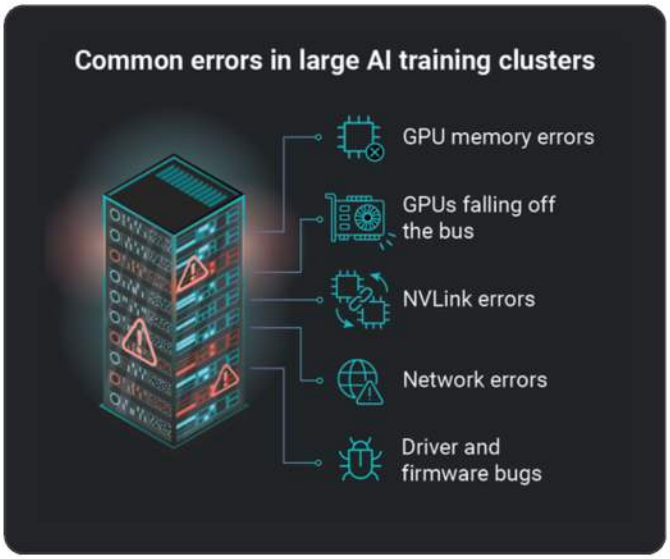
CLOCKWORK.IO

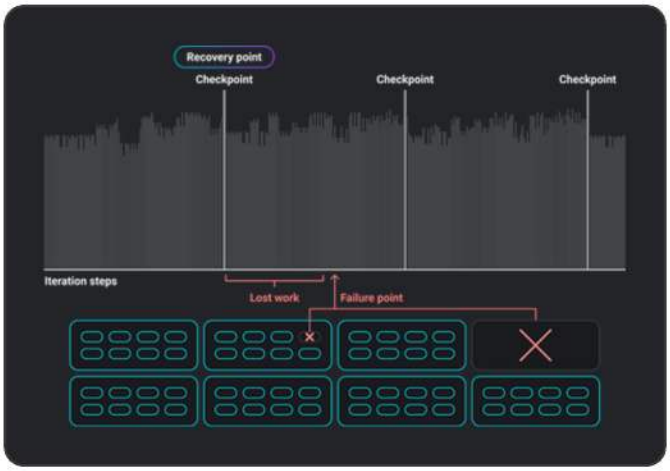
Live GPU Migration for Resilient Distributed Training

TorchPass Technical Overview

Large-scale GPU training jobs are inherently fragile. In clusters with thousands of GPUs participating in distributed training, hardware failures are daily occurrences. GPU memory errors, network link flaps, NVLink failures, power supply issues, and thermal events can crash an entire training job, wasting thousands of GPU-hours and forcing recovery from the most recent checkpoint, which typically wastes 30 minutes to several hours of wall-clock time per incident, and training clusters at scale experience multiple such incidents daily.



TorchPass is an infrastructure software solution for live GPU migration. It sits between the training framework (PyTorch) and the cluster scheduler. It automatically detects when a node is experiencing problems or has failed, provisions a healthy replacement, and migrates the training state from the problematic node to the replacement, minimizing restore and restart time and eliminating lost training progress. From a training job's perspective, there is just a short pause after which training resumes with no lost progress.



From a training job's perspective, there is just a short pause after which training resumes with no lost progress.

TorchPass is particularly well suited to NVL72 rack-scale systems where every GPU is topologically equivalent to every other GPU in the rack. When TorchPass migrates a training worker from a failing GPU or from a failed tray (node) to a spare in the same rack, there is no topology penalty. Operators are already adopting a practice of setting aside 1 or 2 trays as spares that TorchPass can leverage to enable continuous training.

Without TorchPass, the state of the art is to restart the entire training job from the most recent persistent checkpoint - losing GPU hours and increasing job completion time due to restart, restore and recompute. TorchPass enables a radically different paradigm - live GPU migration to a spare node, often already in place and waiting – delivering a dramatically different SLA!

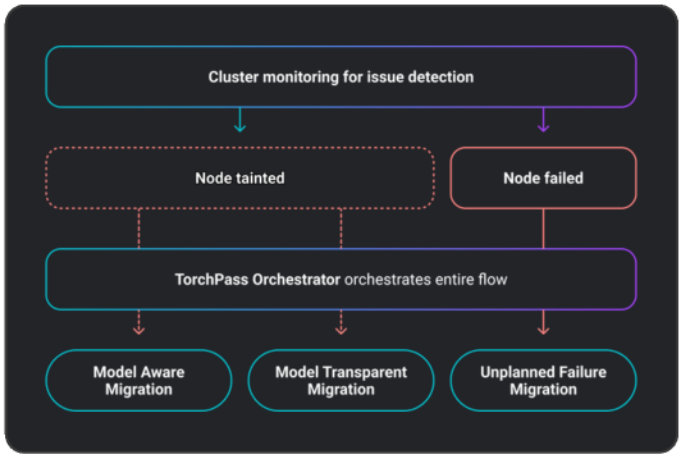
TorchPass Mechanisms

TorchPass provides 3 migration mechanisms:

1. **Planned Migration (model-aware):** Offers the fastest migration path for planned events. Since it uses the model's own checkpoint save and load functions, minor code changes are required to register the functions with TorchPass at training startup.

