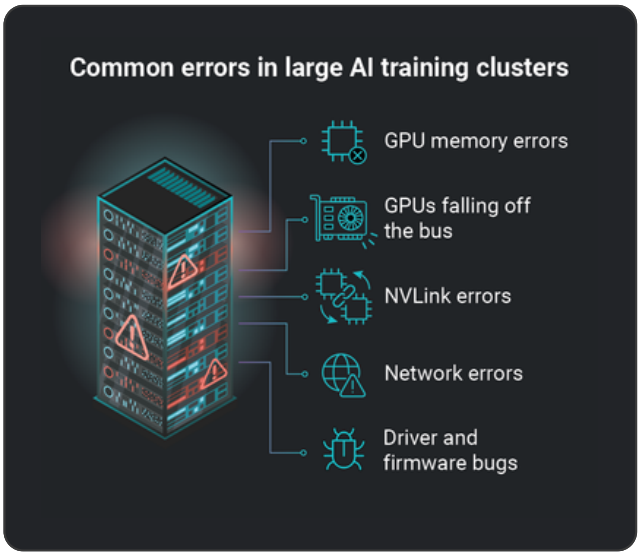
CLOCKWORK.IO

From taint-drain-checkpoint-restore to taint-and-migrate

Zero-integration, operator-controlled proactive fault tolerance for GPU training

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 that sits between the training framework (PyTorch) and the cluster scheduler. It automatically detects when a node is experiencing problems, provisions a healthy replacement, and migrates the live training state from the problematic node to the replacement, minimizing restore time, restart time and eliminating lost training progress.

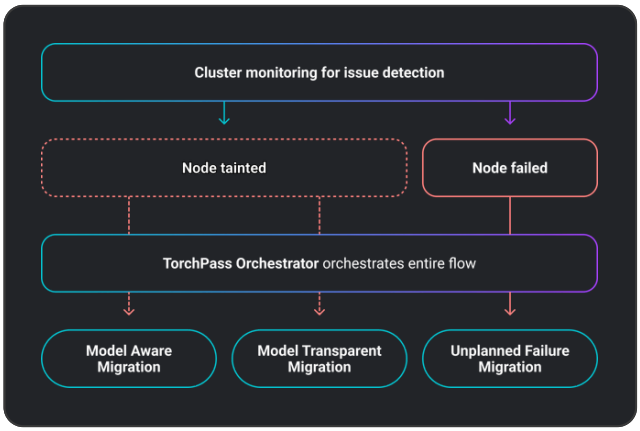
TorchPass is particularly well suited to NVL72 rack-scale systems where every GPU is topologically equivalent to every other GPU in the rack. When TorchPass migrates a training worker from a failing GPU or from a failed tray (node) to a spare in the same rack, there is no topology penalty. Operators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

The state of the art today is that Cluster Operators taint and drain a node, and let the tenant restore from a persistent checkpoint and restart the job—losing GPU hours and increasing Job completion time. TorchPass enables a radically different paradigm. Cluster operators can go beyond tainting, and automatically migrate the workload to a spare node, often already in place and waiting—a dramatically different SLA!

TorchPass Mechanisms

TorchPass supports multiple migration mechanisms to accommodate different operational environments:

1. **Model-aware Migration:** For teams that can make minor modifications to their training scripts, Model-aware migration offers the fastest migration path (approximately 30 seconds).
2. **Model-transparent Migration:** For operators who cannot modify model code or Python libraries even in small ways, Model-transparent migration facilitates non-disruptive migration of tainted nodes with no model changes.
3. **Unplanned Failure Migration:** For unplanned failures where a node has already crashed, JIT checkpoint recovery migrates state surgically from surviving workers onto spare workers, leveraging model-developer-provided checkpoint saving and loading functions.



This document focuses exclusively on the second mechanism, Model-transparent Migration, which leverages CRIU (Checkpoint Restore In User Space) and NVIDIA's cuda-checkpoint technology to capture the complete execution state of a training worker process, including all CPU memory, GPU memory, file descriptors, thread states, and CUDA context, and restores it on a different node. More detail on Model-aware migration is in the attached blog.

