CLOCKWORK.io

Rethinking Collective Communications for Fault-Tolerant, High-Performance AI Training

1. Introduction

Collective communications are the nervous system of distributed AI training. Every time a large language model synchronizes gradients, gathers activations, or routes tokens to experts, it executes a collective operation that touches every GPU in the job. Operations such as AllReduce, AllGather, ReduceScatter and AllToAll consume **30–50% of total training wall-clock time** in distributed workloads, with the fraction rising as GPU compute throughput outpaces network bandwidth improvements 1. Yet the libraries that implement them, NVIDIA's NCCL (Nvidia Collective Communication Library) and AMD's RCCL (Radeon Collective Communications Library), were designed for performance on healthy hardware, not for the three challenges that now define large-scale training: **resilience** against pervasive failures, **adaptability** to shifting cluster conditions, and **optimization** of the communication operations themselves.

The first challenge is failure. At the scale of today's frontier training runs, failure is not a risk, but a daily reality. Meta recorded 419 unexpected interruptions over 54 days during LLaMA 3 pre-training 2; a subsequent reliability study measured mean time to failure at just 1.8 hours for a 16,384-GPU job, projecting 17-18 minutes at 100,000 GPUs 3. Google reports hardware failures "multiple times per hour" at Gemini training scale 4.

© 2026 CLOCKWORK. All rights reserved.

The dominant recovery mechanism is checkpoint-restart (training restart based on restoration from the most recent checkpoint) which loses all computation since the last save across every GPU in the job, costing thousands of dollars per event at production scale. And current collective libraries offer no middle ground: even a single link flap or GPU fault can poison the entire communicator group, killing the job outright 6

The second challenge is placement. Because collective operations are synchronous, every rank must complete the operation before any rank can proceed. Thus, a single slow rank will slow down the entire job making training performance acutely sensitive to where workers are located in the physical topology. Ranks that cross spine hops synchronize slower than ranks within a rack, and ranks split across nodes pay NVLink-to-NIC penalties. Yet placement is a one-shot decision at job launch but conditions can change frequently. When better topologies open up mid-run, when small jobs fragment the resources a larger job needs, or when a failure recovery leaves workers in a suboptimal layout, there is no mechanism to adapt without killing and restarting the job.

The third challenge is performance. NCCL selects algorithms, protocols, and channel counts based on analytical cost models tied to nominal hardware specifications. These models can diverge 10 - 30% from actual performance when conditions differ from the spec sheet caused by factors such as thermal throttling, congestion, and configuration drift. Meanwhile, the collective communication landscape itself is fragmenting. Dense operations like AllReduce remain best served by NCCL's ring and tree implementations, while sparse, dynamic operations like MoE dispatch benefit from GPU-Initiated Networking. No single backend is optimal everywhere, and yet the choice is made statically.

These symptoms: jobs killed by transient link flaps, a straggler causing hours of computation wasted across thousands of healthy GPUs, and performance left on the table by stale configurations, share **three structural root causes.**

1. First, **collective operations enforce synchronous semantics** in that every rank must complete before any rank can proceed, so a single degraded component becomes a system-wide stall

2. Second, the **collective communication layer lacks state visibility**. It cannot observe infrastructure health (which links are degrading, which GPUs are trending toward failure), workload state (which collective each rank has completed, whether quiescing is safe), or its own performance (whether the chosen algorithm is actually optimal for current conditions).

3. Third, there is **no action surface** between "everything is fine" and "kill the job." The libraries provide no mechanism to reroute around a failed link, migrate a rank off a suspect or failed

© 2026 CLOCKWORK. All rights reserved.

GPU, or adjust communication parameters to optimize performance. This is not because these actions are impossible, but because the layer where they would execute has no interface for them.

Addressing these issues - making synchronous operations resilient, making invisible state visible, and creating an action surface beneath running jobs - is the architectural problem this paper addresses. The goal is to not to trade off resilience or adaptability against performance but to optimize them together.

1.1 Thesis

What if the collective communication layer could ***predict*** failures before they happen or catch them before they crash a job, ***adapt*** to them transparently when they do, move workloads to optimal topology in real time, and ***tune*** its own performance based on measured conditions? This paper describes Clockwork's vision for achieving this by building an intelligent collective communication platform that rests on two architectural foundations:

- **Distributed State Tracking (DST)**: a sensing layer providing deep, real-time visibility into collective state, workload state, and infrastructure health through a NCCL interceptor, a custom transport-layer net plugin, and process-level state capture.
- **Dynamic Workload Control (DWC)**: an action layer that translates observed conditions into recovery, placement, or optimization decisions through pre-job GPU auditing, an out-of-band probe mesh (lightweight agents on every node that continuously exchange small test packets to measure network health), runtime anomaly detection, and transparent orchestration beneath running training jobs.

The power of this architecture lies in the intersection of two visibility domains that have historically been siloed. Fleet-level characteristics, such as link health, NIC status, topology, congestion patterns, GPU thermal and error trends, are observable to fabric managers but invisible to training frameworks. **Workload-level characteristics**, which collective operations are executing, their message sizes and parallelism dimensions, which rank has reached which sequence number, and whether quiescing is safe, are observable to training frameworks but invisible to the fabric. Clockwork's DST layer observes both simultaneously, and DWC acts on their intersection. Path failover requires knowing both which link failed (fleet) and which in-flight requests must be retransmitted (workload). Predictive migration requires knowing both which GPU is degrading (fleet) and at which collective sequence number all ranks can safely pause (workload). Communication tuning requires knowing both current congestion conditions (fleet) and the actual message size distribution of the running job (workload).

© 2026 CLOCKWORK. All rights reserved.

No decision in any application domain can be made from fleet state alone or workload state alone — every intelligent intervention requires both.