2. **Planned Migration (model-transparent):** Offers a migration path for planned events that does not require any model code changes. Rather than using the model's checkpoint load and save functions, it uses CRIU (Checkpoint Restore in User Space) and NVIDIA's CUDA-checkpoint technology to capture the complete execution state of the training worker process and restore it on a different node.

3. **Unplanned Migration (model-aware)**: Offers a migration path for sudden failures where a node, GPU or other component has already crashed and the affected workers can no longer be checkpointed. Instead, JIT (just-in-time) checkpoint recovery reconstructs the lost state surgically from surviving data-parallel peers and places it on a spare resource. It requires only minor code changes since it leverages the model's checkpoint save and load functions.

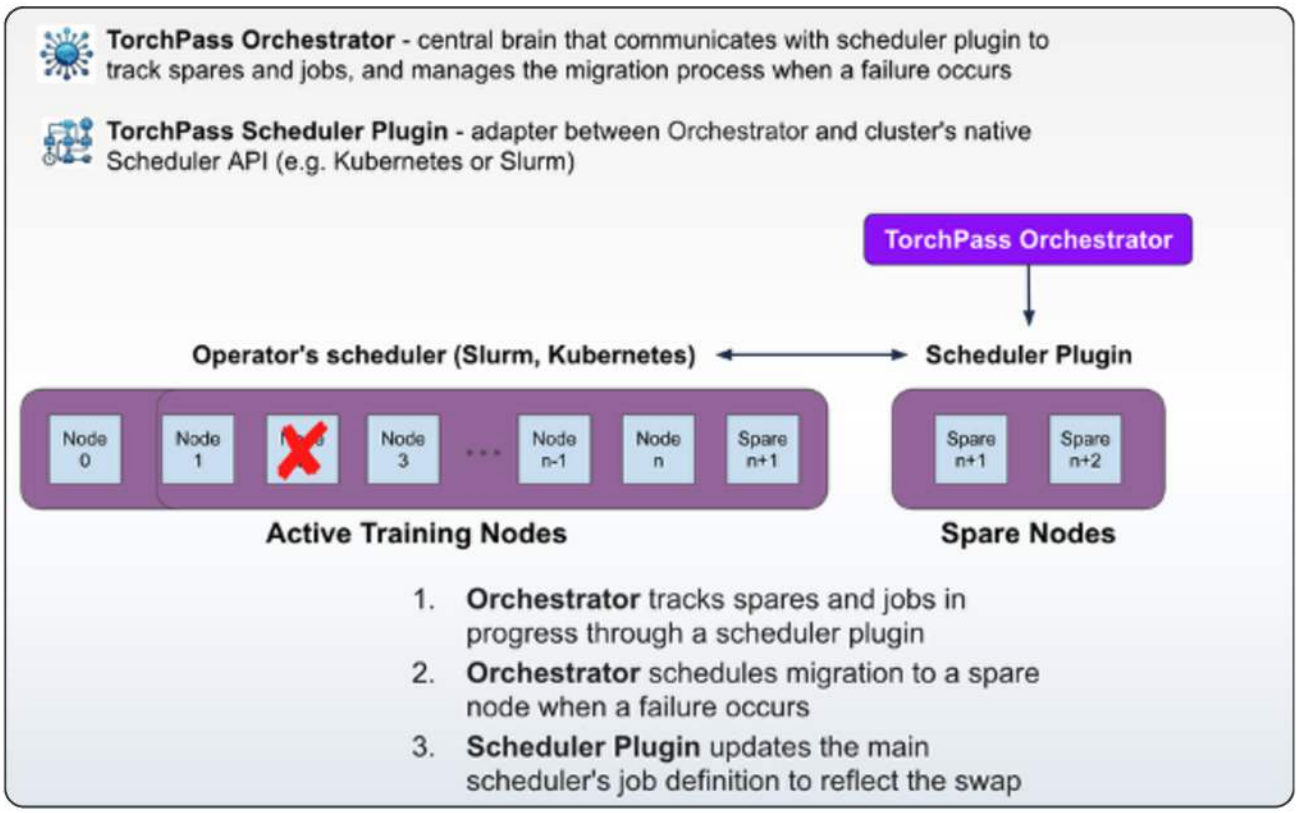


Whether it's planned or unplanned, the training application is completely unaware that a migration has occurred and, after a brief pause during migration, training resumes at the exact same iteration step. Note that no code changes or library imports are required for model-transparent migration, making it well-suited to cloud operators that serve diverse tenant workloads and cannot mandate changes to customer training code.

