

Common Approaches to Fault Tolerant AI Training & Selection Framework

May 26, 2026 · clockwork.io/blog/common-approaches-to-fault-tolerant-ai-training-selection-framework/

When a GPU falls off the bus in a large distributed training run, it is catastrophic. The failed GPU is not the expensive part. The expensive part is the blast radius: hundreds or thousands of healthy GPUs waiting, timing out, restarting, and rolling back because one participant in a tightly synchronized job can no longer participate.

We recently presented on this in a Linux Foundation webinar with [Jordan Nanos](#) from [SemiAnalysis](#). You can watch the full webinar here: [Handling Failures During Training: A Comparative Analysis of Fault Tolerant Training Frameworks](#).

Training clusters break, and the bigger they get, the more they break

We've talked to a variety of Neoclouds and end-user customers and two data points stand out:

- One top 10 software company told us that ~20% of their GPU nodes are non-operational at any given time, split between active failures and provisioning workflows for fresh nodes.
- On modern GB200 and GB300 systems, standard practice is to reserve 8 out of 72 GPUs (~11% which adds up to several hundred-thousand dollars) per rack for warm spares. Failures are common enough that holding this spare capacity has become necessary.

These aren't bad decisions by inexperienced operators. This is what running a large training cluster in production actually looks like today.

The clearest published data on this comes from [Meta's Llama 3 paper](#): **419 unplanned interruptions across a 54-day run on 16,000 GPUs**. That's one failure every ~3 hours, totaling just over 2 million lost GPU-hours – the equivalent of ~1,600 GPUs running flat-out for the entire 54 days, just to absorb failure overhead.

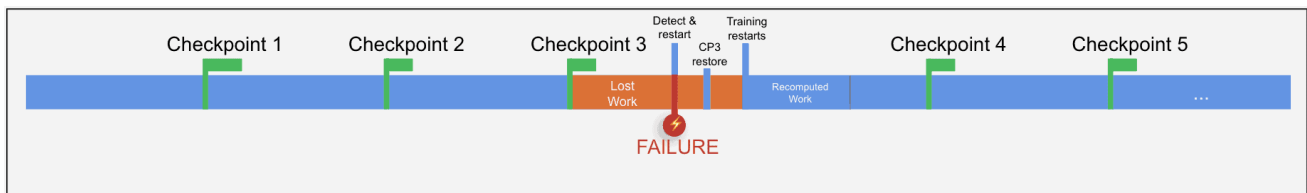
And it gets worse as clusters scale. From [Revisiting Reliability in Large-Scale ML Research Clusters](#):

- 7.9 hours Mean-Time-To-Failure (MTTF) at 1,024 GPUs
- 1.8 hours MTTF at 16,384 GPUs
- ~14 minutes MTTF at 131,072 GPUs

Three Approaches to AI Training Fault Tolerance

During the webinar, we covered three very different approaches to reducing the cost of these failures. Here's a quick summary of each.

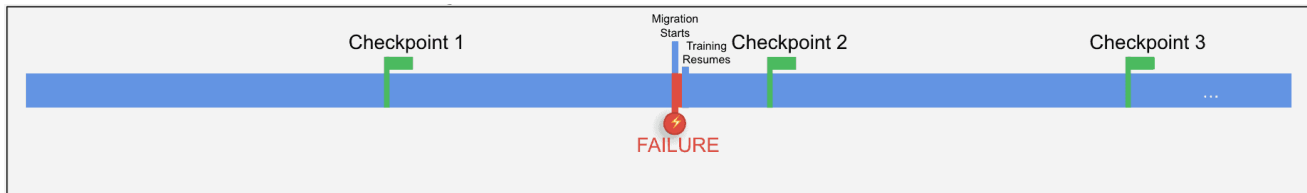
Checkpoint / Restart – the “industry standard”



Checkpoint and restart is the most widely used resiliency strategy. During training, periodic snapshots of the model state are written to persistent storage. Since every failure causes the entire job to abort, a recovery process must be initiated: provision healthy resources, restart the job, restore from the most recent checkpoint, and then recompute everything between that checkpoint and the failure.

