Characterizing and Mitigating Productivity Loss in Large-Scale Model Training: An Empirical Study

Dinghao Xue[◊]
Beihang University
Beijing, China
dinghaoxue@buaa.edu.cn

Yanxiang Chen[◊]
Beihang University
Beijing, China
22371105@buaa.edu.cn

Junhong Liu[◊]
Beihang University
Beijing, China
junhongliu@buaa.edu.cn

Yepeng Zhang
Kuaishou Technology
Beijing, China
zhangyepeng@kuaishou.com

Guangsen Ni
Kuaishou Technology
Beijing, China
niguangsen@gmail.com

Menghao Zhang
Beihang University
Beijing, China
zhangmenghao0503@gmail.com

Tianyu Wo
Beihang University
Beijing, China
woty@buaa.edu.cn

Zheng Zheng
Beihang University
Beijing, China
zhengz@buaa.edu.cn

Chunming Hu
Beihang University
Beijing, China
hucm@buaa.edu.cn

Jin Ouyang
Unaffiliated
Beijing, China
oyjmical@gmail.com

Renyu Yang[✉]
Beihang University
Beijing, China
renyuyang@buaa.edu.cn

Abstract

Training large-scale foundation models often uses thousands of GPUs for weeks or months. In production clusters, training productivity is reduced not only by explicit failures but also by scheduling delays, repeated recoveries, alive-but-unproductive hangs, and performance degradation during seemingly normal execution. However, existing metrics such as Effective Training Time Ratio (ETTR) mainly account for time lost to failures and restarts, and do not directly capture jobs that remain alive but make slow progress. Prior diagnosis and mitigation efforts typically target specific failure modes, leading to case-by-case fixes and offering limited support for cluster-wide attribution of productivity loss or systematic reliability improvements. In this paper, we present the Effective Training Productivity Ratio (ETPR), a reliability-oriented decomposition of end-to-end training productivity to stabilize the large-scale machine learning (ML) systems. ETPR characterizes productivity loss by accounting for pre-execution loss, unproductive allocated compute time, and reduced execution quality. We conduct an empirical study of thousands of training jobs over three months on Kuaishou's production cluster. Our analysis shows that productivity is mainly degraded by diagnosis and recovery for fail-stop and fail-hang incidents, as well as subtle fail-slow anomalies that delay fault localization and worsen overall efficiency. Based on these findings, we build and deploy an automated stability governance framework that integrates: (i) a multi-stage log-based fail-stop diagnosis pipeline, (ii) temporal stack invariance analysis for fail-hang localization, (iii) an adaptive fail-slow detector with straggler detection, and (iv) hierarchical recovery strategies. Experiments show that this framework accelerates failure diagnosis, accurately localizes anomalous ranks, mitigates silent slowdowns, and improves cluster-wide ETPR in production.

CCS Concepts

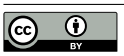
• **Software and its engineering** → **Software defect analysis**; • **Computer systems organization** → *Reliability*.

Keywords

Software Reliability, Model Training, Failure Diagnosis, Failure Recovery

ACM Reference Format:
Dinghao Xue, Yanxiang Chen, Junhong Liu, Yepeng Zhang, Guangsen Ni, Menghao Zhang, Tianyu Wo, Zheng Zheng, Chunming Hu, Jin Ouyang, Renyu Yang. 2026. Characterizing and Mitigating Productivity Loss in Large-Scale Model Training: An Empirical Study. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834462

◊Work done during an internship at Kuaishou Technology.
Corresponding Author: Renyu Yang (renyuyang@buaa.edu.cn).



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834462

1 Introduction

Large foundation models like LLMs and VLMs now often require thousands of GPUs running for weeks or months [1, 3, 14, 21, 23]. Training productivity measures how effectively end-to-end job time and requested GPU resources translate into durable, useful

training progress (i.e., efficiently converting GPU hours into correct and checkpointable model updates). At this scale, productivity depends not only on algorithms and peak hardware performance, but also heavily on training system reliability. In production clusters, hardware failures, network degradation, and software faults [10, 13, 18, 42, 48] are inevitable and reduce productivity by interrupting training, degrading throughput, and delaying diagnosis and recovery. Understanding and managing this reliability-driven productivity loss is essential yet challenging for SRE teams.

In large-scale training, productivity loss can arise at any stage of a job's lifecycle – scheduling, initialization, or training iterations – and can manifest as fail-stop crashes, fail-hang stalls where jobs run but make no progress, or fail-slow degradations that silently slow training [9, 16, 24, 36]. The main challenge is not just debugging individual incidents, but mapping issues across lifecycle stages and types, measuring their impact, and creating system-level diagnosis and mitigation to improve overall productivity. Prior work diagnoses and mitigates specific failure modes or anomalies [17, 18, 20, 40], but does not offer a unified view of productivity loss across cluster training jobs. Existing reliability metrics, such as Effective Training Time Ratio (ETTR), focus on job survivability [22] and do not reveal slowdowns in training progress during execution, making them insufficient for characterizing cluster-wide productivity loss or guiding system-level diagnosis and mitigations.

To address this gap, we introduce the Effective Training Productivity Ratio (ETPR) as an operational metric for characterizing training productivity. ETPR decomposes end-to-end productivity into three components: Scheduling Goodput (SG), Duty Cycle (DC), and Quality of Productivity (QoP). This factorization enables SRE operators to attribute productivity losses to specific stages of the training pipeline, including pre-execution scheduling delay, allocated-but-unproductive runtime, and degraded execution quality. Rather than serving as a standalone optimization objective, ETPR reveals latent inefficiencies and systematically links them to underlying operational issues, enabling more comprehensive diagnosis and governance.

Guided by ETPR and its sub-metrics, we analyze three months of execution logs from one of Kuaishou's production training clusters, covering 20,629 jobs. We find that productivity loss is mainly due to DC and QoP issues. DC loss stems largely from fail-stop and fail-hang behaviors, with most unproductive time spent on fault diagnosis and recovery. Frequent explicit software and user errors cause significant cumulative overhead from repeated triage, while hardware and network faults require even longer diagnosis and repair due to their complexity. QoP loss is driven primarily by fail-slow anomalies such as communication slowdowns, hardware performance variability, and other infrastructure inefficiencies. While less frequent, these anomalies severely reduce end-to-end training productivity because they are hard to localize and fix, and cause long delays.

We design and deploy a stability framework that offers an end-to-end pipeline for anomaly observation, diagnosis, localization, and recovery. It addresses the three main sources of productivity loss identified in our study: diagnosing fail-stop events from execution evidence, detecting and localizing fail-hang stalls that block training progress, and identifying persistent fail-slow anomalies that degrade training throughput. Diagnosis results are mapped to remediation strategies—rollback, in-place recovery, or pod replacement—based on inferred fault category and severity. Case studies and component-level evaluations show that the framework accelerates fail-stop diagnosis, localizes suspicious hang and slow ranks, and enables targeted recovery for hardware-induced failures, improving ETPR by reducing DC and QoP loss.

The main contributions of this paper are as follows.

- **Operational Productivity Decomposition:** We introduce ETPR, a practical metric that decomposes end-to-end training productivity into Scheduling Goodput, Duty Cycle, and Quality of Productivity, enabling reliability-oriented attribution of productivity loss across the training lifecycle.
- **Production Empirical Study:** Applying ETPR to three months of Kuaishou production training data shows that the dominant losses arise after resource allocation, through allocated-but-unproductive runtime and degraded productive execution.
- **Stability Enhancement Framework:** Based on these findings, we build an automated framework to diagnose, localize, and mitigate major sources of productivity loss. We deploy it to improve cluster-level ETPR.

2 Background

This section introduces the lifecycle of large-scale distributed training, and discusses why existing reliability or productivity metrics are insufficient for reliability-oriented productivity attribution.

2.1 Large-Scale Distributed Training

Modern AI models, such as large language models and generative recommendation models, have scaled rapidly, necessitating extensive GPU clusters operating over prolonged periods. For example, pretraining the 405-billion-parameter LLaMA 3 model required more than 16,000 accelerators running continuously for 54 days [14].