By contrast, model-aware migration requires only very small, well-bounded changes to training code - importing the TorchPass Python library and registering existing checkpoint save/load functions.
The remainder of this document provides a technical overview and practical considerations for implementing TorchPass in a production environment.

TorchPass Control Plane Components

TorchPass is designed as a minimal-footprint addition to existing cluster infrastructure. The same control plane is installed regardless of which migration mode is used. What changes between migration mechanisms is how state is captured and restored, not what gets installed in the cluster. The TorchPass binaries are built for both X86 and ARM, so the same images can be deployed on x86 hosts and Grace ARM CPUs in NVL72 racks.



The control plane has the following components:

TorchPass Orchestrator

The orchestrator is the central coordination service for TorchPass. It runs as a single long-lived process and is responsible for tracking all active training jobs, monitoring node health through the scheduler plugin, deciding when migrations should occur and coordinating the migration process across all affected nodes.

One orchestrator is deployed per cluster (either as a Kubernetes Deployment or as a systemd service on the Slurm controller in Slurm environments). It uses gRPC for communication with per-node agents and the scheduler plugin. It also uses gRPC to communicate with a web port that exposes a monitoring dashboard.

Scheduler Plugin

The scheduler plugin is a thin adapter layer between TorchPass and the cluster's native scheduling infrastructure. Its job is to translate between TorchPass concepts (spare nodes, taint events, migration decisions) and the scheduler's native mechanisms (Kubernetes pod lifecycle, Slurm job management). It reports node health status, responds to spare provisioning requests, and notifies the orchestrator when jobs start or exit. It injects the Launchpad, a `torchrun` replacement, as a per-node agent to manage PyTorch worker processes and commute with the TorchPass orchestrator. The scheduler plugin also arranges for the CRIU companion and the NCCL Interceptor to be loaded into each worker process to coordinate NCCL quiescing during migration.

Scheduler plugins are provided for Kubernetes (Kubeflow v1) and Slurm:

Kubernetes with Kubeflow v1

Kubernetes support is provided through the Kubeflow v1 scheduler plugin (the scheduler interface is extremely lightweight, making it easy to add support for other Kubernetes-based job schedulers and managers). It runs as a Deployment alongside the orchestrator. At startup, it registers a Mutating Admission Webhook with the Kubernetes API server, which watches for the creation of PyTorchJob pods (managed by the Kubeflow training-operator v1). The plugin also monitors Kubernetes events and pod status changes to detect when nodes are tainted or pods fail.

When a training pod is created, the webhook intercepts and saves the pod specification, injects the environment variables that allow Launchpad (see below) to connect back to the orchestrator, and ensures that the TorchPass binaries are mounted into the container. The container's entry point is rewritten so the Launchpad replaces torchrun for that pod. The net effect is that training jobs submitted through the standard Kubeflow PyTorchJob interface are automatically instrumented with TorchPass capabilities, with no changes to the job specification required.

Slurm (including Slurm on Kubernetes)

Slurm support ships as a single .deb or .rpm package (torchpass-slurm) that installs onto the Slurm controller. The package contains the scheduler, the orchestrator and the Launchpad. An sbatch wrapper intercepts Slurm job launches and a TaskProlog script injects environment variables that allow the Launchpad to connect back to the orchestrator and ensures the TorchPass binaries are mounted into the container.

The Slurm scheduler plugin detects node health issues by monitoring Slurm's DRAIN state. When a node is placed into DRAIN state (either manually by an administrator or automatically), the plugin reports this to the orchestrator as a taint event, triggering the migration process.