It is highly effective, simple to adopt, and comes with a heavy penalty for each failure.

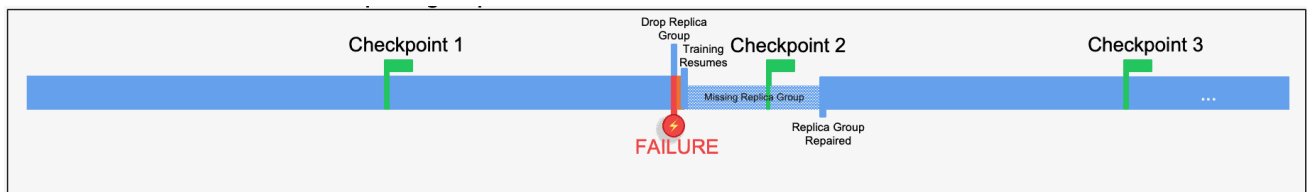
Clockwork TorchPass – live GPU migration



TorchPass is the approach we built. Instead of treating point-in-time checkpoints as the recovery mechanism, TorchPass captures model state at the moment of failure and migrates it to a healthy spare resource without causing the training job to abort.

From a training job's perspective, the failure looks like a short pause – about 100 seconds for a 109B-parameter model in our testing – after which training resumes from the exact step where it paused.

TorchFT – fault tolerance via replica-group removal



TorchFT, written by a team of Meta engineers, takes a different approach. When a failure occurs, the impacted replica group is removed entirely and training continues on the surviving groups. Training rapidly proceeds, but all samples assigned to the missing replica group are never processed. If a replacement later becomes available, it can rejoin the quorum so training can continue with all replica groups.

A Framework for Deciding on a Training Fault Tolerance Approach

The framework in the diagram below provides a helpful way to help decide which is best suited to your workload, infrastructure and requirements.

1. Failure types What are the most common failure types? What types of failures does the FT approach protect against? Does it use the same method for all? failure types?	
2. Checkpointing When are checkpoints taken? What happens at recovery? Resources needed for recovery?	
3. Model code changes What code changes are required to implement the fault tolerance strategy? What is the complexity of the changes?	
4. Performance impact What is the performance impact at steady state and upon failure?	
5. Training semantics Are training semantics altered before, during or after a failure?	

We'll walk through each of these elements:

1. Failure types

Failures can be bucketed into three groups:

Predictable ¹	Transient or Recoverable	Permanent Failures
• Rising GPU temperature • High-rate of correctable ECC (Xid 92) • Fan speed degrading • Power fluctuations (Xid 54) • NVLink marginal errors (Xid 74) • SSD SMART data degrading • Planned outages (e.g. patching/updates)	Software • Software bug (12.9%) • NCCL watchdog timeout • App exception (e.g. Xid 13) • Deadlock Hardware (some resolve on reboot/reset) • Transient network link flap or NVLink single link (Xid 74) • GPU thermal throttling • DRAM UCE contained (Xid 48, 63, 94) • PMU errors (Xid 61, 62)	• Faulty GPU incl. GPU fallen off bus Xid 79 (30.1%) • GPU HBM3 Memory (17.2%) • Network switch / cable failure (8.4%) • GPU SRAM (4.5%) • GPU system process (4.1%) • NVSwitch / NVLink fatal (SXid) • Fan failure (complete) • PSU failure (sudden) • CPU / system memory failure • SSD failure

¹Predictable detection requires implementation of comprehensive monitoring and threshold configuration
More than 75% of failures attributed to issues in red (Llama 3 results). Some could be predictable.