Executing such training workloads depends on a complex, multi-layered software and hardware stack. This stack encompasses distributed training frameworks such as PyTorch, Megatron-LM, and DeepSpeed [31, 34, 35]; GPU software components such as CUDA and NCCL [19]; high-speed interconnects and communication fabrics, including RDMA and InfiniBand [25]; underlying hardware resources such as CPUs, GPUs, main memory, and storage; and cluster- and job-level resource managers such as Kubernetes and SLURM [5, 45]. In practice, a large-scale training job typically comprises three distinct phases:

- **Scheduling Phase:** A training job is initially submitted to the cluster scheduler, which verifies the availability of a sufficient number of idle GPUs. Subsequently, the scheduler allocates the job to compute nodes based on current resource availability and topology-aware placement policies. The required containerized runtime environment is instantiated on each assigned node, with inter-container networking across all participating nodes configured. In our cluster, we treat scheduling mainly as the queueing and resource-acquisition stage before GPU resources are granted, during which the job does not physically occupy GPUs.

- **Initialization Phase:** Once launched, the job performs setup before computation, including parsing arguments, verifying resource paths, initializing distributed process groups, and constructing the model and data pipelines.
- **Iteration Phase:** After initialization, training proceeds through repeated iterations or steps, each of which may consist of multiple micro-batches under pipeline parallelism or gradient accumulation. During each iteration, GPUs distributed across many physical nodes perform forward and backward computation, exchange gradients and intermediate activations, and synchronize model states through collective communication primitives. Modern large-scale training commonly combines data, tensor, pipeline, and expert parallelism to coordinate computation and communication across thousands of GPUs. In addition, jobs periodically save checkpoints [37] to memory and persistent storage so that training can resume after failures, preemptions, or manual restarts.

2.2 Existing Reliability and Productivity Metrics

Large-scale training systems use several metrics for reliability and productivity. Reliability metrics like Effective Training Time Ratio (ETTR) [22] quantify failure-related loss from interruptions, preemptions, and recovery, capturing time lost to explicit disruption. However, they do not identify and characterize those jobs that experience performance slowdowns. Goodput metrics, such as ML Productivity Goodput (MPG) [39], instead quantify realized machine learning productivity at fleet scale by decomposing productivity across system layers. MPG is primarily designated to identify and characterize performance- and utilization-related bottlenecks – e.g., those arising from compiler efficiency and heterogeneous resource scheduling – rather than to capture productivity losses attributable to reliability issues.

Inspired by MPG, but with a different objective, we aim to attribute training productivity degradation to the failure manifestations across training stages. We introduce the Effective Training Productivity Ratio (ETPR), a reliability-oriented attribution framework that decomposes end-to-end training productivity into scheduling loss, allocated-but-unproductive runtime, and degraded productive execution, and systematically associates these losses with actions that tackles fail-stop, fail-hang, and fail-slow issues.

3 Methodology

This section describes how we define ETPR and measure training productivity in the production cluster, what data sources were collected for the empirical study, and how failure cases were curated for later analysis.

3.1 Definition of ETPR

To measure the end-to-end training productivity in large-scale distributed training, we introduce the Effective Training Productivity Ratio (ETPR), which provides an operational decomposition of the training process from job submission and resource allocation to training execution quality.

To formally define ETPR, we adopt the training job as the fundamental unit of analysis. As shown in Figure 1, for each training job

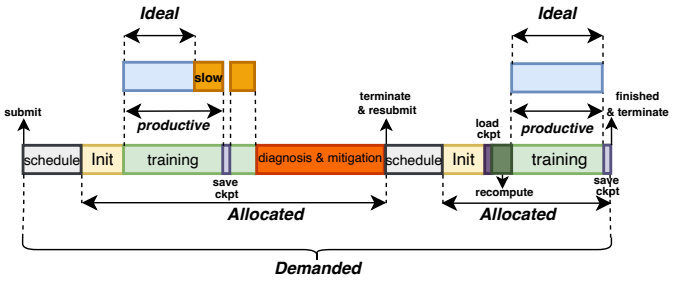


Figure 1: Overview of a training job's lifecycle.

i, we decompose its lifecycle, covering both normal execution and failure-recovery periods, into four temporal quantities:

- **Demanded Time** is the end-to-end time of the job from submission to termination.
- **Allocated Time** is the time during which the job occupies assigned GPUs after successful scheduling.
- **Productive Time** is the allocated time spent on training iterations whose progress is eventually preserved by valid checkpoints, excluding recomputation after recovery.
- **Ideal Time** is the estimated time required to complete the same durable iterations under a healthy execution environment without slowdown.

We further describe how these time quantities are measured in § 3.2. Based on these quantities, ETPR is computed as a cluster-level decomposition over all aggregated training jobs in the observation window:

$$\text{ETPR} = \underbrace{\frac{\sum_i \text{Allocated}_i}{\sum_i \text{Demanded}_i}}_{\text{Scheduling Goodput}} \times \underbrace{\frac{\sum_i \text{Productive}_i}{\sum_i \text{Allocated}_i}}_{\text{Duty Cycle}} \times \underbrace{\frac{\sum_i \text{Ideal}_i}{\sum_i \text{Productive}_i}}_{\text{Quality of Productivity}} \quad (1)$$

Scheduling Goodput (SG) measures how much requested time becomes actual allocated execution. Low SG indicates scheduling losses such as queue waiting or launch failures before resources are granted.

Duty Cycle (DC) measures how much of the allocated execution eventually contributes to durable training progress. Low DC reflects allocated-but-unproductive runtime due to fail-stop events, fail-hang stalls, and the subsequent diagnosis, recovery, or failover processes.

Quality of Productivity (QoP) measures how close productive execution is to the healthy throughput baseline. Low QoP indicates that the job is still completing training iterations but slower than expected (e.g., from fail-slow degradations). QoP measures throughput efficiency only during active progress; fail-hang stalls yield zero progress and are counted as Duty Cycle loss, not QoP loss.

3.2 Experimental Setup

Profiling and Data Collection. We collected data from Kuaishou's production training cluster over a span of three months, encompassing 20,629 distributed training jobs with an average allocated runtime of 30.48 hours. Jobs requested 63.2 GPUs on average, and

the workloads span from recommendation models to large foundation model training.

Our study combines three sources of evidence. First, we gathered scheduler records and job-level telemetry needed to compute ETPR and its sub-metrics, including job submission time, scheduled time, termination time, requested GPU count, and execution attempts. Second, we collected training logs and monitored metrics generated during job execution, which record iteration progress, runtime status, and failure symptoms. Third, we aggregated incident tickets created from monitoring alarms, system failures, and user-reported training issues. These tickets record the affected job, observed symptoms, related logs and telemetry, and the diagnosis and mitigation actions taken by operations engineers.

Measurement of ETPR. In measurement, each temporal quantity in Eq. 1 is first measured over its corresponding wall-clock interval and then converted to GPU-time by multiplying the interval length by the job's requested GPU count N_i. This normalization makes jobs of different scales comparable. Demanded Time is measured from job submission to termination; its pre-allocation segment does not represent physical GPU occupancy, but converts queuing and waiting time to the same GPU-time scale. Allocated Time is measured from the moment the job starts occupying the assigned GPUs until the corresponding pods are terminated. Productive Time is derived from timestamps in training logs and checkpoint records, and counts the execution that produces new durable progress preserved in valid checkpoints. Thus, recomputation after recovery, restart overhead, checkpoint overhead, failure handling, and hangs without durable progress are excluded from Productive Time and treated as Duty Cycle loss. Ideal Time maps the same durable progress to a healthy per-iteration runtime estimated from sufficient executions under the same hardware environment and comparable training configuration. ETPR is therefore an operational estimate of cluster-level productivity attribution, not exact per-step accounting.

Incident Collection and Root-Cause Annotation. We obtain execution failure logs from the internal training-log platform and detect incidents via monitoring alarms, manual inspection of Grafana (e.g., stalls, slow nodes), and user reported issues. For each resolved incident, the ticket links the affected job with its logs, telemetry, and diagnostics. We determine root causes mainly by analyzing contextual error messages in logs, and when explicit error signals are missing, we infer causes by correlating symptom patterns with telemetry and documentation.