Launchpad

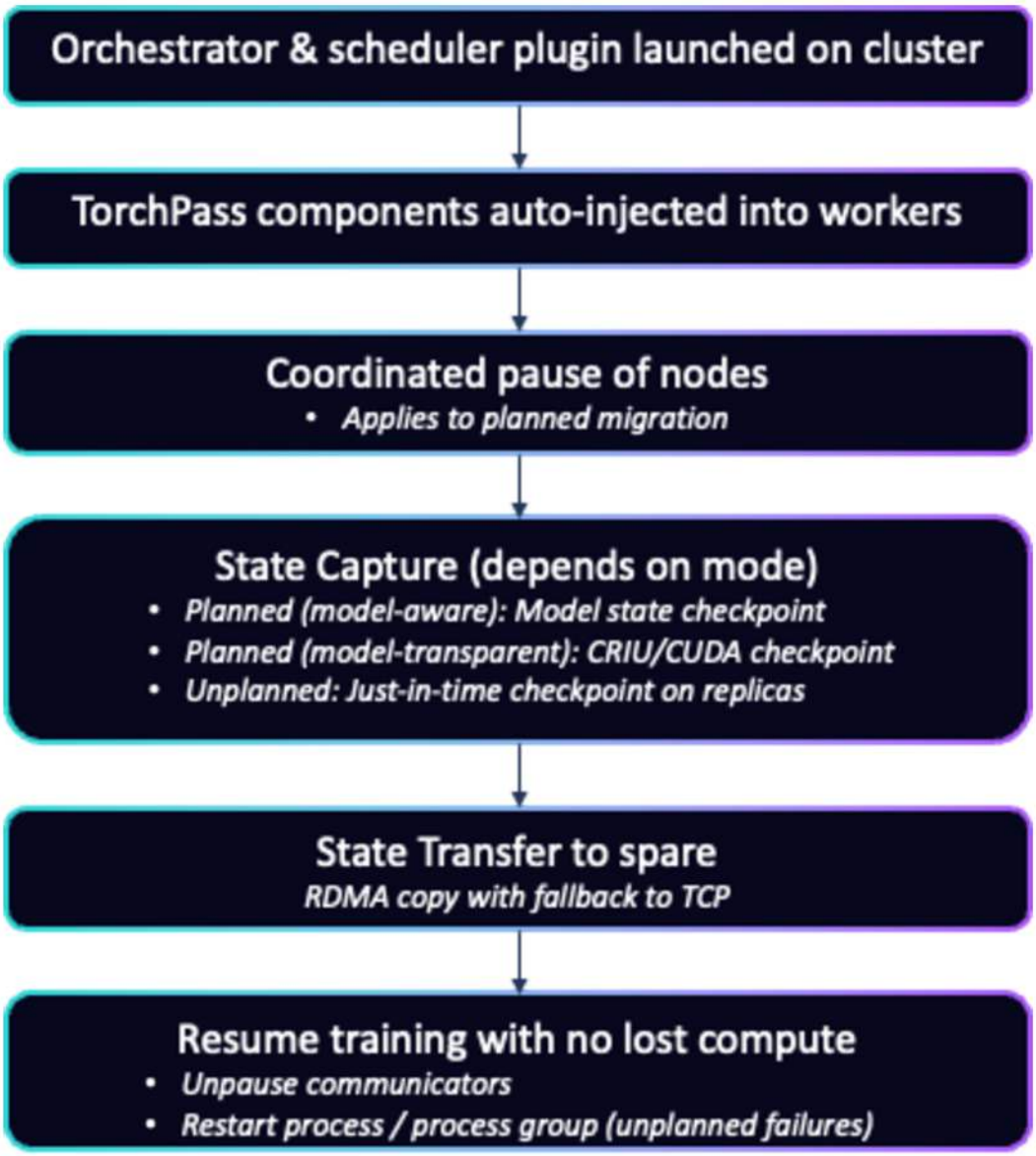
Launchpad runs as the per-node parent of the PyTorch worker processes. It is transparently swapped in by the scheduler plugin as a replacement for torchrun. Launchpad supervises the local workers, sets up their environment, and stays in contact with the Orchestrator throughout the job.

Launchpad also hosts wormhole, a UCX-based state transfer engine that migrates state between leavers and joiners. It chooses the best fabric option available and supports NVLink, RDMA (RoCE and InfiniBand), TCP and EFA. Wormhole is NUMA-aware (one stream per NUMA node). It abstracts the state transfer step so it is identical across all three migration modes.

How These Components Interact

The operational flow from the Cluster Operator's perspective is straightforward. The Operator deploys the orchestrator and the appropriate scheduler plugin (Kubernetes or Slurm). For Kubernetes, the Helm chart creates all necessary Kubernetes resources including Deployments, Services, ServiceAccounts, ClusterRoles, and the Mutating Admission Webhook configuration. Training jobs are submitted through the existing workflow (Kubeflow PyTorchJob or Slurm sbatch). The webhook or Slurm prolog script transparently instruments the containers. The orchestrator monitors health and performs migrations as needed. The Operator observes the system through standard Kubernetes and Slurm monitoring.

The Migration Process



Part I: Planned Migration

Planned migration covers two use cases. The first is **preemptive** where the node is still operational, but telemetry indicates a failure is imminent (e.g. rising correctable-ECC rates, climbing temperatures, fan slowdowns, accumulating NVLink errors, etc.). The second is **proactive** where the operator wants to vacate a node for maintenance such as firmware updates, security patching, workload rebalancing, or removing a known straggler.

The operator can choose between model-transparent and model-aware planned migration mechanisms:

- **Model-transparent** does not require any model code changes, making it well-suited to cloud operators or operations teams that cannot mandate changes to customer training code.
- **Model-aware** requires very minor, well-bounded changes to training code - importing the TorchPass library and registering existing save/load functions. It typically provides the fastest migration performance.

Both mechanisms share the steps below for detection, scheduling, pause, transfer, and resume; they differ only in how state is captured and restored.