The training application is completely unaware that a migration has occurred, and training resumes after a pause of ~3 minutes at the exact same iteration step. **No code changes, no library imports, no checkpoint hooks—Model-transparent migration is well suited to cloud operators that serve diverse tenant workloads and cannot mandate changes to customer training code.**

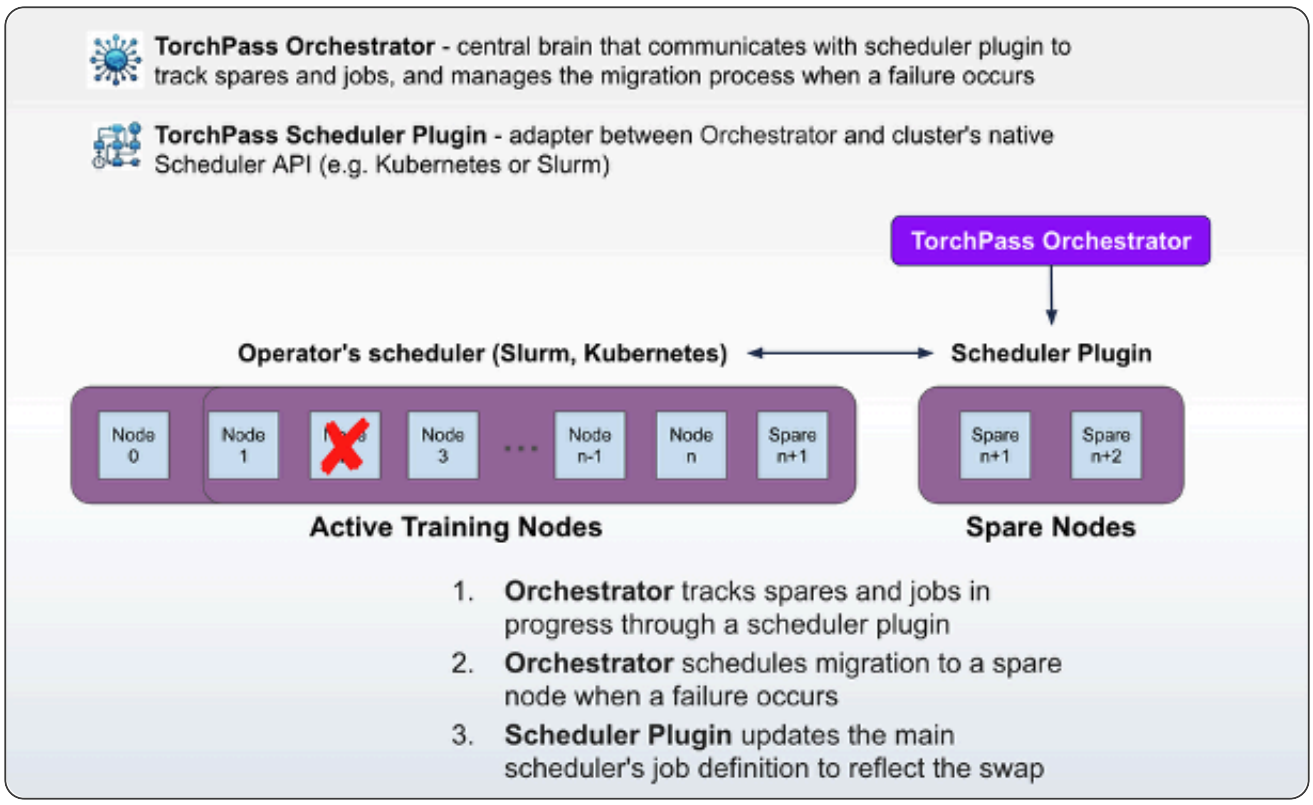
Model-Transparent Migration: What An Operator Has To Deploy

TorchPass is designed as a minimal-footprint addition to existing cluster infrastructure. There are two components that the cluster operator interacts with directly.

2.1 The TorchPass Orchestrator

The orchestrator is the central coordination service for TorchPass. It runs as a single long-lived process (deployed as a Kubernetes Deployment in Kubernetes environments) 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. It is not in the training critical path during normal



operation; it only becomes active during migration events. The orchestrator:

- Uses gRPC for communication with the per-node agents and the scheduler plugin
- Uses gRPC to communicate with a web port that exposes a monitoring dashboard.
- Hosts a PyTorch-compatible TCPStore service that replaces the default TCPStore that PyTorch normally runs inside the rank 0 training worker, ensuring that the coordination service remains available even if rank 0 is migrated. Each active training job gets its own store instance, providing the key-value coordination that PyTorch needs without depending on any single training worker's availability.