Our root-cause taxonomy merges categories from recent LLM training failure studies [18, 20] with operational classifications from recovery practice. In production, recovery actions align with fault domains: user and software errors usually need job restarts or parameter tuning without node replacement; hardware faults require node isolation or replacement; and network faults demand more complex troubleshooting across hardware degradation and communication issues.

Five experienced SREs and software engineering postgraduates independently reviewed 443 confirmed failure incidents with error logs or diagnostics, assigning root-cause labels based on observed patterns and fixes. To ensure labeling consistency, we assessed inter-annotator agreement with Cohen's kappa [6], obtaining 0.82, indicating high consistency. Disagreements were resolved through discussion with the incident submitter or responsible SRE until consensus.

Table 1: ETPR decomposition across operational dimensions. Note that the "contributors" like Stop Failures, Stragglers not only indicate the incidents themselves, but also include the required extra time to mitigate them.

Dimension	Retained Ratio	Contributors
Scheduling Goodput	99.1%	Queue Waiting
Duty Cycle	92.5%	Regular overhead Stop Failures Hangs Maunal Stops
Quality of Productivity	96.1%	Stragglers
ETPR	88.1%	–

3.3 Research Questions

We conduct an empirical study to answer the following questions.

- **RQ1 (Symptoms):** Which observable symptom patterns degrade training productivity in large-scale workloads, as indicated by drops in ETPR sub-metrics?
- **RQ2 (Root Causes):** What root causes and their characteristics underlie these patterns of symptoms, and what challenges hinder their diagnosis and mitigation in practice?
- **RQ3 (Diagnosis & Mitigations):** What diagnosis and mitigation solutions are used to systematically address failures in production training?
- **RQ4 (Effectiveness):** How effective is the proposed framework in diagnosing, localizing, and recovering from productivity-degrading failures, and how does it affect ETPR?

4 Empirical Study Results

4.1 RQ1: Symptoms of Training Productivity Degradation

Table 1 reports the retained productivity ratio of each ETPR dimension. The overall ETPR is 88.1%, meaning that 11.9% of end-to-end training productivity is lost. The losses are uneven across components. Scheduling Goodput remains close to ideal at 99.1%, while Duty Cycle and Quality of Productivity are lower, at 92.5% and 96.1%, respectively. Since Scheduling Goodput contributes only a small loss, the dominant productivity degradation in our cluster occurs after resources have been allocated, either because allocated time does not produce durable training progress or because productive execution proceeds more slowly than the healthy baseline.

Scheduling Goodput Loss. Scheduling loss is relatively small in our cluster. The main observed contributors are queue waiting and resource fragmentation before the job obtains its requested resources. Since this component accounts for only 0.9% loss, we do not further analyze Scheduling Goodput in the rest of the paper.

Duty Cycle Loss. This loss reflects the gap between allocated GPU time and productive GPU time. It consists of both regular baseline overhead and abnormal unproductive runtime. Regular

baseline overhead, including pod setup, initialization, data loading, and checkpoint load/save, accounts for nearly 20% of the Duty Cycle loss. The remaining loss comes from unproductive periods caused by fail-stop events, fail-hang stalls, manual stops, and the ensuing diagnosis and recovery. Notably, the failure itself is often only the beginning: diagnosis, debugging, and recovery can dominate the unproductive period. For example, our logs reveal an incident in which an incorrect parallel configuration led to repeated fail-stop attempts: the user restarted the training job multiple times within a 30-minute window, yet none of the attempts achieved any training progress. Fail-hang stalls are even harder to handle because the job remains alive while making no durable progress. Manual stops also contribute to Duty Cycle loss when users or operators terminate a job before it resumes durable training progress.

Quality of Productivity Loss. This loss occurs when a job continues to run, but its productive throughput remains below the healthy baseline. It is directly reflected by inflated iteration time: the job still completes durable iterations, but each iteration takes longer than expected. Combining QoP measurements with the system telemetry, we observe straggling behavior as the most visible manifestation, where one or a few ranks repeatedly lag and stretch synchronized iterations. Such fail-slow periods enlarge Productive Time because the job keeps completing iterations at degraded throughput, often for a long period before the slowdown is noticed. Even after the anomaly becomes visible, the job may continue running while operators collect evidence and narrow down the slowdown, further widening the gap between Productive Time and Ideal Time.

> **Finding#1**: *The dominant productivity losses occur after scheduling. Duty Cycle loss mainly comes from allocated-but-unproductive periods caused by fail-stop events, fail-hang stalls, manual stops, and the ensuing diagnosis and recovery, while QoP loss comes from fail-slow periods where jobs keep completing iterations but at degraded throughput.*

4.2 RQ2: Root Causes of Training Productivity Degradation

4.2.1 Root Causes Analysis and Characteristics. To understand the root causes of Duty Cycle and Quality of Productivity losses, we trace incident tickets to the responsible components and examine the related logs, telemetry, and mitigation actions by users and operations engineers. Table 2 summarizes the root-cause categories, their typical symptoms, and the average time for manual diagnosis and mitigation.

Software / User Faults. Software and user faults make up most incidents in our traces. They usually appear as explicit fail-stop events, though inconsistent distributed behavior can sometimes cause hangs. Typical causes include configuration errors, framework API misuse, dependency mismatches, buggy user scripts, and invalid parallelism settings. Because users often modify existing codebases for new features, model variants, or training parameters, small inconsistencies can block job startup or cause crashes during execution. These faults are often repairable once identified, but fault diagnosis is a non-trivial task since errors are hidden in large distributed logs and can trigger many cascading messages.

Network Faults. Network-related incidents affect the communication path required by distributed training. Representative root causes include RDMA/InfiniBand congestion or packet loss, switch-port failures, and routing misconfiguration, which may be exposed as NCCL-related errors or communication timeouts. They may surface as explicit fail-stop errors, fail-hang stalls, or fail-slow stragglers, depending on severity and timing. For example, abrupt communication disruption can terminate a job with explicit errors, while congestion, packet loss, or unstable links may prolong collective operations, cause ranks to wait at synchronization points, and inflate iteration time.

Hardware Faults. Hardware faults occur intermittently in our cluster and are often difficult to predict, especially under the sustained stress of long-running LLM training workloads. The hardware tickets we analyzed mainly correspond to physical component failures such as uncorrectable ECC memory errors, XID Error, GPU fall-off-bus events, and GPU overheating. Because large-scale training relies on tightly synchronized distributed execution, a hardware failure on one GPU or interconnect link can quickly propagate to the rest of the job, causing the entire training process to halt or hang. For example, thermal throttling on a single device in our cluster occasionally first appeared as a local slowdown, but then gradually became a system-wide bottleneck that affected all participating ranks.

Resource Exhaustion. Resource exhaustion occurs when jobs exceed available memory, storage, or shared system resources, typically causing fail-stop events during initialization or execution. Examples include CUDA out-of-memory errors, disk quota exhaustion, and shared-memory allocation failures. For instance, we saw NCCL communicator initialization fail due to exhausted shared memory, reporting `failed to extend /dev/shm/nccl-* to N bytes`.

4.2.2 Manual Diagnosis and Mitigation Challenges. The diversity of these underlying causes inevitably constitutes a substantial operational bottleneck. Previously, stability operations within our cluster have depended predominantly on manual analysis of logs, telemetry data, and execution traces, followed by ad hoc remediation actions (e.g., restarting jobs, replacing nodes). Nevertheless, this conventional methodology exhibits significant constraints that profoundly degrade overall training productivity.

For explicit fail-stop from software and user-end faults, the main challenge is the sheer volume of diagnostic data and cascading noise. Even when jobs fail with error logs, users and operators must search through logs from thousands of distributed processes. Pinpointing the initial triggering event among numerous downstream assertions is slow, labor-intensive, and often inaccurate. A single root cause can trigger a cascade of error messages across multiple processes, obscuring the original failure amid downstream noise. Thus, even if the fix (e.g., correcting an index number) is simple, root cause analysis is still time-consuming.