How Planned Migration Works

1. Detections

Hardware degradation in GPU clusters manifests through a variety of signals: rising ECC error counts in GPU memory, a GPU becoming unresponsive to the PCIe bus (“falling off the bus”), temperatures exceeding safe thresholds, NVLink errors accumulating above a threshold, power supply irregularities, or cooling fan degradation. Operators typically have existing health checks, or can use Clockwork Fleet Monitoring, to monitor hardware telemetry across the cluster. When a health check determines that a node is exhibiting signs of impending failure, it marks the node as unhealthy. In a Kubernetes context, this typically results in the node being tainted. In a SUNK/Slurm context, the node is placed into DRAIN state.

The distinction between a taintable condition and an actual failure is critical. A taintable condition means the hardware is still functional but is being deliberately vacated. This gives TorchPass a window of opportunity to perform a deliberate, orderly migration before the hardware actually fails or is shut down for maintenance, preserving training state with zero lost work.

2. Scheduler Plugin Reaction

The TorchPass scheduler plugin continuously polls the cluster state. For Kubernetes, it watches node and pod events through the Kubernetes API. For Slurm, it queries node states via Slurm’s management interfaces. The polling interval is configurable (default 5 seconds). When the scheduler plugin detects that a node participating in an active training job has been tainted or drained, it reports this event to the orchestrator via gRPC. The report identifies the affected node, the training job it belongs to, and the specific workers running on that node.

3. Orchestrator Decision

The orchestrator receives the taint report and determines the appropriate response. It identifies which workers on the tainted node are “leavers” (workers that need to be moved) and looks up which migration mechanism the job is configured for - model-aware (if the training script imported the torchpass library and registered save/load functions) or model-transparent (the default, used when the job has no TorchPass code-level integration). It then calculates how many spare nodes are needed and requests them from the scheduler plugin. The scheduler plugin responds by provisioning spare resources (in Kubernetes, creating new pods with the same container specification as the original training pods but marked as spares; in Slurm, submitting new jobs with the spare designation). These spare resources start up, their Launchpad agents register with the orchestrator, and the orchestrator waits until all required spares are available. A configurable timeout governs how long the orchestrator will wait for spares before falling back to alternative recovery strategies.

4. Coordinated Pause of All Workers

Before any state can be captured, all workers in the training job must be brought to a globally consistent point. This is necessary because distributed training involves NCCL collective operations (AllReduce, AllGather, etc.), and if some workers are in the middle of a collective while others are paused, the collective will deadlock.

TorchPass achieves this by tracking all collective communication operations and pausing them in a coordinated fashion across all workers (not just the tainted worker) simultaneously. When the orchestrator decides to migrate, TorchPass ensures that every worker stops issuing new collective operations and waits for any in-flight operations to complete. Once all workers confirm they are paused at the same logical point, the system has a globally consistent state from which state capture can safely proceed.

5a. State Capture: Model-Aware Path

In model-transparent planned migration, the training script has no knowledge of TorchPass and cannot help with state capture. Instead, TorchPass captures state from outside the process.

With all workers paused, the orchestrator instructs Launchpad on each leaver node to capture a complete process checkpoint. CRIU (Checkpoint/Restore in Userspace), a standard Linux tool, captures the entire execution state of the PyTorch training process. This includes all host (CPU) memory: the Python interpreter state, all Python objects, the PyTorch runtime, model parameters stored in CPU memory, optimizer state, data loader state, random number generator states, and any other memory allocated by the process. Additionally, Launchpad leverages NVIDIA’s cuda-checkpoint, invoked through CRIU, to capture GPU device memory contents, CUDA context state, active stream states, and allocated device memory records. The cuda-checkpoint process works by completing all submitted GPU work, then copying device memory contents to a staging area. It also captures the process’s file descriptor table, signal handlers, thread states, and CPU register values. The NCCL interceptor, along with other network proxies, will quiesce and capture all network connections via RDMA, TCP, HTTP, or etc.
The resulting checkpoint is a complete snapshot of everything needed to resume the process from the exact instruction where it was paused.

6. State Transfer to the Spare Node

The captured state - whether a model state dictionary (model-aware) or a full process checkpoint (model-transparent) - must now be moved from the leaver node to the healthy spare node. Launchpad's wormhole state-transfer engine handles this using the fastest available transport. On clusters with RDMA-capable networking (InfiniBand or RoCE), the wormhole uses UCX to establish RDMA reliable connections (RC queue pairs) and performs a receiver-driven pull of the checkpoint data. This can saturate the available network bandwidth, transferring tens of gigabytes in seconds. On clusters without RDMA, the data is transferred over standard TCP. On rack-scale NVL72 systems, intra-rack transfers benefit further from the NVLink fabric's bisection bandwidth.

7a. State Restoration: Model-Aware Path

The spare node's Launchpad agent receives the model state dictionary and hands it to the registered load function. This load function is the same code path the model would use to resume from a persistent checkpoint, so no new code is exercised. The load function reconstructs model parameters on the spare's GPU, restores optimizer state, restores RNG state, and rewinds the data loader to the correct position. Once the load function returns, the spare worker is in the same logical state as the leaver was at the moment of pause.