These foundations enable three progressive application domains. **The first is Fault tolerance (currently shipping)** which facilitates transparent link failover, preemptive live GPU migration, and rapid recovery from unplanned failures without job restarts or code changes. **The second is Workload Rebalancing (roadmap)** facilitating live migration of running ranks for topology consolidation, defragmentation, and post-failure rebalancing, transforming scheduling from a one-shot decision into continuous optimization. **The third is Communication Performance Optimization (roadmap)** facilitating empirical tuning of NCCL algorithm, protocol, and channel-count decisions, plus intelligent backend routing between NCCL and GPU-Initiated Networking based on observed workload characteristics.
Each domain is an application of the DST + DWC architecture. The telemetry built for fault detection also powers placement assessment and communication tuning. The orchestration built for failure recovery also drives scheduled migrations. This is the compounding value of a platform approach where every new capability extends foundations already deployed and hardened in production at the world's largest GPU clusters.

2. Background and Motivation

This section examines the structural gaps that define the current state of collective communications in large-scale AI training. Each gap represents a dimension where existing infrastructure falls short, and where our platform approach can create compounding value.

2.1 Collective Communication Libraries Were Not Designed For Fault Tolerance

NCCL, RCCL, and NVSHMEM assume a stable, healthy fabric beneath them. Their error model is essentially binary: either the collective operation succeeds, or the entire communicator group is poisoned. A critical link flap or GPU failure can hang all workers until they timeout with no graceful recovery path. NCCL's historical error handling was "crash and restart", where the default watchdog timeout of 10 minutes allowed jobs to hang silently before being killed, and CUDA's asynchronous execution model made NCCL timeout errors notoriously difficult to diagnose. Recent NCCL improvements help at the margins but do not solve the fundamental problem. ncclCommShrink enables dynamic communicator resizing without full reinitialization, and the NCCL_SHRINK_ABORT flag can abort hung operations rather than wait for the full timeout. But shrinking a communicator is an emergency measure, not a proactive resilience strategy.

© 2026 CLOCKWORK. All rights reserved.

The library still has no awareness of alternate network paths, no concept of spare resources, no understanding of workload state, and no ability to distinguish between a transient link flap and a permanent hardware failure. The error boundary remains the entire job as there is no concept of partial failure or graceful degradation.

2.2 Failures Are Rampant

At scale, failures are not edge cases; they are the steady state. The data is unambiguous. Meta's reliability study measured mean time to failure (MTTF) of 7.9 hours for 1,024-GPU jobs, dropping to 1.8 hours at 16,384 GPUs and a projected ~17-18 minutes at 100,000 GPUs. GPU hardware issues accounted for the majority of unexpected interruptions during LLaMA 3 pre-training 3. Google's Gemini 2.5 report states that at training scale, hardware failures cause interruptions "multiple times per hour." 4. A critical insight is not just that failures are frequent but that many are preceded by observable signals. Rising ECC error rates, GPU temperature excursions, NVLink bandwidth degradation, increasing collective latencies are all precursors that appear minutes to hours before hard failure.

2.3 Checkpoint-Restart Is Wasteful At Scale

The dominant recovery mechanism today is checkpoint-restart: save model state periodically and, when a failure occurs, kill the job and restart all workers from the last checkpoint. This approach wastes all computation since the last checkpoint across every GPU allocated to the job, not just the one that failed. Further time is also wasted performing the restore and restarting the training.

Production LLM training jobs typically checkpoint every two to four hours 7. A single disruptive failure therefore destroys one to two hours of computation across hundreds or thousands of GPUs, plus 10–30 minutes of recovery time during which the entire allocation sits idle (this can be significantly longer if new resources need to be provisioned). At $3 per GPU-hour, and ~3 disruptive failures per day in a 1,024-GPU cluster, wasted compute accounts for ~$1 - $2 million annually, without even accounting for lost engineering productivity and the opportunity cost of delayed jobs. Worse, as models get larger, the checkpoint overhead itself gets larger, even with aggressive asynchronous checkpoints, given the time it takes to serialize state. Further, duplicating this state in RAM for massive models to allow the model to resume training immediately can lead to Out-of-Memory (OOM) errors.

The industry needs surgical alternatives: migrate only the affected rank, preserve state on healthy workers, and resume rather than kill and restart everything.

© 2026 CLOCKWORK. All rights reserved.

2.4 The Collective Communication Landscape Is Fragmenting

NVIDIA's introduction of GPU-Initiated Networking (GIN) with the GDAKI (GPUDirect Async Kernel-Initiated) backend and symmetric memory models (a programming abstraction where all participating GPUs register identical memory regions at the same virtual address, enabling any GPU to read from or write to any other GPU's region by offset without address translation) has created a new axis of optimization. The GDAKI backend enables GPUs to initiate RDMA transfers directly without CPU proxy involvement, achieving round-trip latencies as low as 16.7μs for small messages 8. This is transformative for sparse, dynamic communication patterns—particularly AllToAll operations used in Mixture-of-Experts architectures, where per-transfer latency compounds across dozens of GPU-to-GPU exchanges per dispatch phase.

The result is that no single communication backend is optimal for all operations. Dense collectives (AllReduce, AllGather, ReduceScatter) remain best served by NCCL's highly optimized ring and tree implementations, which amortize proxy overhead across large bulk transfers. Sparse operations (AllToAll for MoE dispatch and combine, point-to-point for expert routing) benefit from GIN's low-latency, device-initiated transfers. Yet the choice between backends is currently made statically at job launch and applied uniformly to all operations. As model architectures evolve, particularly toward Mixture-of-Experts and dynamic sparsity, the ability to route different collective operations to different backends per-operation becomes a performance imperative.