2.2 The Scheduler Plugin

The scheduler plugin is the 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.

For Kubernetes with Kubeflow

The Kubernetes scheduler plugin 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 training job pods (specifically, pods associated with PyTorchJob custom resources managed by KubeFlow). 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 the pod specification and environment variables that allow the per-node agent to connect back to the orchestrator, and ensures that the TorchPass binaries are mounted into the container. These binaries include logic that quiesces NCCL, invokes a CRIU system snapshot coordinated with a CUDA checkpoint, and migrates state. 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.

For SLURM (including Slurm on Kubernetes)

For SLURM, a TaskProlog script intercepts Slurm job launches and injects the environment variables that allow the per-node agent to connect back to the orchestrator, and ensures that the TorchPass binaries are mounted into the container. These binaries include logic that quiesces NCCL, invokes a CRIU system snapshot coordinated with a CUDA checkpoint, and migrates state, coordinated by the orchestrator.

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.

2.3 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, Slurm, or both). 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 the orchestrator's web dashboard or through standard Kubernetes and Slurm monitoring.

How Migration Works: From Taint to Resumed Training

This section walks through the complete lifecycle of a Model-transparent migration event, from the initial detection of a problem to the resumption of training on healthy hardware.

1. Detection: How a Taintable Condition Surfaces

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 health checks that continuously monitor hardware telemetry across the cluster (Clockwork Fleet Monitoring has its own health checks as well), and when the health checks determine 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 showing warning signs. This gives TorchPass a window of opportunity to perform a deliberate, orderly migration before the hardware actually fails, preserving training state with zero lost work.

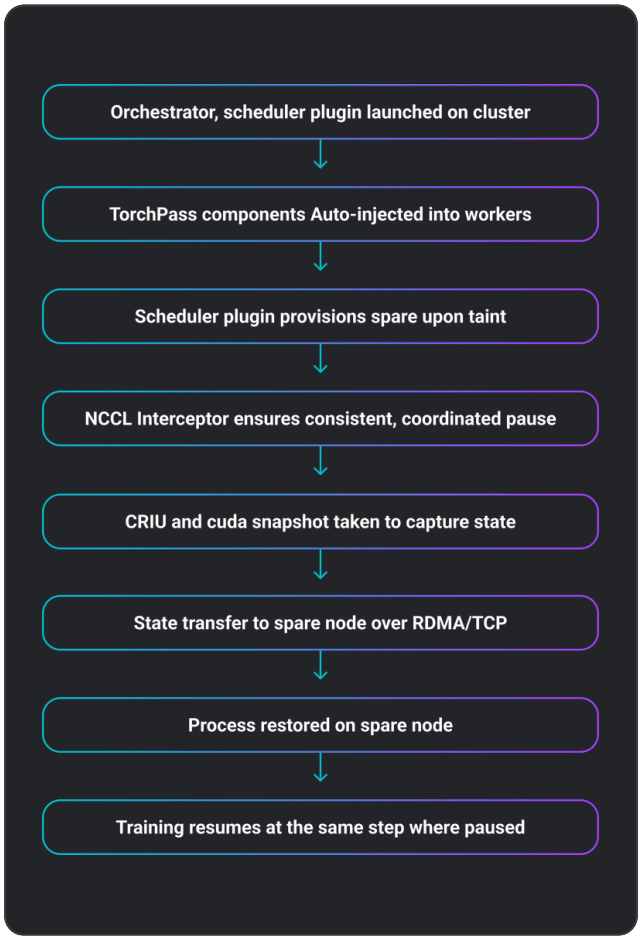
2. Scheduler Plugin Reaction

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 calculates how many spare nodes are needed to receive them.



The orchestrator then requests spare nodes from the scheduler plugin. The scheduler plugin responds by provisioning new containers (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 containers start up, their TorchPass 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 process state can be captured, all workers in the training job must be brought to a globally consistent state. 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 process checkpointing can safely proceed.