- **Predictable** – These are not yet failures! They are warning signs in telemetry (e.g. rising GPU temperatures, climbing correctable-ECC rates, fan slowdowns, etc.) that indicate a failure is likely. With the right monitoring and threshold configuration, these can be caught early and handled gracefully.
- **Transient or recoverable** – These are faults that clear on a restart or reset (e.g. software bugs, NCCL watchdog timeouts, link flaps, etc.). They’re disruptive in the moment but don’t require physical replacement to resolve.
- **Permanent failures** – Hard faults (e.g. faulty GPUs, HBM memory errors, dead network switches, etc.) that require physical replacement. These dominated the Llama 3 incident data.

The implication: a strategy optimized for a GPU falling off the bus may not be the optimal approach to deal with predictable conditions in advance. When evaluating an approach, ensure it is well suited to the failures that occur most frequently in your environment.

2. Checkpointing

Checkpointing, or some form of state capture, is used in different ways across all the fault tolerant approaches. But before describing the differences, it's worth deep diving into checkpoint / restart since it's the most common strategy and can also be used as a fallback strategy for all approaches.

Taking Periodic Checkpoints

Checkpoint load and save functions are native to PyTorch via the Distributed Checkpoint library (DCP). The framework or model builder will typically call `torch.distributed.checkpoint.save` to take a checkpoint every N steps. Choosing the right value for N is a trade-off between:

- **Checkpoint overhead** – How long are GPUs blocked for?
- **Storage and bandwidth cost** – Especially relevant for shared persistent storage.
- **Tolerable recompute** – How much work are you willing to lose when a failure occurs?
- **Expected failure frequency** – The more things break, the more aggressive N has to be.

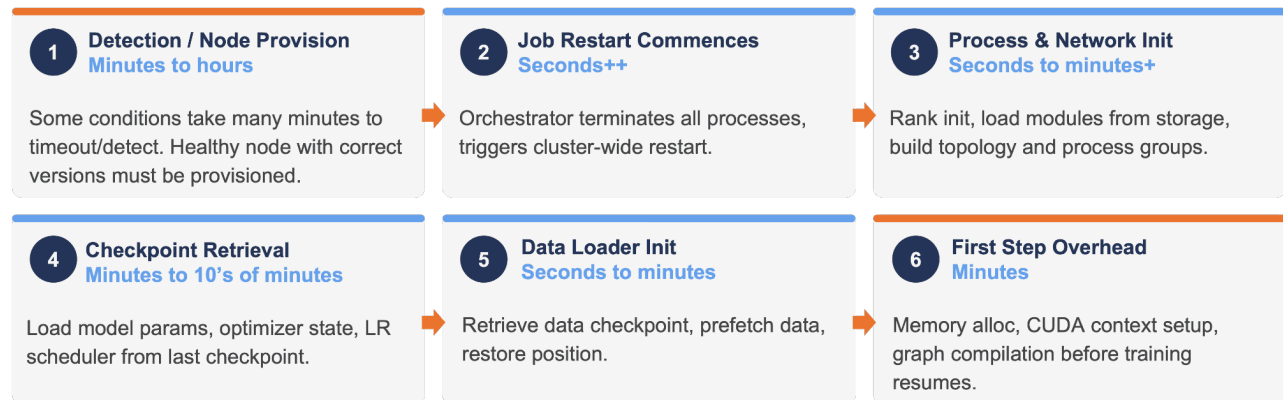
Checkpoint Types

- **Synchronous checkpointing** is the most common and foolproof mechanism. GPUs are blocked and training freezes while the model state is written to persistent storage (which can be many minutes for large models).
- **Asynchronous checkpointing** minimizes blocking by writing state to memory and then asynchronously copying it to persistent storage (often local NVMe for performance and shared storage for resiliency against node failure). It comes with higher memory cost and implementation complexity.

Recovery Process

When a failure forces a checkpoint restart, there is a multi-stage restart penalty (which covers detection, resource provisioning, restore and restart) as well as a recompute penalty (since all the work between the last checkpoint and the failure has to be recomputed). As shown in the diagram below, this can quickly add up to an hour or more each time there's a failure.