2.5 Static Configuration In A Dynamic World

NCCL's algorithm selection (ring versus tree), protocol selection (LL, LL128, Simple), and channel count are driven by analytical cost models based on nominal hardware specifications. These models predict execution time as a function of message size, GPU count, and protocol overhead, but they cannot account for real-world conditions. Published measurements show these models can deviate significantly from actual performance, particularly near algorithm transition boundaries (1–10MB messages) and protocol transitions (~16KB, ~64KB). Critically, these parameters are set once at job launch and remain fixed for the job duration. The cluster, however, changes continuously as thermal conditions shift, competing jobs arrive and depart, and links degrade, but the collective configuration does not adapt. The gap between the model's assumptions and the cluster's reality widens over multi-day and multi-week training runs, leaving performance on the table throughout.

2.6 Topology-Dependent Placement in a Shared, Dynamic Cluster

Because collective communications are synchronous, every rank must complete before any rank can proceed, causing training performance to be acutely sensitive to where ranks are

© 2026 CLOCKWORK. All rights reserved.

placed relative to each other in the physical topology. Ranks that cross spine hops synchronize slower than ranks within a rack; ranks split across nodes pay NVLink-to-NIC penalties compared to ranks sharing an NVSwitch (the in-node switch that provides full-bisection NVLink bandwidth between all GPUs on a server). ByteDance's MegaScale system found that communication remains the dominant bottleneck even after aggressive optimization, with topology-sensitive placement directly affecting model FLOPs utilization 9.

Yet in practice, GPU clusters are shared and dynamic. New jobs arrive, old jobs finish, hardware gets tainted, and the optimal placement for any given job shifts over time (all these are exacerbated when jobs run on cloud infrastructure). Today, placement is a static, one-shot decision: the scheduler places ranks at launch and they stay there for the duration. Two scenarios illustrate the cost of this rigidity. First, if a job is placed across racks, every collective pays a spine-hop latency penalty for the entire job duration even if a rack-local topology opens up after the job has started. Second, small fine-tuning jobs occupying 2 - 4 GPUs across multiple 8-GPU nodes can block a large pretraining job that needs full contiguous nodes, with the only option being to wait.

In each case, the only way to "move" a running job is to kill it and resubmit, losing all state since the last checkpoint. The ability to move running workloads without killing them would transform scheduling from a static bin-packing problem into a continuous optimization.

These gaps: brittle collective libraries, pervasive failures, wasteful recovery, a fragmenting communication landscape, static configuration, and rigid placement, are not independent problems. They share a common root cause which is that the collective communication layer lacks the ability to observe what is happening across the infrastructure, and to act on those observations in real time. The next section introduces the architectural foundations that Clockwork has built to close these gaps.

3. Architectural Foundations

Every capability described in this paper - link failover, live GPU migration, workload rebalancing, communication tuning - rests on two architectural pillars. Distributed State Tracking (DST) provides the sensing layer: deep, real-time visibility into collective communication state, workload state, and infrastructure health. Dynamic Workload Control (DWC) provides the action layer: translating observed conditions into concrete recovery, placement, or optimization decisions. This section describes both pillars and the technology enablers that make them possible.

© 2026 CLOCKWORK. All rights reserved.

3.1 Distributed State Tracking (DST): The Sensing Layer

DST gives Clockwork a continuously updated, multi-layer view of the entire training system, from individual RDMA queue pairs up through collective operations and into the training workload itself. **Crucially, it fuses two visibility domains, fleet characteristics** (link health, NIC status, topology, congestion) and **workload characteristics** (collective types, message sizes, parallelism structure, per-rank progress), that are observable only in isolation by existing tools. It comprises several instrumentation points, each providing visibility that the others cannot.

3.1.1 NCCL Interceptor

The NCCL interceptor is a shared library that interposes on every NCCL API call without modifying NCCL's source code or the training application. The interceptor's primary responsibilities are: tracking all NCCL communicators and their membership across ranks; maintaining per-communicator collective operation sequence numbers that provide a globally consistent logical counter (a monotonically increasing counter that establishes a total ordering of events across all ranks within a communicator without relying on synchronized wall-clock time); implementing coordinated quiescing, which pauses all workers at the same collective sequence number to establish a consistent distributed snapshot boundary; and repairing communicators after migration by excluding departed ranks and incorporating joiners.
The interceptor knows exactly which collective each rank has completed, and can coordinate all workers to a common logical point without any application-level hook.

3.1.2 Custom Net Plugin

Clockwork's NCCL net plugin leverages direct access to the RDMA (Remote Direct Memory Access, a network transport that allows direct memory-to-memory transfers between nodes, bypassing the operating system kernel) layer beneath NCCL's collective algorithms. The plugin implements NCCL's versioned net plugin interface (ncclNet_v9), providing the required isend, irecv, and test function pointers using libibverbs for InfiniBand and RoCE fabrics. This interface is inherently non-blocking: isend and irecv may return a NULL request when the operation cannot immediately proceed, signaling NCCL to retry on its next polling cycle rather than block. Clockwork's implementation preserves this contract while adding capabilities that the stock plugin lacks: per-queue-pair failure detection via send timeouts and QP completion errors (QP: the fundamental RDMA connection endpoint pairing a send queue and receive queue between two nodes); a failover state machine that can reset queue pairs, select backup NICs, create new connections, and retransmit in-flight requests; and granular per-flow telemetry at the individual queue-pair level. The non-blocking contract is what makes transparent failover possible.

This transport-layer visibility is essential because many failure modes, such as link flaps, NIC degradation, and transient congestion, are invisible to NCCL's collective-level abstraction.

By the time NCCL detects a problem, typically through its 10-minute watchdog timeout, the damage is done. The Clockwork net plugin detects these conditions in milliseconds and can act before the collective layer is aware anything has gone wrong, without any performance impact during normal operations.

3.1.3 Out-of-band Probe Mesh

Clockwork deploys lightweight user-space agents on every node that continuously exchange small probes over both frontend (Ethernet) and backend (RDMA) networks. These probes measure one-way delays (OWDs) between NIC pairs, enabled by Clockwork's software-based clock synchronization technology that aligns node clocks to near-nanosecond accuracy across tens of thousands of hosts.