On the other hand, hardware and network faults present a different diagnosis and mitigation challenge. While these faults account for a relatively smaller proportion of overall incidents, they take a much longer time to diagnose, localize, and heal as shown in Table 2. These faults often lack direct, actionable framework-level logs and may surface as silent hangs, persistent stragglers, or intermittent

Table 2: Root-cause categories, typical symptoms, and manual diagnosis & recovery time. The *Proportion* indicates the percentage of incident tickets in each category.

Category	Typical Symptoms	Representative Root Causes	Proportion (%)	Avg Time for Diagnosis and Recovery (min)
Software / User Faults	Fail-stop, Fail-hang	Code bugs, API misuse, dependency issues, misconfigurations, driver mismatch, etc.	74.1	20.3
Network Faults	Fail-stop, Fail-hang, Fail-slow	Communication timeout, connection failure, degraded bandwidth, suspected fabric instability, etc.	15.9	148.2
Hardware Faults	Fail-stop, Fail-hang, Fail-slow	ECC page failure, PCIe fatal error, GPU fallen off the bus, GPU overheating, NVLink-related errors, etc.	5.1	74.5
Resource Exhaustion	Fail-stop, Fail-hang	OOM, disk quota exceeded, shared -memory exhaustion, etc.	4.9	53.1

communication errors. Therefore, the operators need to correlate the training process with hardware telemetry, health signals, and network counters to further localize the degraded component. The underlying reason for the fault could be more complicated to identify. This cross-layer localization makes hardware and network anomalies disproportionately costly despite their lower frequency.

Finding#2: *Root causes behind fail-stop, fail-hang, and fail-slow symptoms are complex and cross-layer. They also show an uneven distribution between incident volume and governance cost: software and user faults occur most frequently and create a heavy log-diagnosis burden, whereas hardware and network faults are fewer but require much longer localization and recovery.*

4.3 RQ3: Diagnosis & Mitigations

Driven by the symptoms and root-cause characteristics identified in RQ1 and RQ2, we build a stability framework that turns operational observations into end-to-end fault handling. As shown in Figure 2, the framework includes a tracing plane that continuously collects logs, traces, and signals from training process, feeding to diagnosis modules for identifying the likely failure category and faulty component, and the recovery plane selects the least disruptive action to restore durable training progress. Together, these components provide an end-to-end path from failure observation to productivity recovery in production training.

Logs and Telemetry Tracing. The tracing plane provides the evidence for downstream diagnosis modules. Each pod deploys a tracing plane with four components. First, a structured log parser ingests stdout/stderr logs and forwards fault-relevant entries to the diagnosis pipeline. Second, a heartbeat monitor tracks rank liveness and triggers stack collection when progress stalls. Third, a CUDA activity tracer instruments collective communication operators via the CUPTI Activity API [28] and records operator-level timing, including start and end timestamps. Fourth, a system telemetry monitor collects hardware and communication telemetry, such as GPU/CPU utilization, memory usage, network status, NIC bandwidth, and PCIe bandwidth. A Master Agent aggregates these signals and performs diagnosis, localization, and recovery using process-level information from all pods.

The Master Agent serializes diagnostic events per job to prevent conflicting recovery during concurrent or cascading failures. Diagnostic routines are mutually exclusive, so no new diagnosis for a job starts until the current one finishes. In cascading cases (e.g., network degradation → fail-slow → fail-hang heartbeat timeout), evaluations run in order—hang detection waits for slow detection to complete. A disruptive fail-stop event has highest priority for recovery actions; the diagnostic pipeline then combines all slow- and hang-detection results with crash logs to produce a single root-cause diagnosis.

4.3.1 Diagnosis. To reduce wasted time and the cost of degraded execution, the diagnosis module shortens the time interval between symptom onset and actionable diagnosis. For fail-stop events, it quickly filters massive distributed logs, finds the primary error entry, and classifies the likely root cause so targeted recovery can be selected. For fail-hang and fail-slow symptoms, where explicit error logs may be missing, it prioritizes fast detection and localization of the rank that blocks progress or slows training. When symptoms are pinpointed, and hardware or network issues are suspected, the diagnosis triggers the recovery plane to isolate or replace the faulty component. We then detail diagnosis schemes for each failure type.

Fail-Stop Diagnosis. The diagnosis pipeline consists of an offline knowledge-base construction phase and an online diagnosis phase. Offline, we mine high-frequency patterns from historical failure logs to build lexical rules, parse logs into structured templates with Drain [15], and store template embeddings with validated labels in a vector knowledge base.

Online, fail-stop events emit explicit error messages, but the root cause may be obscured by large distributed logs and cascading secondary errors. Using the common earliest-error heuristic from elastic distributed-training engines [33], the framework compares timestamps across processes, finds the earliest-failing process, and extracts its primary error entry, filtering downstream cascading errors. Diagnosis then proceeds in three increasingly general stages: Aho–Corasick lexical matching for frequent patterns, semantic retrieval over the knowledge base when rules are inconclusive, and LLM reasoning for cases with low retrieval confidence. SRE

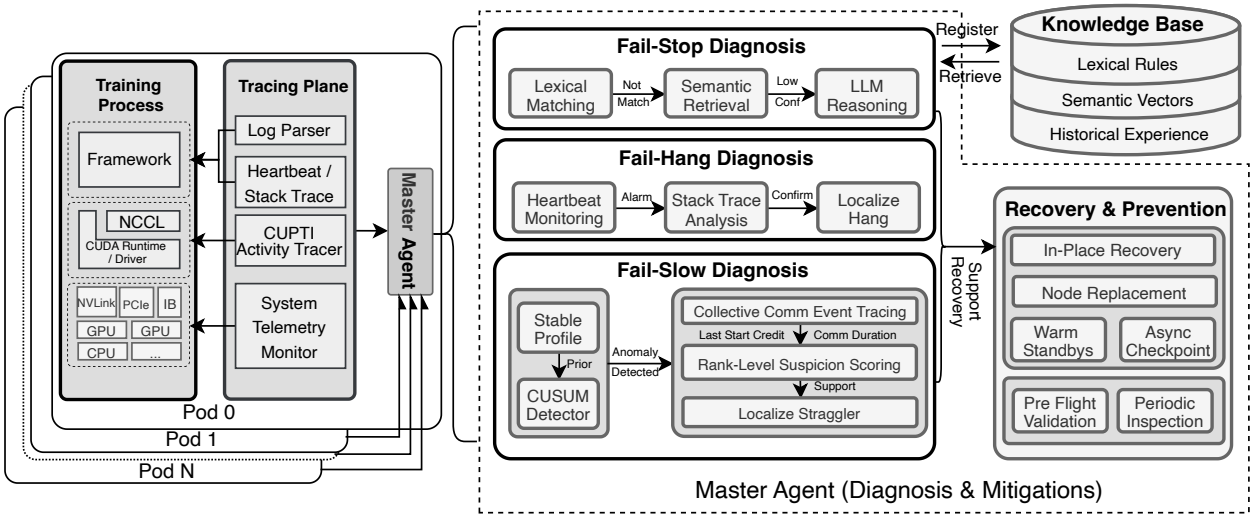


Figure 2: Overview of the stability framework.

engineers validate new failure cases before adding their embeddings or rules to the knowledge base.

When the primary error suggests a hardware or network fault but lacks a precise faulty rank, the framework performs targeted localization before recovery. It correlates hardware-related errors with device health signals and validates suspected pods or nodes using tools such as `dcgmi diag` [27]. For network-related errors, including `mpirun` SSH failures or source-unclear collective errors, it runs targeted tests such as `nccl-tests` [26], `ib_write_bw` [29], or cross-node all-reduce checks to identify the faulty communication path.

Fail-Hang Diagnosis. Silent training stalls are difficult to diagnose because the process ceases to make measurable progress while failing to emit an explicit error signal. To identify and localize such hangs, and to distinguish them from expected long-latency operations such as checkpointing, we employ a diagnostic pipeline that integrates heartbeat-based liveness monitoring, periodic stack trace sampling, and statistical outlier detection.

