baseline because failures are handled through live GPU migration rather than changes to the training semantics. When a worker fails, TorchPass transfers the training state to a replacement GPU and resumes the job at the same step with the same parallelism configuration. As a result, the only overhead introduced is the short migration window (about 10 minutes total across the six failures).

TorchFT introduces additional per-step overhead and communication inefficiencies even under normal training conditions (i.e., when no failures occur). Its architecture requires coordination through a central Lighthouse and per-group Managers that exchange heartbeats and recompute quorum membership, creating an additional synchronization point at each training step. More significantly, TorchFT relies on the Gloo collective communication library for cross-replica gradient all-reduce operations so that replica groups can be safely reconfigured after failures. Unlike NCCL, which uses GPU-direct networking and high-bandwidth interconnects, Gloo is CPU-based and communicates over TCP/IP sockets, requiring gradients to be copied from GPU memory to CPU memory and back each iteration. This introduces significant overhead, particularly for large models. This introduces significant overhead, particularly for large models. This is explained in more detail here.

Checkpoint restart achieves high training loop performance when running normally, but every failure is expensive: the computational work performed since the last checkpoint is lost entirely and must be recomputed. Additionally, time is lost bringing up the new resources, loading the checkpoint and restarting the training loop.

Conclusion

During random failure injection testing during Llama-4 MoE Scout (109B) training, we compared three different approaches to failure recovery: Clockwork TorchPass, standard checkpoint restart recovery and TorchFT.
All tests completed successfully after 3,000 steps with good training efficacy. TorchPass was the fastest at 404.6 min, checkpoint restarts took 817.5 minutes (2X slower) and TorchFT 930 minutes (2.3X slower).
If you're interested in testing TorchPass, please contact Clockwork at https://clockwork.io/contact/.

Industry and Customer Voices

"In our testing, Clockwork.io TorchPass delivered the fastest and most efficient fault-tolerant performance for a gpt-oss-120B training run. We used TorchTitan on a Kubernetes cluster with 64x H200 GPUs. During our testing we measured job completion time (JCT) and Model FLOPs Utilization (MFU) against a standard approach (checkpoint-restart) and the leading open-source fault-tolerant training framework (TorchFT). We simulated multiple hardware failures on the cluster in order to stress test the fault-tolerant training frameworks.

When compared to checkpoint-restart, TorchPass was significantly faster to recover from failures. This reduced overall JCT and maintained high MFU. And when compared to TorchFT, TorchPass had a significantly higher MFU. This reduced overall JCT while also maintaining an equal time to recover from failures.

Using TorchPass also has a downstream effect where it provides users with an opportunity to reduce or even remove checkpointing from their training code. This means larger effective batch sizes, lower risk of out of memory errors (OOMs), and less time spent thinking about storage. For a research organization, this can ultimately mean a faster time to reach their training objective," concluded Nanos.

Jordan Nanos, Member of Technical Staff and lead author of ClusterMAX—SemiAnalysis

---

"As Blackwell clusters roll out with an NVL72 domain, and we look to the future with Rubin Ultra's NVL576 domain, the idea that a single GPU error or network link flap can take down an entire run is totally unacceptable," said Patel. "TorchPass solves a huge challenge with cluster reliability: it provides transparent failover and live workload migration that keeps MFU high, which in turn drives better GPU economics."

Dylan Patel, Founder and CEO, SemiAnalysis

"Managing compute output across large-scale GPU clusters is vital to ensuring we're delivering reliable capacity to our customers. By using TorchPass we have the support of a company that focuses on resilience like it is a core business function: it replaces any specific failing GPU and keeps the rest of the job moving, rather than making one small problem impact our large-scale operations. In our evaluation, Live GPU Migration preserved both run continuity and throughput under real fault conditions, which is exactly what you need to deliver predictable time-to-train and a better customer experience at scale."

David Power, CTO of Nscale