The probe mesh provides three capabilities that no in-band instrumentation can match. First, topology inference: measured OWDs under no-load conditions reveal the physical hierarchy of the cluster - which nodes share a leaf switch, which cross a spine hop - without requiring access to switch configurations. Second, connectivity monitoring: if probes stop flowing between a NIC pair for a defined interval, the agent flags a disconnection immediately, even on paths that carry no workload traffic. Third, congestion detection: under load, increased OWDs reveal congestion hotspots at specific switch ports, enabling correlation with job-level performance degradation Because the probe mesh operates out-of-band, independent of any training job, it provides continuous visibility into fabric health even when no workloads are running. This is the foundation for pre-job auditing, post-failure diagnostics, and continuous fleet-wide health maps that inform scheduling decisions.

3.2 Dynamic Workload Control (DWC): The Action Layer

DST produces a continuous stream of signals. DWC consumes those signals and translates them into coordinated, multi-step interventions that execute beneath running training jobs. DWC comprises four capabilities.

3.2.1 Pre-Job GPU Audit

Before a training job launches, the audit system validates every allocated node through a structured and programmable sequence of tests: static dependency checks (driver versions, kernel modules, RDMA configuration); DCGM hardware diagnostics that stress GPUs, memory, and intra-node interconnects to surface latent errors; a NCCL AllReduce benchmark that measures achieved bus bandwidth across all allocated nodes and NICs; and a temporary all-to-all probe mesh that verifies backend connectivity and measures baseline one-way delays. Nodes that fail any stage are tainted and replaced before the job starts. The audit transforms GPU health from a runtime surprise into a pre-flight guarantee - and the same tests serve as the diagnostic suite after failures occur, closing the loop between detection and root cause.

3.2.2 Runtime Anomaly Detection

During training, DWC correlates signals from all four DST instrumentation points. For example, rising per step iteration times correlated with collective latencies correlated with OWD spikes from the probe mesh suggest slowdowns from network congestion. Runtime GPU monitoring can provide indicators of a GPU straggler or a GPU that is degrading toward failure. The goal is not just to detect failures after the fact, but to identify degradation trends early enough to trigger preemptive migration: moving a rank off a suspect GPU before it fails, rather than recovering after.

3.2.3 Process-Level State Capture

DWC has the ability to capture and transfer the complete state of a running GPU training process. This builds on two recent technology enablers. NVIDIA's cuda-checkpoint capability (driver 550, April 2024) can quiesce GPU execution, export GPU memory and context to host memory, and later restore on a different physical GPU via UUID remapping 9. CRIU (Checkpoint/Restore In Userspace) 4.0 (late 2024) added an official CUDA plugin that integrates cuda-checkpoint with Linux process checkpoint/restore, enabling transparent capture of the full process state, including host memory, file descriptors, network connections, GPU memory and CUDA contexts, in a single coordinated operation 10.

The key advantage of process-level state capture is zero steady-state overhead: the training process runs at native speed with no API interception in the critical path. This mechanism is what enables user-transparent migration, i.e. the training code does not need to implement any checkpoint hooks or register any callbacks.

3.2.4 Tiered Migration Orchestration

When DWC determines that a rank must move, the available recovery options depend on whether the triggering event is predictive (the hardware is degrading but still functional) or unplanned (the hardware has already failed).

Planned migrations are triggered when DWC detects degradation precursors such as rising ECC error rates, NVLink bandwidth drops, or escalating collective latencies, while the affected hardware is still operational.

- **Model-aware migration** (~100 seconds on a 109B parameter model) is the fastest: the training code registers checkpoint/restore callbacks via Clockwork's TorchPass library, workers agree on a migration boundary through an orchestrator, and the affected rank transfers model state to the spare.
- **Model-transparent migration** (~3 minutes) is slower but requires zero application integration: the NCCL interceptor coordinates all workers to pause at the same collective sequence number, establishing a globally consistent state, and the entire process, including

including host memory, GPU memory, CUDA contexts, and file descriptors, is captured and restored on a replacement GPU transparently.

Unplanned-failure migration deals with unplanned failures such as a GPU falling off the bus or a full node crash eliminating the failed rank before any coordinated quiescing can occur. Recovery depends on the most recent available state. Clockwork's TorchPass coordinator initiates a "Just-in-time" checkpoint on healthy data parallel replicas immediately upon a failure, transfers the checkpoint from the healthy replica onto the designated spare, restarts the process group and continues training.

Across both scenarios, the orchestrator manages spare resource allocation and joiner provisioning, and communicators are repaired to include joiners and exclude leavers, all beneath the running job. A future release will allow resumption with a reduced world size when no spares are available. The same orchestration machinery that drives fault-tolerance migrations also provides the foundation for the scheduled migrations described in the sections below.

3.2.5 Transparent Infrastructure Injection

A critical design principle is that Clockwork operates as platform software-driven infrastructure, not as an application feature. Scheduler plugins that support Kubeflow and Slurm currently (more planned over time) enable TorchPass functionality without the need to modify job specifications or training code. Operations teams enable Clockwork cluster-wide; ML teams benefit without any coordination and with minimal to no code changes. This transparency is what allows the platform to be deployed across heterogeneous workloads from diverse teams - the same infrastructure that protects a TorchTitan pretraining run also protects a Megatron-LM fine-tuning job.

4. Application Domains

The architectural foundations described in the previous section, Distributed State Tracking and Dynamic Workload Control, are general-purpose capabilities. Their value is realized through the specific application domains they enable. This section describes three such domains, ordered by maturity: fault tolerance (shipping in production), workload rebalancing (planned), and collective communication performance optimization (planned). Each domain applies the same sensing and orchestration infrastructure to a different class of problem, and each builds incrementally on what the previous domain has proven.