The process starts with the master rank monitoring the heartbeat of the training loop. Instead of using a fixed timeout to kill the job immediately, the framework treats a sustained heartbeat loss as a signal to enter diagnosis mode. It then uses `py-spy` [11] through remote shells to sample the main-thread stack of every rank three times at equal intervals. A hang is confirmed only when the sampled call stacks remain identical across all three sampling rounds. This criterion facilitates the discrimination of genuine deadlock conditions from transient synchronization delays. Once a hang is verified, the system identifies the problematic rank among correlated ranks. In the context of synchronous collective communication, non-faulty ranks typically block at the same synchronization barrier and consequently exhibit identical call stacks. Based on this observation, the system collects the final stack traces and applies majority voting to identify the dominant execution pattern across ranks. Any rank whose stack deviates from this dominant pattern is treated as an outlier and mapped to its physical host node.

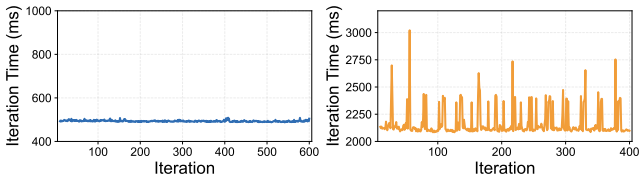


Figure 3: Iteration time distributions of dense and MoE models under standard execution conditions.

Fail-Slow Diagnosis. Fail-slow anomalies do not produce explicit error logs and instead appear as degraded throughput while the job continues to complete iterations. We handle them with a two-phase detection-then-localization pipeline. Detection operates at the iteration level with low overhead; localization is activated only after a persistent slowdown is confirmed, reducing tracing overhead during normal execution.

Anomaly Detection. Hardware or network degradation typically manifests as shifts in iteration time. However, normal training also exhibits periodic fluctuations, and different model structures may naturally produce different timing distributions. For instance, training workloads based on Mixture-of-Experts (MoE) architectures may exhibit periodic iteration-time spikes even under healthy hardware, while dense models remain relatively stable, as shown in Figure 3.

To separate genuine slowdowns from normal workload variation, the framework builds a stable reference profile from historical anomaly-free iterations using the median and Median Absolute Deviation (MAD). Since the model's training steps may exhibit cyclic behavior, we use the Auto-Correlation Function (ACF) [4] to estimate the dominant period and compute phase-aware baselines. Incoming iteration-time signals are normalized against these baselines into anomaly scores. The detector is a non-parametric method that calibrates itself entirely from each job's own anomaly-free stable prefix. Phase-aware baselines are built from the median and

MAD of clean iterations, and both the noise-filtering threshold and the alarm limit are derived from the deviation-score distribution over the same stable segment. An alarm is raised only when abnormal scores accumulate persistently beyond this self-calibrated limit, distinguishing sustained degradation from transient jitter.

Straggler Localization. A fail-slow rank progresses more slowly than its peers due to degraded computation or network. Unlike other ranks that quickly enter the next collective after completing the preceding computation, the fail-slow rank often lags behind and becomes one of the last to start. This persistent delay leads to two observable effects: increased start-time skew among ranks in the collective and prolonged communication, since synchronization waits for all participants. Motivated by this, we quantify for each rank how frequently and how strongly it appears as the last starter, weighting this measure by the communication duration of the event.

Once the iteration-level anomaly detector confirms a persistent fail-slow event, the framework activates a fine-grained localization module to identify the fail-slow rank and map it to its hosting node. For each collective communication event e in communication group G, every Rank $r \in G$ reports its invocation start and end timestamps, denoted by t_r^{start} and t_r^{end}.

A normalized score that quantifies how strongly Rank r behaves as the latest starter in event e:

$$w_{r,e}^{\text{latest}} = \frac{t_r^{\text{start}} - \min_{r' \in G} t_{r'}^{\text{start}}}{\max_{r' \in G} t_{r'}^{\text{start}} - \min_{r' \in G} t_{r'}^{\text{start}}}. \tag{2}$$

For each event, we also measure the effective communication duration δ_e^{comm} as the elapsed time from the latest collective start in group G to the latest collective end. We then aggregate the latest-start signal weighted by communication duration:

$$\Phi_r = \frac{1}{|E_r|} \sum_{e \in E_r} w_{r,e}^{\text{latest}} \cdot \delta_e^{\text{comm}}, \tag{3}$$

where E_r is the set of collective events participated in by Rank r, and Φ_r is the straggler localization score of Rank r. Because localization is triggered only after an alarm, we compute Φ_r using only events observed in this window, enabling fast transient localization.

4.3.2 Fault Recovery. After diagnosis and localization, the recovery plane selects the least disruptive action that can restore durable training progress. The goal is to reduce additional unproductive time while avoiding unnecessary reconstruction of the entire job environment. We build a recovery plane in which an agent runs inside each training pod and serves as a job-local control plane. The agent bridges the scheduler and training framework, reports process-level status, and executes recovery actions according to the diagnosed fault type.

In-Pod Recovery Plane. In our deployment, a training pod is directly bound to an underlying physical GPU server node. We therefore use the pod as the operational replacement and recovery unit. Instead of letting the pod lifecycle directly determine the lifetime of the training job, the agent decouples the training process lifecycle from the pod lifecycle. It takes over process launch, environment validation, code distribution, network setup, and recovery execution within the pod. This allows recoverable faults to be handled by process-level relaunch within the existing pod set, avoiding full pod reconstruction or re-scheduling when possible. The global scheduler remains responsible for resource allocation and pod replacement, while high-frequency job-specific recovery actions are delegated to job-local agents.

Table 3: Recovery actions selected according to diagnosis and localization results.

Diagnosis Result	Recovery Action
User code, API misuse, or configuration fault	Provide traceback and knowledge-base hints; relaunch in place after correction when applicable.
Environment or dependency mismatch	Validate the runtime environment and roll affected components back to known-good versions.
Resource exhaustion	Clean non-durable temporary files, caches, or stale shared-memory segments while preserving checkpoints and user data.
Transient fail-slow symptom	Observe across windows; apply in-place relaunch if degradation persists but localization confidence is insufficient.
Localized hardware, network, or persistent straggler fault	Replace the affected pod or node, update rank placement, and relaunch from the latest valid checkpoint.

In-Place Recovery. In-place recovery is applied when the diagnosed fault can be resolved without replacing the underlying pod or physical machine. As shown in Table 3, for user-side code bugs or API misuse, the system extracts the complete traceback from logs and retrieves related issues and solutions from the knowledge base with LLM assistance. For environment-related faults, such as driver, library, or dependency mismatches, the agent runs validation scripts and rolls affected components back to known-good versions. For resource exhaustion, the agent cleans non-durable temporary files, caches, and stale shared-memory segments while preserving durable checkpoints and user data. After remediation, the agent relaunches all pods from the latest valid checkpoint within the existing pod set, preserving prepared environments and avoiding a new scheduling and pod initialization cycle.

Pod Replacement and Recovery. Pod replacement is applied to faults that cannot be safely resolved within the existing pod, including confirmed hardware or network faults and persistent localized stragglers. Once the faulty pod is identified, the scheduler evicts it from the active training placement and replaces it with a healthy pod from the warm standby pool. The job is relaunched from the latest valid checkpoint with updated placement and rank mapping. After replacement, the faulty pod and its underlying physical machine are marked and excluded from future scheduling until repair or inspection is completed.

4.3.3 Proactive Fault Prevention. Beyond reactive diagnosis and recovery, the framework also performs proactive checks to reduce avoidable failures before and during training.

Pre-Flight Job Validation. Before training begins, the system validates job configurations, dependency consistency, GPU health, topology, communication readiness, and storage capacity. It checks whether GPU counts are compatible with parallelism dimensions, validates YAML configurations, filters GPUs with uncorrectable ECC or thermal issues, verifies PCIe/NVLink topology, runs a short communication test, and estimates memory and storage requirements. Nodes with frequent failure history are flagged and excluded from future scheduling.

Table 4: Classification quality and measured per-log decision time on 254 held-out logs not resolved by lexical matching. "W" denotes weighted metrics.

