Evolution of Aegis: Fault Diagnosis for AI Model Training Service in Production

Jianbo Dong[*], *Kun Qian*[*], *Pengcheng Zhang*[*], *Zhilong Zheng, Liang Chen, Fei Feng, Yichi Xu, Yikai Zhu, Gang Lu, Xue Li, Zhihui Ren, Zhicheng Wang, Bin Luo, Peng Zhang, Yang Liu, Yanqing Chen, Yu Guan, Weicheng Wang, Chaojie Yang, Yang Zhang, Man Yuan, Hanyu Zhao, Yong Li, Zihan Zhao, Shan Li, Xianlong Zeng, Zhiping Yao, Binzhang Fu, Ennan Zhai, Wei Lin, Chao Wang, Dennis Cai*
Alibaba Cloud

Abstract

Despite the success of diagnosis systems in traditional cloud computing, these systems are not suitable for pinpointing faults in AI model training cloud scenarios due to the differences in computing paradigms between traditional cloud computing and model training. As one of the largest cloud providers, we present Aegis, a fault diagnosis system specifically designed for AI model training service. We share our experience in the motivation, design, and evolution of Aegis. Keeping easy-to-deploy as the primary principle, Aegis Phase-1 started by enhancing existing general-purpose diagnosis systems. After several months of evolution, Aegis Phase-2 cogitatively customized the collective communication library for sophisticated failure localization in runtime without modifying customer code. Besides the failure localization, we further equipped Aegis with the capabilities on handling performance degradation and failure checking before delivery. Aegis has been deployed in our production training cloud service for one year. Aegis decreases more than 97% of the idle time wasted by diagnosis, 84% of the training task restart count, and 71% of the performance degradation.

1 Introduction

The AI model training has brought about tremendous revolutions to today's cloud services. In a typical AI model training scenario, the customer (i.e., model designer) submits the AI model as a task to the training cluster maintained by a training cloud service provider (e.g., AWS, Azure and Google). The hosts within the training cluster perform model computation, then synchronize intermediate results via collective communication, then perform model computation, and continue. Eventually, these hosts finish a training process (typically lasting weeks) based on the above synchronization way.

Reliability is important. The reliability of the training cluster is crucial to both the training service provider and customers. If a failure occurs during model training, the entire training process should be restarted, resulting in a significant loss of time and money [36, 45, 60]. As the scale of the AI model increases, maintaining the reliability of the training cluster becomes increasingly more important and challenging.

Why maintaining the reliability is challenging? The large-scale AI model training clusters are equipped with tens of thousands of high-tier GPUs (e.g., NVIDIA A100 [13] and H100 [16]) to cooperate through high-speed network connections (including rail-optimized network [1] and NVLINK [17]) to achieve high-throughput training. Nevertheless, the failure rates of these high-performance components (both hardware and network) are much higher than regular ones such as CPU and Clos network used in general cloud computing scenario (see §2.1.1 for more details). Even worse, due to the synchronization nature of model training, the single-point failures resulting from the high-performance GPUs within one host can lead to a cascading failure across all hosts in this training task. The diagnosis systems used in traditional cloud computing [23–26, 28, 29, 31, 32, 34, 35, 37–43, 46–48, 50–55, 57–59, 61, 63, 64], in principle, localize the root causes by tracing back the source-destination path through the sequence of system component calls such as the 5-tuples or devices of given faults; however, in the model training, task failures spread from a single point to the entire cluster, causing the culprit to be hidden among error reports from all hosts. As a result, the diagnosis systems used for traditional cloud computing are not suitable for localizing the root causes in model training cluster.