The progression is deliberate. Fault tolerance came first because the need was most urgent and the value most immediate. Failures at scale are a daily reality that costs organizations hundreds of thousands to millions of dollars annually. Workload rebalancing follows naturally since the

same migration machinery that moves a rank away from a failing GPU can move a rank toward a better one, transforming scheduling from a static, one-shot decision into a continuous optimization. Communication performance optimization completes the picture. The same per-collective telemetry that detects anomalies for fault tolerance and assesses topology for placement also reveals where NCCL's static configuration choices are leaving performance on the table.

Importantly, each successive domain is not built in isolation but leverages common underlying building blocks. Workload rebalancing requires no new instrumentation; only new triggering logic atop proven migration mechanisms. Communication optimization only requires a tuner plugin that consumes telemetry the platform already collects. This compounding return on shared foundations is the central architectural advantage of a platform approach over point solutions.

5. Fault Tolerance (Shipping Today)

Fault tolerance is Clockwork's first and most mature application domain. Network fault tolerance via path failover shipped in 2025; workload fault tolerance via live GPU migration followed in early 2026. This section describes the two complementary capabilities and how they work together.

5.1 Network Fault Tolerance: Path Failover

Link failures including flapping optics, transient bit errors and NIC degradation, are among the most common causes of training disruptions. Without intervention, a single link failure between the GPU and a Leaf switch can hang all workers as NCCL waits for data that will never arrive, until it times out. Clockwork's net plugin transforms this from a job-killing event into a transient throughput dip.

The failover mechanism operates entirely within the NCCL net plugin, at the RDMA transport layer beneath NCCL's collective algorithms. A dedicated FailureHandler thread continuously monitors each communicator for failure signals, and upon such a signal firing, the FailureHandler executes a coordinated state machine between the local and remote communicator endpoints. The receiver resets all queue pairs, drains completion queues, identifies healthy backup NICs, and creates new QPs on those backups. The sender then connects to the new QPs and both sides reconcile in-flight requests, retransmitting only those that were incomplete on both ends, before resuming normal operation.

The result is that training continues with a slight throughput reduction due to fewer available network paths (typically unnoticeable unless bandwidth is already close to saturation which is mitigatable when paired with Live GPU migration - see below), and recovers automatically to full performance as soon as the original link heals. No job restart, no checkpoint restore, no lost

computation. When the FailureHandler holds the mutex during the failover state machine, the NCCL thread's try-lock fails and the operation returns immediately without making progress, rather than blocking. This works because NCCL's net plugin interface is inherently non-blocking. NCCL's existing retry behavior therefore absorbs the failover pause naturally, and from PyTorch's perspective the collective simply takes slightly longer to complete during failover.

5.2 Workload Fault Tolerance: Live GPU Migration

Path failover handles network-layer disruptions. Live GPU migration handles everything else: GPUs falling off the bus, uncorrectable memory errors, software and driver faults, thermal shutdowns, and full node failures. When one or more components fail or show signs of impending failure, TorchPass migrates the affected ranks to spare resources: dedicated GPUs held in reserve, or floating spares drawn dynamically from lower-priority jobs or available cluster capacity.

Planned migration. When DWC detects degradation precursors while the affected hardware is still operational, two approaches to planned migration exist.

- Model-aware teleporting (~100 seconds) is the fastest: workers agree on a migration boundary through the orchestrator, the affected rank exports model state via registered callbacks, a joiner loads that state on spare hardware, and processes are restarted to establish new communicators and resume training.
- Model-transparent migration (~3 minutes) requires zero application integration: the NCCL interceptor coordinates all workers to pause at the same collective sequence number, establishing a globally consistent state, and the entire process is captured and restored on the replacement GPU transparently.
- Note: In both modes, healthy workers keep their GPU memory intact, and only the affected rank transfers state. Training resumes at exactly the next iteration with no lost computation.

Unplanned-failure migration. When a GPU or node fails without warning, the failed process no longer exists to checkpoint, so Model-aware migration and Model-transparent migration cannot be used. Instead, recovery proceeds through JIT checkpointing: all surviving workers are signaled to export their current training state. All workers are then restarted, and on startup each worker checks for a complete, consistent JIT checkpoint. If one exists, workers resume from it, losing only the computation between the last completed step boundary and the failure. If, in a worst-case scenario, the JIT checkpoint is incomplete or inconsistent, workers fall back to the last persisted checkpoint. In either case, the orchestrator provisions replacement resources and the restarted job resumes training. JIT-C dramatically reduces lost computation compared to conventional checkpoint-restart by capturing state at the moment of failure detection rather than relying on periodic checkpoints that may be hours old.

5.3 Layered Defense: Network and Workload Fault Tolerance Together

The two fault tolerance capabilities form a layered defense. When a network disruption occurs, the net plugin's path failover provides immediate, sub-second protection, and training continues on backup paths with a modest throughput reduction. If the fault self-heals (as approximately 85% of link flaps do), performance recovers automatically. If the fault persists, the sustained operation on backup paths gives the orchestrator time to prepare and execute a planned live GPU migration to fully healthy resources. A hard network failure is converted from a job-killing event into at most a temporary slowdown followed by graceful relocation.

5.4 Fault Tolerance Summary

For GPU and node failures, the response depends on whether precursors were detected. Predictive failures trigger preemptive migration with zero lost computation. Unplanned failures trigger recovery from a Just-in-time Checkpoint. In all cases, the training application requires minimal to no code changes, no special error handling, and no awareness that a failure occurred - Clockwork operates as transparent infrastructure beneath the running job.

This currently shipping technology is entirely software-based, runs on standard Ethernet (RoCE) or InfiniBand networks, supports both NVIDIA (NCCL) and AMD (RCCL) GPUs, and integrates with popular training frameworks including TorchTitan, Megatron-LM, and DeepSpeed on both Slurm and Kubernetes schedulers.

5.5 Preliminary Evaluation of Fault Tolerance Performance

This section presents results from a controlled fault-injection benchmark comparing TorchPass against standard checkpoint-restart and Meta's TorchFT, supplemented by observations from production deployments at clusters with tens of thousands of GPUs.