7b. State Restoration: Model-Transparent Path

Launchpad on the spare node receives the process checkpoint and restores the process using CRIU. The restoration reconstructs the process's memory layout, reloads GPU memory contents onto the spare node's GPU (cuda-checkpoint supports GPU UUID remapping, so the process can be restored on a different physical GPU than the one it was checkpointed from), restores file descriptors, and positions the program counter at the exact instruction where execution was paused.

8. Resuming Training

With the leaver's state captured and restored on the spare, the leaver node's Launchpad agent exits cleanly, and TorchPass tears down any NCCL communicators that included the old leaver and rebuilds them to include the spare. Communicators between healthy workers that did not involve the leaver remain untouched. The orchestrator instructs all workers (both the healthy workers that were merely paused and the newly restored worker on the spare) to restart collective operations for model-aware or to resume for model-transparent. Training resumes at the exact iteration step where it was paused. No training progress is lost.

If the leaver node and the underlying fault is later cleared (e.g., the operator applies a patch and reboots the node), TorchPass can optionally migrate the worker back to its original location, keeping the spare pool topped up.

9. Timeline Summary

The end-to-end migration timeline depends on the size of the model, cluster-type and which mechanism is used. Model-aware planned migration was benchmarked using TorchTitan Llama-4 MoE Scout (109B) on 64 H200 GPUs with an average migration time of 101 seconds. Model-transparent planned migration on the same hardware is estimated to take approximately 180-240 seconds.

During the entire migration window, all workers in the training job are paused. After completion, training resumes at full throughput with zero lost iterations.

Part II: Unplanned Migration

Unplanned migration covers cases where failures have already occurred - a GPU has fallen off the bus, an uncorrectable HBM error, a node has lost power or a kernel has panicked. By the time the failure is detected, the leaver node is gone, and there is no opportunity to capture its state in place.

Model-aware unplanned migration requires minor, well-bounded changes to training code: importing the TorchPass library, registering existing save/load functions, passing jitc=True to the Manager constructor, and wrapping the per-iteration tensor-update region with the manager.crit_region() context manager.

How Model-aware Unplanned Migration Works

1. Detection: How a Hard Failure Surfaces

Hard failures present differently from taintable conditions. Common signatures include:

- **GPU fallen off the bus** - Xid 79 or similar; the driver loses PCIe access to the GPU and CUDA calls begin returning errors
- **Kernel panic / node power loss / NIC failure -** the entire node disappears from the cluster's view; existing TCP connections drop, gRPC heartbeats stop, the kubelet stops reporting
- **Process crash** - the training process exits unexpectedly (e.g., uncorrectable memory error, driver bug); surviving ranks block on the next NCCL collective
- **NCCL watchdog timeout** - all surviving ranks notice the missing peer when a collective times out

The orchestrator usually detects these failures within seconds primarily through Launchpad's heartbeat (which stops) and through scheduler-plugin events (pod marked as failed; Slurm node marked DOWN).

2. Orchestrator Decision

When the orchestrator confirms an unplanned failure, it identifies the missing rank(s) and determines whether model-aware unplanned migration is feasible. The mechanism requires some degree of state replication, which can usually be achieved via data parallelism (DDP or HSDP), so the missing rank's state can be reconstructed from surviving peers in its data-parallel group. Inner parallelism, FSDP, tensor parallelism (TP), pipeline parallelism (PP), expert parallelism (EP), and similar, is supported as long as it is wrapped by a data-parallel layer. Most modern large-model training configurations meet this requirement. Jobs that don't (for example, a pure tensor-parallel or pipeline-parallel layout with no outer data-parallel replication) fall back to standard checkpoint restart from the most recent persistent checkpoint.

If the failed node is still functional, the orchestrator will perform an audit to determine whether its CUDA (GPU), RDMA (Backend network), and UDP (Frontend network) are still functioning correctly. Otherwise, it requests one or more spare workers from the scheduler plugin, exactly as in planned migration. While spares are starting up, surviving workers continue to wait at the coordinated pause boundary (described next).

3. Coordinated Pause of Surviving Workers

The dead worker cannot pause since it is already gone. But the surviving workers need to be brought to a globally consistent state before reconstruction can begin. TorchPass detects in-flight collectives that involved the dead leaver, aborts them cleanly via Launchpad, and instructs all surviving workers to pause at the next safe boundary (typically the start of the next iteration step). Once every surviving worker confirms it is paused, the system has a globally consistent reference point, and reconstruction can proceed.

4. Just-in-time Checkpoint Capture from Surviving Peers

Since the leaver's state cannot be captured directly, TorchPass instead sends an abort signal (SIGTERM) to all remaining workers. The workers catch the signal and take a just-in-time (JIT) checkpoint of their state by invoking the registered save function.