The state-of-the-art efforts. To fill this gap, some diagnosis systems for the model training scenarios were proposed, such as SuperBench [60] deployed in Microsoft and MegaScale [36] deployed in ByteDance. SuperBench provides a comprehensive benchmark suite for gray failure checking before the cluster deployment. While useful, it is hard to localize the root causes of faults occurred in the model training runtime. On the other hand, MegaScale identifies failures by monitoring CUDA events from "critical code segments" in customer model. This is only suitable for scenarios where the model designer and model training service providers belong to the same party; otherwise, no model designer (i.e., customer)

* Equal Contribution.

is willing to allow the training service provider to monitor or modify their code. Such an assumption is impractical to public model training service providers.

Our approach: Aegis. As one of the largest model training service providers in this world, we decided to build a practical diagnosis system that achieve our key goal: diagnosing root causes of failures in service runtime without modifying customer code. This goal is important to public model training service providers like us, because (1) we need to cover failures during the entire lifecycle, not only the cluster deployment phase and (2) our diagnosis system should be general and transparent for various customers. To this end, we build Aegis. This paper shares our experience in the motivation (§2), design (§3), and evolution (§4) of Aegis.

Driven by our in-production investigation and motivation (detailed in §2), we started the design of Aegis. In the Phase-1 of Aegis (§4.1), we enhanced our existing diagnosis system with the training-output log and a diagnosis procedure fitting in the large-scale model training scenario. This enhancement enabled us to narrow the entire task-level failure down to specific error devices in major failure cases. For more sophisticated cases, we employ offline diagnosis as the backstop.

While Aegis Phase-1 constructs a comprehensive diagnosis system handling various failures in our model training scenario, once the offline diagnosis is involved, all used hosts in this large-scale AI model training task need to be isolated, greatly complicating the scheduling of the training cluster and harming the overall cluster utilization. Therefore, Aegis evolved to Phase-2 (§4.2). Phase-2 focused on diagnosing most failure cases directly at runtime. We conducted a comprehensive investigation of the possible runtime information sources. Adhering to the principle of minimizing customers' perceptions, we chose to customize the Collective Communication Library (or CCL) to acquire runtime status because of the following two reasons. (1) in mainstream training frameworks (e.g., Megatron [49] and DeepSpeed [8]), CCL is integrated as an independent plugin. Customizing CCL would not introduce any code modification in customers' models or training frameworks. (2) CCL "sits at the boundary" of computation and communication, making it a perfect place for generating ideal failure diagnosis information. Through defining appropriate CCL metrics and constructing the corresponding runtime diagnosis system, we improve the runtime diagnosis ratio from 77% to close 100%.

Besides failure diagnosis, significant performance degradation is another crucial problem. Based on our experience, we further develop the basic-metric correlating diagnosis and enhanced procedure-aware diagnosis to discover the root cause of performance degradation (§5).

Other than solving issues during the training procedure, we note that more than 73% of tasks failed at the initialization phase. This phenomenon indicates errors already existed in the cluster before the model training tasks start. To minimize unnecessary retries from customers, we deploy Check Before Delivery (CBD) to conduct essential checks before delivering hosts to customers (§6).

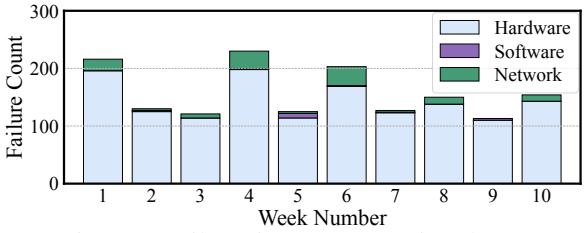


Figure 1: Failures in a representative cluster.

We present the diagnosis efficiency on our training clusters during the evolution of Aegis (§7). For confidential reasons, we use the statistics from our in-house LLM training project as a representation. During the last 16 months, the training scale of our project increased by more than 40×. Impressively, with the deployment of Aegis Phase-1, the GPU idle time caused by failure diagnosis is decreased by 71%. With the deployment of Aegis Phase-2, almost all model training task failures can be diagnosed in runtime. The GPU idle time is further reduced by 91%. For handling performance degradation, with the gradual deployment of the correlation diagnosis and enhanced procedure-aware diagnosis, the degree of performance degradation is decreased by 71%. With the deployment of CBD, the training task restart count is decreased by 84%.