5.5.1 End-to-End Fault Tolerance Benchmark

We trained a Llama 4 MoE Scout 109B model using TorchTitan on 64 NVIDIA H200 GPUs (8 nodes × 8 GPUs) on Google Cloud Platform with parallelism configuration PP4, DP2, FSDP4, TP2, EP4, ETP2. Training ran for 3,000 steps with asynchronous checkpointing every 100 steps. GPU faults were injected at random intervals between 2,700 and 4,500 seconds using a fixed seed, until training was completed with a cap of 12 fault injections. Four configurations were tested: a fault-free baseline; standard checkpoint-restart; TorchPass with Model-aware live GPU migration; and TorchFT with replica-group isolation.

Analysis

TorchPass completed training in 405 minutes, six minutes faster than the fault-free baseline, because the total migration overhead of approximately 10 minutes across six events is less than the cumulative stochastic variance over a 3,000-step run. Individual migrations ranged from 1 minute 37 seconds to 1 minute 46 seconds, a 9-second spread reflecting the deterministic nature of the recovery.
Checkpoint-restart took 817 minutes (2X of baseline). Each event incurred 20 - 26 minutes of recovery (checkpoint restore, job restart) plus 28 - 120 rolled-back steps, depending on where in the 100-step checkpoint interval the failure fell. Checkpoint-restart's overhead is structural: it discards all computation since the last save across every GPU, not just the one that failed.

TorchFT took 930 minutes (2.3X of baseline), slower even than checkpoint-restart. The bottleneck was not recovery per se but steady-state per-step overhead from Gloo-based cross-replica gradient synchronization. TorchFT uses Gloo (CPU-based TCP sockets) rather than NCCL (GPU-direct RDMA) for inter-replica all-reduce operations to avoid a deadlock during replica-group teardown and rebuild. This overhead compounds across every training step, not just during failures.

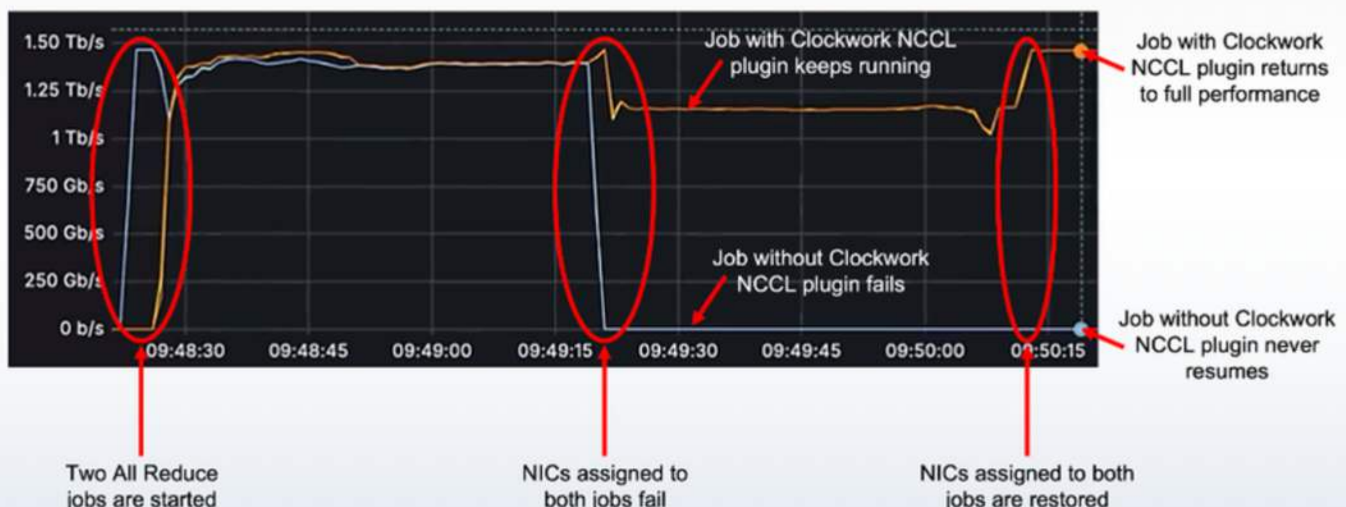
Correctness

We validated three success criteria: the post-recovery training loss tracked the baseline within normal stochastic variation (no divergence); all 3,000 steps completed without manual intervention or timeout-triggered restarts (no hangs); and the final model checkpoint was

consistent with the expected state at step 3,000 (identical final state). For TorchPass, the first post-migration gradient synchronization was verified to produce results consistent with the pre-migration trajectory.

5.5.2 Component-Level Observations

- **Path failover.** Network fault tolerance via the NCCL net plugin has been deployed across tens of thousands of GPUs since September 2025. In controlled testing with a NIC failure on an 8-node cluster, the job with Clockwork's net plugin continued running while the job without it failed immediately and never resumed. Training continued on backup paths with reduced throughput; full performance recovered automatically when the original NIC was restored. The throughput reduction observed in this scenario was approximately 15%, though the actual impact depends on backup path availability and traffic distribution.
- **Migration timing.** TorchPass provides a spectrum of migration mechanisms with different speed/integration tradeoffs: Model-aware migration (~100 seconds, requires checkpoint hooks), Model-transparent migration (~3 minutes, zero integration), Unplanned-failure migration (2 - 5 minutes, basic integration), and persisted checkpoint fallback (5–30 minutes). The 1m 37s - 1m 46s times in the benchmark are consistent with expectations for a 109B MoE model. The narrow 9-second spread across six events confirms that migration time is dominated by deterministic costs (process restart, checkpoint loading).
- **No Steady-state overhead.** The NCCL interceptor maintains per-communicator sequence counters and membership metadata but does not interpose on the data path. The net plugin adds per-flow telemetry using existing completion queue polling. Process-level state capture (CRIU + cuda-checkpoint) imposes zero overhead until triggered: training runs at native speed with no API interception in the critical path. Public research on CRIUgpu corroborates this property, measuring no performance degradation during normal training of Llama 3.1 8B and GPT-2 1.5B workloads on H100 GPUs.