5. State Reconstruction on the Spare

The joining worker - which has registered with the orchestrator and is now part of the parallel topology in the leaver's old slot - receives the relevant slices from the surviving peers. The model's registered load function is then invoked on the spare, exactly as it would be during a checkpoint restore. The load function rehydrates the missing rank's parameters, optimizer state, RNG state, and data loader position, leaving the joiner in the same logical state the leaver was in at the moment of failure.

6. NCCL Repair and Resume

With the spare populated, the orchestrator instructs all workers - the surviving peers and the newly populated spare - to restart collective operations. Training resumes at the exact iteration step where the failure occurred, with no recompute of completed work.

7. Fallback to Checkpoint Restart

If model-aware unplanned migration cannot complete (for example if more workers have failed than the parallelism configuration can recover from), TorchPass automatically falls back to standard checkpoint restart from the most recent persistent checkpoint. This fallback path is identical to what would happen without TorchPass; the goal is to never make recovery worse than the existing baseline.

8. Timeline Summary

The end-to-end timeline for model-aware unplanned migration typically takes approximately the same amount of time as for model-aware planned migration (see above).

Operational Dependencies and Requirements

This section describes the cluster-level and per-job dependencies that apply across all three TorchPass migration mechanisms, with notes on where the mechanisms differ in their requirements.

Sparing

Migration requires healthy or preemptable spare workers to be available as migration targets. Without spare resources, the orchestrator cannot perform a migration and will fall back to restarting the job from a persisted checkpoint. Spares are created on demand, i.e. they don't have to be preprovisioned as running pods or processes.

Provisioning spares is fast in practice because TorchPass identifies which image is being used when a job is submitted and preloads actively used training images on the resources that can be used as spares. On Kubernetes, the recommended setup is a labeled pool that TorchPass schedules spares onto. On Slurm, each spare is an independent batch job, so operators typically dedicate a high-priority spare partition or pre-reserve nodes, and TorchPass submits spare batch jobs against that capacity on demand. TorchPass also supports preemption of lower-priority job(s). This can be set-up at installation time by specifying the jobs that can be preempted. Job preemption only occurs when a TorchPass spare is needed and no additional capacity is available on the cluster.

When a migration is triggered, TorchPass asks the cluster's scheduler to create new spares immediately and waits up to 10 minutes by default (wait period is configurable) for them to come online. If only some arrive in time, the orchestrator proceeds with a partial migration of the workers it can move; if none arrive, it falls back to checkpoint restart.

Memory / Storage Requirements for State Capture

- **Model-aware Planned Migration** captures only the model state dictionary — parameters, optimizer state, RNG, and data loader position. The footprint is bounded by the model's own checkpoint size, typically a few hundred MB to a few tens of GB per rank.
- **Model-transparent Planned Migration** captures the complete process state, including all GPU device memory and all host (CPU) memory used by the training process. The storage requirement is therefore larger than just the model parameters. On GB200 systems with 192 GB of HBM3e per GPU, a fully utilized GPU can produce up to 192 GB of device memory checkpoint data alone. The host-side process footprint (Python interpreter, PyTorch runtime, data loader buffers, CPU-side tensors) can add tens of gigabytes on top of that. As a guideline a tmpfs should be provisioned to at least the sum of the GPU memory in use plus the host memory footprint of the training process per worker. If the tmpfs cannot be sized to be large enough, TorchPass can be configured to stage checkpoints on local NVMe storage instead, at the cost of somewhat slower checkpoint and restore times.
- **Model-aware Unplanned Migration** captures JIT checkpoints from surviving peers, with a per-rank tmpfs footprint comparable to Model-aware Planned.

Network Bandwidth

State transfer speed is directly determined by available network bandwidth between source and destination nodes. For RDMA-capable networks, transfer of a 40 GB Model-transparent checkpoint completes in approximately one second; smaller Model-aware and JIT payloads complete in fractions of a second. For TCP-based networks, transfer time increases proportionally. The Operator needs to ensure that the network fabric between potential migration source and destination nodes has sufficient bandwidth and low latency. In rack-scale deployments (such as GB200 NVL72), intra-rack transfers are typically very fast. Cross-rack transfers depend on the fabric bandwidth.

Container Runtime Requirements

The training containers must run with sufficient privileges for state capture to operate.

- **Model-transparent Planned Migration** requires CRIU's SYS_PTRACE and SYS_ADMIN permissions, and the container runtime must allow access to /proc for process introspection. The TorchPass Helm chart and Slurm Taskprolog script configure these capabilities automatically, but the cluster's security policies must permit them.
- **Model-aware Planned Migration** and **Model-aware Unplanned Migration** do not require elevated capabilities, since state capture happens through the application's own save/load functions.

Code Changes to Training Scripts

Code-change requirements vary by mechanism:

- **Model-transparent Planned Migration** requires zero changes to training code. The TorchPass binaries are mounted into the container by the webhook or TaskProlog and operate below the model layer.
- **Model-aware Planned Migration** requires the training script to import the torchpass Python wheel and construct a torchpass.Manager with get_state and set_state callables, then call manager.check() once at startup and manager.step() at the start of every iteration. Typically a handful of lines of code that reuse existing checkpointing logic.
- **Model-aware Unplanned Migration (JIT-C)** uses the same Manager but with jitc=True, and additionally requires wrapping the per-iteration tensor-update region with the manager.crit_region() context manager (one critical region per iteration). For most training loops this is a small structural change.