Methods	W-Precision	W-Recall	W-F1	Avg. Time
Retrieval-Only	79.47%	74.70%	74.85%	0.275s
LLM-Only	84.27%	82.21%	82.54%	4.210s
Retrieval+LLM	82.34%	80.24%	80.46%	0.312s

Table 5: Performance comparison among different methods on Fail-Slow Anomaly Detection.

Methods	Accuracy↑	FPR↓	FNR↓
Z-Score	68.75%	30.00%	35.71%
MAD	73.43%	26.00%	28.57%
BOCD	89.06%	10.00%	14.28%
Ours	93.75%	4.00%	14.28%

Continuous Fleet Inspection. An independent background process performs low-frequency periodic inspection across the cluster, covering CPU frequencies, interconnect status, hardware error logs, memory errors, storage health, network switch connectivity, and hardware temperatures. If a serious issue is detected, the affected resource is isolated, or the training job is safely paused to prevent cascading failures.

4.4 RQ4: Effectiveness

To evaluate the effectiveness of the proposed stability framework, we conduct case studies covering comparative experiments with other baselines and present real production cases to validate its performance in diagnosing and mitigating failures. In addition, we report the impact on ETPR and operational experience from a 4-week deployment of the stability framework in the cluster.

Case#1: Effectiveness of Fail-Stop Diagnosis. We collected 14,651 failure logs from the cluster log store to build a diagnostic knowledge base supporting lexical matching, semantic retrieval, and LLM-based reasoning. We use `bge-small-en` [41] for retrieval and GPT-5.4 [30] for LLM reasoning, with temperature set to 0 to reduce randomness.

To test diagnosis beyond frequent lexical patterns, we use a held-out set of 254 logs not resolved by lexical matching. All configurations take the same extracted primary error entry and predict predefined categories. Retrieval-Only classifies using the historical knowledge base; LLM-Only predicts directly from the error entry. Retrieval with Conditional LLM first retrieves and calls the LLM only when retrieval confidence falls below its threshold. Table 4 shows that Retrieval + Conditional LLM matches LLM-Only classification performance while reducing average per-log decision time, yielding a better accuracy–latency trade-off.

Case#2: Diagnosis & Recovery from Hardware-induced Fail-Stop. During a distributed training job utilizing 128 GPUs, the workload abruptly crashed at iteration 15,030 (out of a scheduled 327,500 iterations). The failure triggered a massive cascade of error logs across multiple ranks, including generic `std::runtime_error` and framework-level job termination messages indicating aborted MPI processes. Utilizing the lexical fast path of our fail-stop diagnostic framework, the system effectively bypassed the downstream noise and isolated the true root cause on the earliest-failing unit (Rank 14): an `uncorrectable ECC error` surfaced in the NCCL watchdog thread. Upon classifying this as a permanent hardware fault, our framework automatically cordoned off the offending node associated with error event and triggered automated recovery. A healthy replacement pod was provisioned, and the training job safely restarted from its most recent checkpoint. The entire automated diagnosis and recovery cycle from failure detection to restart durable training progress was completed in under 10 minutes.

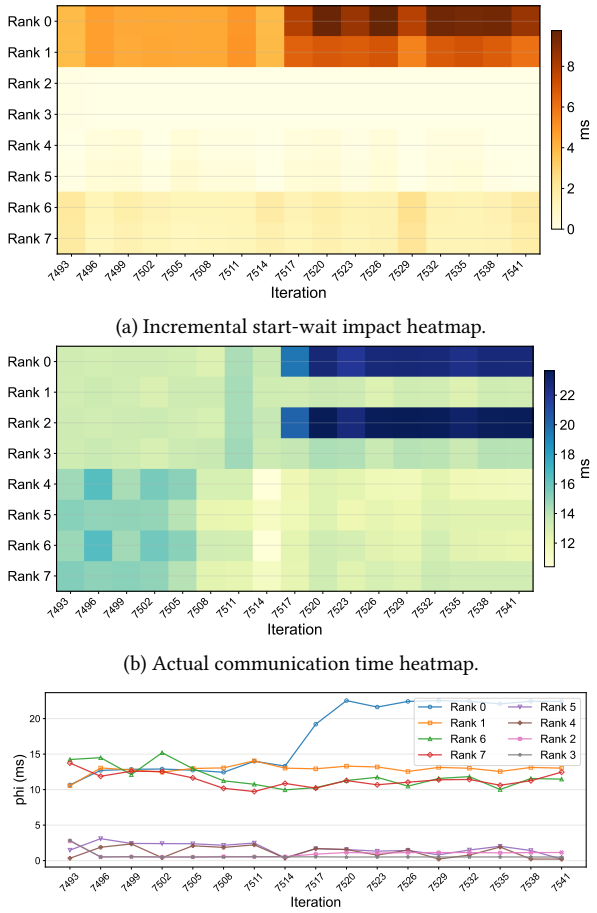
Case#3: Diagnosis & Recovery from Fail-Hang. During a training job, our framework detected sustained heartbeat loss and entered diagnosis mode. It then used `py-spy` to sample the stacks of participating ranks for three rounds at fixed intervals. Within 6 minutes, the framework confirmed the hang and identified the suspicious rank. Stack-pattern analysis showed that one outlier rank on a specific node was stuck at `irecv` during the forward pass. After the confirmation, the framework issued a graceful stop signal and isolated the affected node, replaced it with a hot-standby machine, and restarted the job. The hang symptom did not recur after replacement.

Case#4: Effectiveness of Fail-Slow Anomaly Detection. To evaluate our fail-slow anomaly detection, we collected a dataset consisting of 64 distinct training sessions: 50 completely normal runs and 14 fail-slow anomaly cases. To ensure robustness across different workloads, these sessions include both the Dense model and the Mixture-of-Experts model trained by Megatron-LM. We compared our adaptive approach against three widely used statistical time-series baselines: Z-Score, Median Absolute Deviation (MAD), Bayesian Online Change Point Detection (BOCD) [2].

Table 5 shows the performance of all detection methods. On this dataset, our algorithm outperforms the baselines, achieving an accuracy of 93.75% with a 4.00% False Positive Rate (FPR). Legacy baselines like Z-Score and MAD struggle with the iteration time fluctuations, leading to higher false alarm rates. In contrast, our method effectively ignores these acceptable variations while remaining sensitive to genuine slowdowns.

Case#5: GPU Frequency Throttling Injection for Straggler Localization. We evaluate straggler localization with a compute-side fail-slow injection on an 8-GPU cluster spanning 4 nodes, configured with PP = 2, DP = 2, and TP = 2. The injected fault is applied only to Rank 0 by capping its GPU SM clock frequency, reducing its compute throughput by approximately 40%. As a result, Rank 0 reaches collective synchronization later than other ranks, causing peer ranks to spend more time waiting. Once a stable slowdown is confirmed, the localization module ranks all participating ranks using Φ_r within the post-alarm window. Rank 0 becomes the top-ranked rank within 3 iterations and remains consistently separated from the others, demonstrating that the method can localize the injected slow rank with a relatively short response delay.

Deployment Impact and Experience. By deploying the full stability framework, we compute the aggregated ETPR over all finished



(a) Incremental start-wait impact heatmap.

(b) Actual communication time heatmap.

(c) Per-Rank Φ_r score over iterations.

Figure 4: Diagnostic views for Case#5: GPU frequency throttling injected on Rank 0. Rank 0 dominates both heatmaps and achieves the highest Φ_r score throughout the anomaly window, confirming it as the root-cause rank.

training jobs within four weeks to evaluate its production impact. The aggregated ETPR increases from the previous baseline of 88.1% to 91.9% after deployment. This gain is mainly attributable to improvements in Duty Cycle and Quality of Productivity, which reach 95.6% and 97.1%, respectively, while Scheduling Goodput remains nearly unchanged at 99.0%.