6. Future Application Domains (Roadmap)

6.1 Workload Rebalancing (Roadmap)

The migration machinery described in Section 4 was built for fault tolerance, but its capabilities extend well beyond failure recovery. The same NCCL interceptor, coordinated quiescing, CRIU/cuda-checkpoint integration, and communicator repair that move a rank away from a failing GPU can also move a rank toward a better GPU: one with a more favorable network topology, one not being throttled due to overheating, or one with lower resource contention. This section describes workload rebalancing: the application of live migration to scheduling optimization, independent of any failure.

6.1.1 Static Placement in a Dynamic World

Today, GPU schedulers treat placement as a one-shot decision at job launch. Ranks are assigned to physical GPUs, and they stay there for the duration, whether it is hours or weeks. But the cluster changes continuously: jobs arrive and depart, hardware gets tainted and repaired, and the optimal topology for any given workload shifts over time. All of this is exacerbated when deployed in the cloud. Three scenarios illustrate the cost of this rigidity.

- Topology mismatch. A job placed across racks when a rack-local topology opens up. Every collective in the job pays a spine-hop latency penalty for the entire remaining duration of the run because there is no mechanism to relocate running ranks to the better topology.
- Resource fragmentation. Small fine-tuning jobs occupying 2 - 4 GPUs across multiple 8-GPU nodes might prevent a large pretraining job from acquiring the contiguous nodes it needs. Today the only option is to wait for the small jobs to finish or to kill and resubmit them, losing their state.
- Post-failure suboptimality. After a node is tainted and a worker is migrated to a spare, the resulting topology may be suboptimal. The spare might be in a different rack, introducing asymmetric latency across the job's collective operations. There is currently no mechanism to re-optimize placement once the immediate failure is handled.

In each case, the only way to "move" a running job today is to kill it and resubmit, losing all state since the last checkpoint. The cost is measured in hours of wasted computation across hundreds or thousands of GPUs.

6.1.2 Live Migration as a Scheduling Primitive

TorchPass's migration machinery transforms this equation. The scheduler plugin, which already integrates with Kubernetes and Slurm for fault-triggered migrations, can be leveraged into being an active orchestrator. When the scheduler determines that a better placement exists, it communicates this to TorchPass through the standard tainting mechanism, and TorchPass

coordinates the migration transparently using the same mechanisms described in previous sections: Model-aware migration for fast moves (~30 seconds), Model-transparent migration for application-transparent moves (~3 minutes), or Unplanned-failure migration with full restart for maximum compatibility.

This enables three new scheduling capabilities. **Topology-aware compaction** moves ranks from cross-rack placements to rack-local placements as contiguous nodes become available, reducing collective latency for the moved job without disrupting other running workloads. **Defragmentation** consolidates small jobs onto fewer nodes, freeing contiguous GPU blocks for large pretraining jobs. **Post-recovery rebalancing** re-optimizes topology after a fault-tolerance migration, moving ranks from suboptimal spare locations to better placements as healthy resources become available. In every case, the migration is invisible to the training code.

6.1.3 Toward Continuous Topology Optimization

The long-term vision is a closed-loop system. Fleet telemetry from the probe mesh reveals which jobs are suffering from suboptimal placement based on collective latencies that correlate with cross-spine or cross-rack traffic patterns. The scheduler identifies better placements as cluster topology evolves.

TorchPass executes the migrations transparently, and telemetry confirms the improvement. The scheduler's role shifts from static bin-packing at launch to continuous optimization of a living, breathing cluster.

This capability is on our roadmap: the migration machinery is already proven and shipping for fault tolerance. The scheduling application, the logic that determines when and where to move workloads for placement optimization, is the next frontier.

6.2 Collective Communication Performance Optimization (Roadmap)

Collective communication operations consume 30–50% of total training wall-clock time in communication-bound distributed workloads 9. Yet the libraries that execute these operations make all their critical runtime decisions - algorithm selection, protocol choice, channel count, backend routing - based on static models and fixed configurations. This section describes how Clockwork's telemetry infrastructure enables a fundamentally different approach: empirical, adaptive optimization of collective performance using the same observability platform that drives fault tolerance and workload rebalancing.

6.2.1 Backend Routing: Choosing Between NCCL and GIN

As model architectures evolve toward Mixture-of-Experts (MoE) and dynamic sparsity, the

coordinates the migration transparently using the same mechanisms described in previous sections: Model-aware migration for fast moves (~30 seconds), Model-transparent migration for application-transparent moves (~3 minutes), or Unplanned-failure migration with full restart for maximum compatibility.

This enables three new scheduling capabilities. **Topology-aware compaction** moves ranks from cross-rack placements to rack-local placements as contiguous nodes become available, reducing collective latency for the moved job without disrupting other running workloads. **Defragmentation** consolidates small jobs onto fewer nodes, freeing contiguous GPU blocks for large pretraining jobs. **Post-recovery rebalancing** re-optimizes topology after a fault-tolerance migration, moving ranks from suboptimal spare locations to better placements as healthy resources become available. In every case, the migration is invisible to the training code.

6.1.3 Toward Continuous Topology Optimization

The long-term vision is a closed-loop system. Fleet telemetry from the probe mesh reveals which jobs are suffering from suboptimal placement based on collective latencies that correlate with cross-spine or cross-rack traffic patterns. The scheduler identifies better placements as cluster topology evolves.

TorchPass executes the migrations transparently, and telemetry confirms the improvement. The scheduler's role shifts from static bin-packing at launch to continuous optimization of a living, breathing cluster.

This capability is on our roadmap: the migration machinery is already proven and shipping for fault tolerance. The scheduling application, the logic that determines when and where to move workloads for placement optimization, is the next frontier.