Fallback to Checkpoint Restart

In all three mechanisms, if migration cannot complete because spares are unavailable, because state capture or transfer fails, or because the failure pattern exceeds what the active mechanism can recover from, TorchPass automatically falls back to standard checkpoint restart from the most recently persisted checkpoint. The fallback path is identical to what would happen without TorchPass; the design goal is that TorchPass strictly improves on the baseline, never regresses below it.

Application to NVIDIA GB200/GB300 NVL72 Rack-Scale Systems

NVL72 is a rack-scale system that unifies 72 Blackwell GPUs and 36 Grace CPUs into a single NVLink domain. The rack contains 18 compute trays, each housing 2 Grace CPUs and 4 Blackwell GPUs (organized as two Superchips per tray). All 72 GPUs are interconnected via an NVLink Switch fabric (consisting of dedicated NVLink switch trays in the rack), providing 1.8 TB/s of bisection bandwidth across the full rack. The rack-scale NVLink domain is both the GB200/GB300's greatest performance advantage and its most significant reliability challenge. Because all 72 GPUs share a single NVLink fabric, the blast radius of a failure depends critically on which component fails.

Failure Type	Affected Component	Blast Radius	Typical Symptoms
Single GPU memory error (correctable ECC)	One GPU on one tray	One GPU; taintable, pre-emptive migration possible	Rising ECC error count; GPU still functional but degrading
Single GPU hard failure (uncorrectable ECC, fallen off bus)	One GPU on one tray	One GPU initially but can affect other GPUs on same node and may cascade if training collective stalls	GPU unresponsive, CUDA errors on affected worker
NVLink port failure (single link)	One GPU's NVLink connection	One GPU; collective bandwidth degrades for that GPU	Reduced NVLink throughput for affected GPU, potential collective timeouts
Tray-level failure (PSU, cooling, CPU crash)	All 4 GPUs and 2 CPUs on one tray	One tray (4 GPUs); training workers on that tray crash or become unreachable	All workers on the tray fail simultaneously; tray power or thermal alarm
NVLink switch tray failure	NVLink fabric segment	Potentially all 72 GPUs; NVLink domain may become degraded or partitioned	Widespread NVLink errors, collective operations fail across many workers
Rack-level power or cooling failure	Entire rack	All 72 GPUs; entire rack becomes unavailable	Complete loss of all workers in the rack
Network NIC failure	One tray's external connectivity	One tray's ability to communicate with other racks; intra-rack NVLink unaffected	Inter-rack collective operations fail for affected tray's workers
Liquid cooling loop degradation	Multiple trays sharing cooling manifold	Multiple trays may throttle or shut down	Temperature alarms, thermal throttling across affected trays

The key insight is that individual GPU failures are the most common failure mode, followed by tray-level failures. True rack-level failures (NVLink switch or power) are rare but catastrophic. With 18 compute trays in the rack, Neoclouds seem to be adopting a practice of designating 2 trays as spares: this yields 16 active trays (64 GPUs actively training) and 2 spare trays (8 GPUs held in reserve). There is opportunity to reduce the number of spare trays when using TorchPass.

Spare Considerations When Each Tray/Node is a Kubernetes Pod

In a standard NVL72 deployment model, each compute tray runs as a single Kubernetes pod (or Slurm job step) containing all 4 GPU workers. This means a failure on one GPU effectively taints the entire pod. When that happens, TorchPass migrates all 4 workers from the tainted Tray/Node to one of the Spare Trays/Nodes. The migration follows whichever mechanism the job is configured for: a graceful taint triggers planned migration (model-aware or model-transparent); a hard failure triggers an unplanned migration. In either case, all workers pause, state is captured (in place for taints, JIT-from-peers for hard failures), the data is transferred to the spare GPUs (over the NVLink fabric within the same rack, which is extremely fast since all GPUs share the same NVLink domain), and training resumes.

After a tray-level migration, one of the two spare trays is now fully committed to training, leaving one spare tray (4 GPUs) available for future migrations.

Summary

TorchPass eliminates one of the largest sources of wasted GPU-hours in distributed training - checkpoint/restart followed by recomputation of lost work due to failures. Three migration mechanisms - planned migration (model-transparent/zero code changes), planned migration (model-aware), and unplanned migration - cover the full spectrum of failure events from predictable hardware degradation and planned maintenance to hard crashes. Live GPU migration is driven by a lightweight and easy-to-install TorchPass control plane.
The result is a fundamentally improved SLA: training that doesn't stop when hardware misbehaves, with recovery times measured in seconds rather than the hours typical of full job restart.