Restart Penalty



Recompute Penalty

Steps since last checkpoint was taken must be recomputed once job restarts (lost work)

Optimizations

Several optimizations exist to speed up parts of this process, including TorchElastic (PyTorch core since 1.9) and NVIDIA's Resiliency Extensions (NVRx, experimental as of May 2026). They offer capabilities like faster failure detection, in-process and in-job optimized restarts, and elastic world-size reduction, but both remain fundamentally checkpoint / restart based strategies.

TorchElastic (part of PyTorch core since 1.9) optimizes distributed job restarts by detecting failures, killing and relaunching worker groups and re-establishing rendezvous. It also supports world-size changes on resumption (assuming the training job is written to tolerate elastic world-size changes.), which means you can recover a job without requiring spare resources.

This concludes the deep-dive on checkpoint-restart! The following section compares how the three approaches use checkpoints.

Comparing how the three approaches use checkpoints

For comparison purposes, it's useful to look across three dimensions – timing, the recovery process and recovery resources.

Timing – When are checkpoints taken?

- **Checkpoint / Restart** performs periodic point-in-time checkpoints (e.g. hourly) which are used as the primary mechanism for recovery after failure.
- **TorchPass** captures state at the moment of failure and uses it for immediate live GPU migration.
- **TorchFT** captures state when a failed replica group rejoins the quorum.

Recovery process – What happens at recovery?

- **Checkpoint / Restart** recovers by performing a full job restart and a checkpoint restore, after which lost work between the last checkpoint and the failure must be recomputed.
- **TorchPass** keeps the training job running by performing a live GPU migration.
- **TorchFT** keeps the training job running by dropping the impacted replica group (samples assigned to the dropped replica group are never processed)

Recovery resources – what resources are needed?

The resource needed for recovery has cost implications. There are typically three choices:

- The same resource if it is still healthy (this is the case for transient or recoverable failures)
- A spare resource or resource preempted from a lower-priority job
- No spare needed – this requires a fault tolerant strategy that supports training on a reduced world size

Depending on the failure type, checkpoint/restart, TorchPass and TorchFT can resume using the same resource if it remains healthy, or a spare/preempted resource if replacement is needed. TorchFT also supports resumption without a spare (recall that it drops the impacted replica group entirely). Similarly, implementing checkpoint / restart with TorchElastic or NVRx can also add support for resumption with a reduced world size.

It's also important to consider the "blast radius" of the failure. The blast radius for checkpoint / restart is the entire job. For TorchPass it's down to an individual GPU. TorchFT has a blast radius equal to a replica group (i.e. if a single GPU fails in an 8-node x 8-GPU replica group configuration, all 64 GPUs will not be used).

3. Model code changes

The effort requirement to implement fault tolerance varies across the approaches:

- **Checkpoint / Restart** – Standard checkpoint/restart generally requires low or no incremental model-code changes as it is already built into common training frameworks. NVRx optimizations can require significant changes, especially when using the Sections API to improve hang detection.
- **TorchPass** – TorchPass offers two integration modes: **Model Transparent** requires no model-code changes because it operates below the model layer, using infrastructure-level CRIU/CUDA snapshot mechanisms. **Model Aware** requires minor code changes (the user imports the TorchPass library and registers existing load/save functions). Model Aware typically has a

performance benefit.

- **TorchFT** – Requires a medium implementation effort as the training loop must integrate with TorchFT's Manager, model and optimizer wrappers, and fault-tolerant process-group implementations. Users typically provide `state_dict` / `load_state_dict` hooks and run a Lighthouse service to coordinate replica groups through heartbeat/quorum logic. TorchFT handles much of the recovery and reconfiguration, but the application still has to be structured around its step-level recovery model.