6.2 Collective Communication Performance Optimization (Roadmap)

Collective communication operations consume 30–50% of total training wall-clock time in communication-bound distributed workloads 9. Yet the libraries that execute these operations make all their critical runtime decisions - algorithm selection, protocol choice, channel count, backend routing - based on static models and fixed configurations. This section describes how Clockwork's telemetry infrastructure enables a fundamentally different approach: empirical, adaptive optimization of collective performance using the same observability platform that drives fault tolerance and workload rebalancing.

6.2.1 Backend Routing: Choosing Between NCCL and GIN

As model architectures evolve toward Mixture-of-Experts (MoE) and dynamic sparsity, the

the communication mix within a single training iteration increasingly spans two fundamentally different regimes. Dense collectives (AllReduce, AllGather, ReduceScatter) are symmetric, predictable, and bulk-optimizable—ideally served by NCCL's highly tuned ring and tree implementations. Sparse operations (AllToAll for MoE dispatch and combine) are dynamic, non-uniform, and latency-sensitive—better served by GPU-Initiated Networking (GIN) with the GDAKI backend, which eliminates CPU proxy involvement and achieves round-trip latencies as low as 16.7μs for small messages 8.

Our plan would be to create a Clockwork ProcessGroup wrapper that registers as a PyTorch backend and routes each collective operation to the optimal underlying implementation based on operation type. Dense collectives route to NCCL; sparse operations route to GIN with GDAKI. The routing decision is transparent to the training code, as the application initializes with the Clockwork backend, and the wrapper handles all dispatch decisions internally. No training code changes are required beyond specifying the backend name at initialization.

6.2.2 Empirical Tuning Of NCCL Internals

For operations routed to NCCL, our plan is to create a Clockwork NCCL Tuner Plugin that replaces NCCL's analytical cost model with empirical measurements. NCCL selects algorithms (Ring, Tree, CollNet for network-offloaded reductions via SHARP-capable switches, NVLS for NVLink SHARP intra-node reductions), protocols (LL, LL128 for low-latency small messages, Simple for high-bandwidth bulk transfers), and channel counts using a linear performance model based on nominal hardware specifications. This model predicts well for common cases but can deviate meaningfully from actual performance when hardware characteristics differ from reference assumptions, when workload patterns cluster near algorithm transition boundaries (1–10MB messages), or when thermal throttling, network congestion, or memory contention alter achieved bandwidth.

The tuner operates within NCCL through the standard tuner plugin interface. When NCCL invokes the getCollInfo callback, the tuner consults an empirical cost table built from CUDA event timing of actual collective executions. During periodic exploration iterations (typically 5–10% of training steps), the tuner profiles non-default algorithm-protocol-channel combinations, allocating profiling effort proportionally to expected optimization impact, concentrating on collectives that consume the most wall-clock time and have the highest uncertainty. Once sufficient telemetry exists, the tuner's empirical cost estimates replace NCCL's analytical predictions, correcting algorithm selection near transition boundaries and optimizing channel counts where NCCL's conservative heuristic leaves bandwidth underutilized. In essence, performance is optimized to actual rather than theoretical conditions.

6.2.3 The Telemetry Flywheel

The key connective insight is that these optimizations are not standalone features, but are powered by the same telemetry infrastructure that drives fault tolerance and workload

rebalancing. The net plugin that detects link failures for path failover also measures per-flow throughput and latency. The probe mesh that monitors fabric health for failure prediction also reveals congestion patterns that inform backend routing decisions. The NCCL tuner's per-collective measurements feed back into the same observability layer that the orchestrator uses to assess job health.

7. Vision: Autonomous Collective Communications

The three application domains described in this paper, fault tolerance, workload rebalancing, and communication optimization, are not separate products. They are expressions of the same observe-decide-act control loop, running continuously and transparently beneath the training framework. The end state is autonomous collective communications: a system that continuously monitors its own health and performance, predicts failures before they occur, adapts topology and routing in real time, and optimizes communication parameters based on observed conditions.

This vision is grounded in infrastructure that already exists. Distributed State Tracking already provides real-time visibility from the RDMA queue-pair level up through collective operations and into the training workload. Dynamic Workload Control already translates those signals into coordinated, multi-step interventions beneath running jobs.

Collective communications are the nervous system of distributed AI training. Today, that nervous system is static, fragile, and blind. Clockwork is making that nervous system intelligent: able to sense what is happening across the infrastructure, reason about what should change, and act to optimize the workloads it supports. The result is AI training that doesn't just run fast when everything is working, but is optimized to run fast all the time.

8. References

1. Jiang et al. “MegaScale.” USENIX NSDI 2024.
2. Llama Team, AI @ Meta. “The Llama 3 Herd of Models.” arXiv:2407.21783, 2024.
3. Kokolis et al. “Revisiting Reliability in Large-Scale Machine Learning Research Clusters.” IEEE HPCA 2025.
4. Gemini Team, Google. “Gemini 2.5 Technical Report.” 2025.
5. Qian et al. “Alibaba HPN.” ACM SIGCOMM 2024.
6. Jang et al. “Oobleck.” ACM SOSP 2023.
7. Gupta et al. “Just-In-Time Checkpointing.” ACM EuroSys 2024.
8. NVIDIA. “GPU-Initiated Networking for NCCL.” arXiv:2511.15076, November 2025.
9. NVIDIA. cuda-checkpoint API: cuCheckpointProcessLock, cuCheckpointProcessCheckpoint, cuCheckpointProcessRestore, cuCheckpointProcessUnlock. Released with NVIDIA driver 550, April 2024. Supports GPU UUID remapping for cross-GPU restore.
10. CRIU 4.0 (late 2024) introduced the official CUDA plugin for transparent checkpoint/restore of GPU-accelerated containers. CRIUgpu (February 2025) validated transparent checkpointing of LLM training workloads on H100, A100, and AMD MI210 GPUs