By further inspecting the diagnosis and recovery process of failure events, we observe that explicit fail-stop events can be diagnosed and recovered faster after deployment. This improvement comes from the closed-loop diagnosis and recovery pipeline, which enables some software and user faults to be automatically recovered through rollback or restart. In addition, failures caused by configuration mistakes, such as invalid parallelism settings, are clearly reduced due to the prevention mechanisms. For interruptions and slow-node issues caused by network and hardware faults, automated diagnosis and recovery for explicit infrastructure failures, together with faster discovery of potential slow nodes, reduce the average diagnosis and recovery time to less than 30 minutes. These mechanisms also help the SRE team identify more potentially faulty machines, which are then removed from the scheduling pool. Nevertheless, some hard user-side cases, such as out-of-memory errors, still cannot be handled well by the framework, and misdiagnosis can still introduce extra diagnosis and recovery time. We will further discuss these limitations in RQ5.

5 Discussion

Our study and deployment show that ETPR provides a practical operational lens for training productivity governance, and that targeted diagnosis and mitigation mechanisms can accelerate the process of detection, localization, recovery. Nevertheless, some limitations remain.

Approximate ETPR Measurement. ETPR is an operational estimate, not an exact accounting metric. It is computed from scheduler records, training logs, and healthy-baseline estimates, which may miss fine-grained transitions in fragmented production runs. The baseline is derived from healthy runs with the same hardware and similar training configurations, and can shift with changes in model architecture, parallelism, data distribution, or code. For new tasks or significantly changed models, a new healthy baseline is required before ETPR can accurately reflect QoP degradation. Future work can improve ETPR by defining more robust indicators of ideal progress, such as phase- and configuration-aware baselines and confidence estimates for ETPR values.

Hard Cases in Diagnosis and Mitigation. Automated diagnosis and mitigation reduce manual effort, but some failures still require deep expert analysis. CUDA out-of-memory (OOM) is a representative hard case. Naively reducing batch size or other memory-related parameters may avoid OOM, but can also reduce MFU and underutilize hardware. In one recommendation model training incident, the job was interrupted with CUDA OOM and was restarted multiple times by the user. Manual analysis with the torch memory snapshot tool [32] revealed that an extra embedding-related memory region was not released: the user-defined training step function returned loss inside the output object, causing PyTorch to retain the computation graph and its intermediate activations. This diagnosis took about two days to find the root cause. Such cases require an in-depth understanding of user code and framework memory behavior, suggesting future work on memory-leak detection and memory-aware code diagnostics.

Framework Fault Tolerance. Our current framework relies on the master agent to aggregate signals and coordinate diagnosis, localization, and recovery. This design simplifies deployment, but also leaves the framework vulnerable to master-side failures: if the master agent crashes or becomes unreachable, the system cannot automatically detect the failure, elect a new master, or continue recovery without manual intervention. Future work can improve the framework by adding master health monitoring, leader election, and state replication, so that diagnosis and recovery can continue after master failover with minimal interruption.

Misdiagnosis Cost. Misdiagnosis can introduce additional operational costs. For fail-stop events, an incorrect classification of a software or user issue as a hardware or network fault may trigger unnecessary device health checks, isolation, and pod replacement. These actions can increase Duty Cycle loss even when the original

fault could have been resolved with a lighter-weight recovery. For instance, in our early deployment, the knowledge base lacked fine-grained annotations for specific GPU XID errors. When a user's buggy custom CUDA kernel caused an out-of-bounds memory access, the GPU driver reported an `Xid 31` (memory page fault). The semantic retrieval broadly misclassified this as a physical hardware failure, leading to an unnecessary pod eviction. Since the root cause was software-based, the relaunched job predictably crashed again at the same training step on the new pod, triggering a loop of false alarms that wasted hours of compute and depleted the healthy standby pool. To eliminate this misdiagnosis cost, we subsequently refined our knowledge base by exhaustively annotating GPU XID codes to clearly distinguish user-triggered software faults from actual hardware defects. For fail-slow detection, a false slowdown alarm may activate fine- grained straggler localization, including communication tracing and rank-level analysis, which has higher overhead than normal monitoring. Future work can make recovery decisions more confidence-aware by combining diagnosis confidence, historical node health, and estimated recovery cost before escalating to expensive localization or replacement actions.

6 Related Work

This section reviews the most recent work on fault tolerance and the frameworks for harnessing the stability of model training.

Large Model Training Failure Studies. Prior work has studied training reliability from conventional DL systems to modern LLM clusters. Early studies built taxonomies of software bugs, configuration errors, framework failures, and hardware faults in DL platforms [17, 48]. Recent LLM studies analyze user-reported failures in open-source ecosystems [12, 46] and bugs in distributed training frameworks [47], showing that complex training stacks amplify reliability challenges. Production-cluster studies further reveal the operational difficulty of hardware faults, network instability, and recovery procedures [18, 22]. In our study, we present ETPR to analyze reliability-induced productivity loss in production training and reveal the key reliability issues behind it.

Stability Governance Frameworks. Recent work has developed diagnosis and mitigation systems for large-scale training failures. L4 [20] automates log-based root-cause localization using cross-job, spatial, and temporal log patterns, and LLM-augmented diagnosis further improves failure analysis from training logs [18]. AIOps platforms consolidate logs, metrics, and alerts for end-to-end operational diagnosis [38]. However, some failures do not leave clear error logs, especially fail-hang and fail-slow cases. To handle them, recent systems turn to runtime and communication-level signals. Greyhound [40] detects transient stragglers by combining NCCL hooks with Bayesian Online Change Point Detection. Other systems identify abnormal nodes or communication bottlenecks through execution-timeline comparison, in-flight NCCL kernel inspection, or communication-stack dependency tracing [7, 8, 43, 44]. End-to-end training stability frameworks further integrate checkpointing, failure diagnosis, network diagnostics, and recovery into production infrastructure [21, 36]. Our work learns from these systems and organizes diagnosis and mitigation around the productivity losses exposed by ETPR and its sub-metrics, aiming to handle the main reliability issues in our production training cluster and improve end-to-end training productivity.

7 Conclusion

We investigate reliability-induced productivity degradation in large-scale model training and propose the Effective Training Productivity Ratio (ETPR), a metric that decomposes end-to-end training productivity into three components: Scheduling Goodput, Duty Cycle, and Quality of Productivity. Applying the ETPR framework to Kuaishou's production training cluster reveals that the dominant sources of productivity loss are i) time during which resources are allocated but not effectively utilized for productive training work, and ii) excessive execution time arising from a broad spectrum of reliability-related issues, including hardware failures, network disruptions, and user-induced errors.We present an integrated stability-governance framework that unifies log-based diagnosis of fail-stop events, stack-based localization of fail-hang stalls, adaptive detection of fail-slow behavior, straggler localization, and hierarchical recovery. Empirical evaluations and production deployment demonstrate that the framework enhances diagnostic efficiency, accurately identifies anomalous ranks or nodes, and reduces both unproductive execution time and operation in degraded modes, thereby achieving sustained improvements in ETPR through increased duty cycle and improved quality of productivity.

Acknowledgment

We would very much like to thank anonymous reviewers for their valuable comments. Special thanks must go to the overall AI platform team at Kuaishou Inc. and RAIDS Lab team at Beihang University, for their collaborative contribution and countless technical discussion. This work is supported in part by National Key R&D Program of China (No. 2024YFB4505901), in part by NSFC (No. 62402024), in part by Beijing Natural Science Foundation (No. L241050), in part by the Fundamental Research Funds for the Central Universities, and, last but not the least, by Kuaishou Technology Research Fund. For any correspondence, please refer to the project lead and coordinator Prof. Renyu Yang (renyuyang@buaa.edu.cn).

Data Availability

Due to proprietary and confidentiality constraints, the industrial datasets and codes used in this study are not publicly released for the time being.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Ryan Prescott Adams and David JC MacKay. 2007. Bayesian online changepoint detection. *arXiv preprint arXiv:0710.3742* (2007).

[3] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. 2022. Flamingo: a visual language model for few-shot learning. *Advances in neural information processing systems* 35 (2022), 23716–23736.

[4] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. 2015. *Time series analysis: forecasting and control.* John Wiley & Sons.

[5] Eric A Brewer. 2015. Kubernetes and the path to cloud native. In *Proceedings of the sixth ACM symposium on cloud computing*. 167–167.

[6] Jacob Cohen. 1968. Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit. *Psychological bulletin* 70, 4 (1968), 213.