5. Process Checkpointing with CRIU

With all workers paused, the orchestrator instructs the state transfer engine on each leaver node to capture a complete process checkpoint. CRIU, 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, the state transfer engine leverages NVIDIA's cuda checkpoint, invoked by 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 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 checkpoint data must now be moved from the tainted leaver node to the healthy spare node. The TorchPass state transfer engine (wormhole) 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.

7. Process Restoration on the Spare

The spare node's state transfer engine receives the checkpoint data 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, the leaver node's TorchPass agent exits cleanly, and TorchPass tears down any NCCL communicators that included the old leaver node and rebuilds them to include the spare node. 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

resume collective operations. Training resumes at the exact iteration step where it was paused. No training progress is lost.

9. Timeline Summary

The end-to-end migration process for Model-transparent migration typically completes in two to four minutes, broken down roughly as follows: detection and spare provisioning (seconds to one minute, depending on spare availability), coordinated pause (seconds), process checkpointing (30 to 60 seconds, depending on model size), state transfer (10 to 30 seconds over RDMA, longer over TCP), process restoration (30 to 60 seconds), and NCCL communicator repair and resume (seconds). During this entire window, all workers in the training job are paused. After completion, training resumes at full throughput with zero lost iterations.

Operational Dependencies and Requirements

Spare Node Pool

Migration requires healthy spare nodes to be available as migration targets. Without spares, the orchestrator cannot perform a migration and must fall back to restarting the job from a persisted checkpoint. The Operator should provision spare capacity proportional to the expected failure rate and tolerance for migration latency. Having spares pre-provisioned and idle means migration can begin immediately when a taint is detected. If spares must be provisioned on demand (by the scheduler plugin requesting new containers), there is additional latency while the spare containers start and register. Spare pool sizing is a direct cost trade-off against migration speed and availability.

Memory / Storage Requirements for State Capture

CRIU checkpointing 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 substantially 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, /dev/shm 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 /dev/shm 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.

Network Bandwidth

State transfer speed is directly determined by available network bandwidth between the

leaver and spare nodes. For RDMA-capable networks, transfer of a 40 GB checkpoint completes in approximately one 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 CRIU to capture process state. Specifically, CRIU requires the SYS_PTRACE and SYS_ADMIN capabilities, 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.

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

Failure Type	Affected Component	Blast Radius	Typical Symptoms
Single GPU hard failure (uncorrectable ECC, fallen off bus)	One GPU on one tray	One GPU initially; 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 yields 16 active trays (64 GPUs actively training) and 2 spare trays (8 GPUs held in reserve).

TorchPass In Action: Each Tray/Node is a Kubernetes Pod

In the standard 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 the CRIU process described in Section 3: all workers pause, the affected workers' processes are checkpointed, the checkpoint is transferred to the spare GPU (over the NVLink fabric within the same rack, which is extremely fast since all GPUs share the same NVLink domain), and the process is restored on the spare. Training resumes.

After a tray-level migration, one of the two spare trays is now fully committed to training, leaving only one spare tray (4 GPUs) available for future migrations. A second tray-level failure would exhaust all spare capacity.

TorchPass In Action: Each GPU Is Configured As a Kubernetes Pod

Treating each compute tray as a single Kubernetes pod (or Slurm job step) effectively makes all 4 GPUs on a tray migrate even if only one GPU is problematic. An alternative deployment model treats each GPU as a separate pod. In this configuration, each of the 72 GPUs in the rack hosts its own independent Kubernetes pod, each running a single PyTorch worker process. The 64 active GPUs (on the 16 active trays) each run one training worker pod, and the 8 spare GPUs (on the 2 spare trays) each run one spare pod.

This per-GPU pod granularity dramatically improves fault absorption for individual GPU failures. When a single GPU develops a problem, only that one GPU's pod needs to be migrated. TorchPass pauses all 64 workers, checkpoints the single affected worker, transfers its state to one of the 8 spare GPU pods, and restores it there. After migration, there are still 7 spare GPUs available. This means the rack can absorb up to 8 individual GPU failures before spare capacity is exhausted.

The trade-off of per-GPU pod granularity is slightly higher management overhead: 72 pods per rack instead of 18, and more Kubernetes objects to track. However, in practice this overhead is negligible for modern Kubernetes control planes and for the TorchPass orchestrator, which is designed to handle clusters with tens of thousands of GPUs.