4. Performance impact

Performance is arguably the most important dimension since it directly impacts the cost of training. Here's a summary of performance differences at steady state and upon failure across the three approaches:

- **Checkpoint / Restart**
 - **Steady-state:** each checkpoint takes tens of seconds to minutes depending on model-size and destination (blocking time depends on sync/async and in-memory/persistent).
 - **On failure:** as described above, there is a heavy restart penalty and recompute overhead measuring 10's of minutes to an hour or more.
- **TorchPass**
 - **Steady-state:** no measurable overhead.
 - **On failure:** a short pause after which training resumes normally (~100 seconds measured on Llama 4 MoE Scout 109B).
- **TorchFT**

- **Steady-state:** TorchFT uses NCCL (NVIDIA Collective Communication Library) within replica groups but Gloo for cross-replica all-reduce. Gloo is the production-proven choice because process groups across replicas need to handle dynamic membership changes that NCCL doesn't gracefully support (e.g. ncclCommAbort can deadlock on teardown after mid-step failure). Since Gloo operates over TCP/IP sockets (data passes through the CPU) there is a very heavy steady-state penalty (measured at ~50% in the benchmark below). This is explained in further detail [here](#).
- **On failure:** a short pause while the failed replica group is dropped (30–120 seconds measured on Llama 4 MoE Scout 109B).

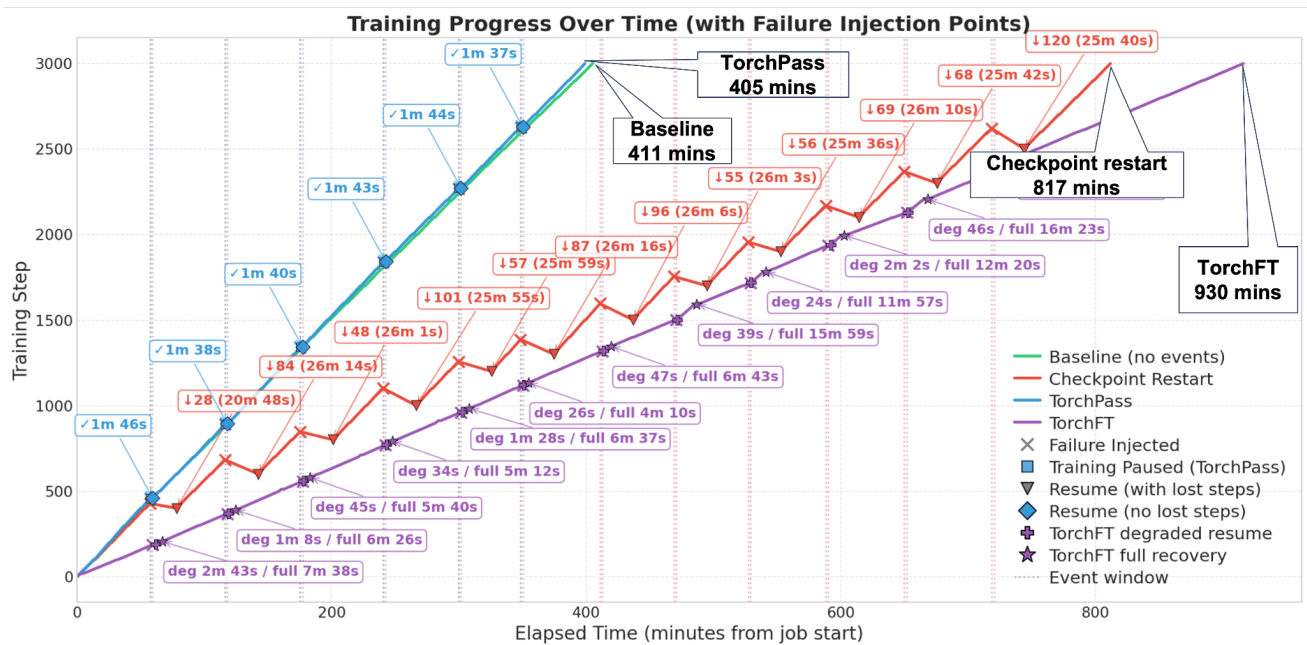
We benchmarked the fault tolerance performance of all three approaches by randomly injecting failures during TorchTitan Llama-4 MoE Scout (109B) training using the following configuration in Google Cloud:

Attribute	Details
Compute	64 x NVIDIA H200 GPUs (8 nodes x 8 GPUs), GCP
Networking	NVLink within node, 8 CX-7 NICs (RoCE v2) internode.
Model	TorchTitan Llama 4 MoE Scout (109B)
Parallelism	PP=4, DP=2, FSDP=4, TP=2, EP=4, ETP=2
Training steps	3,000
Checkpoint Interval	Every 100 steps (asynchronous)

We ran four tests sequentially (the same random number generator seed was used across all tests so that the failures occurred at the same points):

- **Baseline** – no failure injection.
- **Checkpoint Restart** – failures injected randomly every 45-75 mins
- **TorchPass** – failures injected randomly every 45-75 mins
- **TorchFT** – failures injected randomly every 45-75 mins

Upon completion, TorchPass reached 3,000 steps the fastest at 405 minutes (effectively matching the baseline run within run-to-run variation), checkpoint restarts took 817 minutes (2X slower) and TorchFT 930 minutes (2.3X slower). The results are shown below, and full details of the benchmark can be found [here](#).



5. Training semantics

Looking at the training semantics helps answer the question of whether failure recovery impacts the training trajectory the job would have followed without failures. **Checkpoint / restart** and **TorchPass** do not change the training semantics. Since **TorchFT** drops the impacted replica group upon failure (and doesn't process samples assigned to that replica group while it is missing), training semantics are altered. Depending on the frequency and length of failures, this can be acceptable because model convergence is generally robust to such perturbations.

Choosing the Right Solution

	Std Checkpoint / Restart	Clockwork TorchPass	TorchFT
Failure Types	Restart, restore & recompute is used for all scenarios.	Supports both planned and Unplanned migration depending on failure scenario.	Impacted replica group dropped for all scenarios.
Checkpoint Timing	Periodic point-in-time	State capture taken at time needed	State capture when healthy node rejoins
Recovery Process	Full job restart, checkpoint restore and recompute since last checkpoint.	Keeps job running by performing a live GPU migration.	Keeps job running by dropping impacted replica group (assigned samples are never processed).
Recovery Resource	Same (if healthy): ✓ Spare: ✓ No Spare: ☒	Same (if healthy): ✓ Spare: ✓ No Spare: ☒	Same (if healthy): ✓ Spare: ✓ No Spare: ✓

Code Changes	DCP built into PyTorch and standard across frameworks	Model Aware: minor changes Model Transparent: none	Medium-complexity changes required
Performance Impact	Steady-state: sync CPs block; async less impact Failure: heavy restart penalty (min-hrs)	Steady-state : none Failure: ~1–3 min pause	Steady: very large overhead from Gloo Failure: short pause
Training Semantics	Unaltered	Unaltered	Altered; dropped RG samples not processed

The table above provides a color-coded summary of the three fault tolerance approaches across the comparison framework. Each solution has its merits: Checkpoint / restart is already standard and well-proven. TorchPass has the fastest performance. TorchFT tolerates resumption with a reduced world size. Each also has trade-offs: Checkpoint / restart has a heavy penalty on each failure, TorchPass requires spare resources and TorchFT adds a very significant performance overhead at steady state. For most teams, performance is the deciding factor since every extra hour of training translates into thousands of dollars at scale.

For the full session, including Q&A and a discussion on the TCO implications of fault tolerance with Jordan Nanos from SemiAnalysis, watch the webinar: [Handling Failures During Training: A Comparative Analysis of Fault Tolerant Training Frameworks](#). If you'd like to evaluate Clockwork TorchPass on your own training workload, please [get in touch](#).