[7] Weihao Cui, Ji Zhang, Han Zhao, Chao Liu, Jian Sha, Bo Sang, Bingsheng He, Minyi Guo, and Quan Chen. 2026. {FLARE}: Anomaly Diagnostics for Divergent {LLM} Training in {GPU} Clusters of {Thousand-Plus} Scale. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 1021–1035.

[8] Yangtao Deng, Lei Zhang, Qinlong Wang, Xiaoyun Zhi, Xinlei Zhang, Zhuo Jiang, Haohan Xu, Lei Wang, Zuquan Song, Gaohong Liu, et al. 2025. Mycroft: Tracing Dependencies in Collective Communication Towards Reliable LLM Training. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 254–269.

[9] Jianbo Dong, Bin Luo, Jun Zhang, Pengcheng Zhang, Fei Feng, Yikai Zhu, Ang Liu, Zian Chen, Yi Shi, Hairong Jiao, et al. 2025. Enhancing Large-Scale AI Training Efficiency: The C4 Solution for Real-Time Anomaly Detection and Communication Optimization. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1246–1258.

[10] Jianbo Dong, Kun Qian, Pengcheng Zhang, Zhilong Zheng, Liang Chen, Fei Feng, Yichi Xu, Yikai Zhu, Gang Lu, Xue Li, et al. 2025. Evolution of Aegis: Fault Diagnosis for {AI} Model Training Service in Production. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 865–881.

[11] Ben Frederickson. 2026. py-spy: Sampling profiler for Python programs. https://github.com/benfred/py-spy.

[12] Yanjie Gao, Ruiming Lu, Haoxiang Lin, and Yueguo Chen. 2025. An Empirical Study of Issues in Large Language Model Training Systems. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 122–133.

[13] Yanjie Gao, Xiaoxiang Shi, Haoxiang Lin, Hongyu Zhang, Hao Wu, Rui Li, and Mao Yang. 2023. An empirical study on quality issues of deep learning platform. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 455–466.

[14] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[15] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE international conference on web services (ICWS)*. IEEE, 33–40.

[16] Tao He, Xue Li, Zhibin Wang, Kun Qian, Jingbo Xu, Wenyuan Yu, and Jingren Zhou. 2023. Unicron: Economizing self-healing llm training at scale. *arXiv preprint arXiv:2401.00134* (2023).

[17] Yi He, Mike Hutton, Steven Chan, Robert De Gruijl, Rama Govindaraju, Nishant Patil, and Yanjing Li. 2023. Understanding and mitigating hardware failures in deep learning training systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–16.

[18] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. 2024. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 709–729.

[19] Zhiyi Hu, Siyuan Shen, Tommaso Bonato, Sylvain Jeaugey, Cedell Alexander, Eric Spada, James Dinan, Jeff Hammond, and Torsten Hoefler. 2025. Demystifying NCCL: An in-depth analysis of GPU communication protocols and algorithms. In *2025 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 48–59.

[20] Zhihan Jiang, Junjie Huang, Guangba Yu, Zhuangbin Chen, Yichen Li, Renyi Zhong, Cong Feng, Yongqiang Yang, Zengyin Yang, and Michael Lyu. 2025. L4: Diagnosing large-scale llm training failures via automated log analysis. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 51–63.

[21] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 745–760.

[22] Apostolos Kokolis, Michael Kuchnik, John Hoffman, Adithya Kumar, Parth Malani, Faye Ma, Zachary DeVito, Shubho Sengupta, Kalyan Saladi, and Carole-Jean Wu. 2025. Revisiting reliability in large-scale machine learning research clusters. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1259–1274.

[23] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[24] Qingkai Meng, Hao Zheng, Zhenhui Zhang, ChonLam Lao, Chengyuan Huang, Baojia Li, Ziyuan Zhu, Hao Lu, Weizhen Dang, Zitong Lin, et al. 2025. Astral: A datacenter infrastructure for large language model training at scale. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 609–625.

[25] NVIDIA. 2025. InfiniBand. https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17/infiniband.

[26] NVIDIA. 2025. NCCL Tests. https://github.com/NVIDIA/nccl-tests.

[27] NVIDIA. 2025. NVIDIA DCGM. https://developer.nvidia.com/dcgm.

[28] NVIDIA. 2026. NVIDIA CUDA Profiling Tools Interface (CUPTI) 13.2. https://developer.nvidia.com/cupti.

[29] OFED. 2025. RDMA Perftests. https://github.com/linux-rdma/perftest.

[30] OpenAI. 2026. Introducing GPT-5.4. https://openai.com/index/introducing-gpt-5-4/.

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[32] PyTorch Contributors. 2025. Understanding CUDA Memory Usage. https://docs.pytorch.org/docs/2.11/torch_cuda_memory.html.

[33] PyTorch Contributors. 2026. Torch Distributed Elastic Error Propagation. https://docs.pytorch.org/docs/2.13/elastic/errors.html.

[34] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 3505–3506.

[35] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[36] Borui Wan, Gaohong Liu, Zuquan Song, Jun Wang, Yun Zhang, Guangming Sheng, Shuguang Wang, Houmin Wei, Chenyuan Wang, Weiqiang Lou, et al. 2025. Robust llm training infrastructure at bytedance. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 186–203.

[37] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. 2023. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 364–381.

[38] Zeying Wang, Junhong Liu, Penghao Zhang, Xiaoyang Sun, Xu Wang, Tianyu Wo, Chunming Hu, Chengru Song, Jin Ouyang, and Renyu Yang. 2025. KAIOPS: A Platform Solution of End-to-End Multi-Modal AIOps for AI Training at Scale. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 3192–3203.

[39] Arissa Wongpanich, Tayo Oguntebi, Jose Baiocchi Paredes, Yu Emma Wang, Phitchaya Mangpo Phothilimthana, Ritwika Mitra, Zongwei Zhou, Naveen Kumar, and Vijay Janapa Reddi. 2025. Machine learning fleet efficiency: Analyzing and optimizing large-scale Google TPU systems with ML productivity goodput. *arXiv preprint arXiv:2502.06982* (2025).

[40] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. 2025. {GREYHOUND}: Hunting {Fail-Slows} in {Hybrid-Parallel} Training at Scale. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 731–747.

[41] Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. 2024. C-pack: Packed resources for general chinese embeddings. In *Proceedings of the 47th international ACM SIGIR conference on research and development in information retrieval*. 641–649.

[42] Yifan Xiong, Yuting Jiang, Ziyue Yang, Lei Qu, Guoshuai Zhao, Shuguang Liu, Dong Zhong, Boris Pinzur, Jie Zhang, Yang Wang, et al. 2024. {SuperBench}: Improving cloud {AI} infrastructure reliability with proactive validation. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 835–850.

[43] Yitang Yang, Junhong Liu, Jiapeng Chen, Xiaoyang Sun, Tianyu Wo, Chunming Hu, Chengru Song, Jin Ouyang, and Renyu Yang. 2025. Kair: A Statistical and Causal Approach to Pinpointing Stragglers in Distributed Model Training. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 3754–3759.

[44] Zhiyi Yao, Pengbo Hu, Congcong Miao, Xuya Jia, Zuning Liang, Yuedong Xu, Chunzhi He, Hao Lu, Mingzhuo Chen, Xiang Li, et al. 2025. Holmes: Localizing irregularities in {LLM} training with mega-scale {GPU} clusters. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 523–540.

[45] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*. Springer, 44–60.

[46] Guangba Yu, Zirui Wang, Yujie Huang, Renyi Zhong, Yuedong Zhong, Yilun Wang, and Michael R Lyu. 2026. Why Does the LLM Stop Computing: An Empirical Study of User-Reported Failures in Open-Source LLMs. *arXiv preprint arXiv:2601.13655* (2026).

[47] Xiao Yu, Haoxuan Chen, Feifei Niu, Xing Hu, Jacky Wai Keung, and Xin Xia. 2025. Towards Understanding Bugs in Distributed Training and Inference Frameworks for Large Language Models. *arXiv preprint arXiv:2506.10426* (2025).

[48] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An empirical study on program failures of deep learning jobs. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 1159–1170.

Received 2026-05-01; accepted 2026-07-01