2 Motivation

2.1 Challenges Introduced by Model Training

Large-scale model training is fundamentally different from traditional cloud computing. As shown in Figure 1, we studied the repairing tickets from one of our largest training clusters (containing $O(1K)$ hosts with $O(10K)$ GPUs) in production during the last ten weeks. Each week, 100-230 critical failures occurred in the training cluster, which is orderly higher than that in general cloud computing data centers. If not pinpointed and fixed on time, these failures may contribute to model training task crashes. This subsection presents our findings on why more failures occur and why failure diagnosis is harder in large-scale model training.

2.1.1 Higher hardware failure ratio

Large-scale model training introduces unique challenges, particularly in terms of hardware reliability.

High failure rate of high-tier GPUs. High-tier GPUs (e.g., NVIDIA A100 [13] and H100 [16]), the most fundamental component of the large-scale model training cluster, exhibit a significantly higher failure rate than traditional computing hardware. For instance, our statistical data show that an A100 GPU, on average, fails after approximately 400 days of operation, whereas an H100 GPU has an even shorter mean time to failure of around 200 days. Given a large-scale training task involving thousands of GPUs, this failure rate is orders higher than that of general cloud computing scenarios. As shown in Figure 2, 45.6% of failures are caused by GPU-related reasons (i.e., GPU execution error, GPU driver error, GPU memory error, CUDA error, NVLINK error and GPU ECC error).

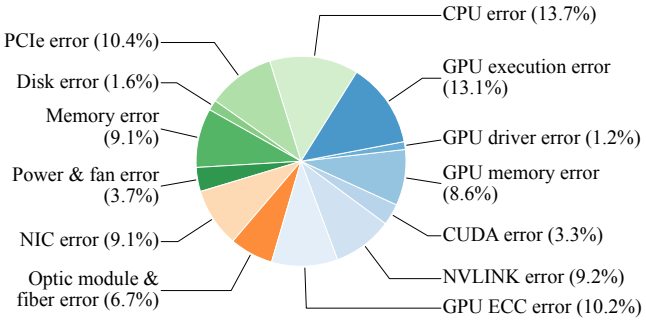


Figure 2: Types of failures encountered in production.

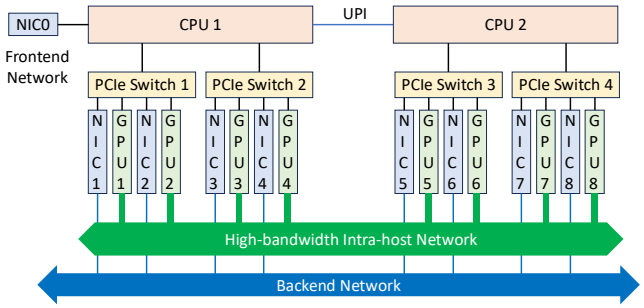


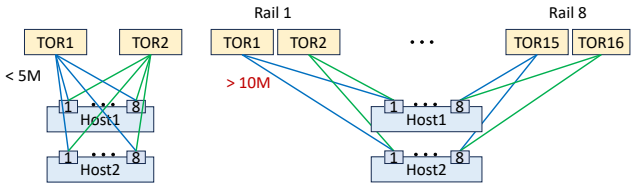
Figure 3: Intra-host network topology.

Complex intra-host network topology. As shown in Figure 3, each host in the our training cluster is equipped with eight GPUs and eight NICs, interconnected via PCIe. GPUs in the same host also utilize NVLINK for inter-GPU communication, which, while enhancing performance, introduces more complex forwarding paths and higher failure rates. Our operational experience has revealed 9.2% issues related to NVLINK failures, as well as 10.4% performance anomalies due to PCIe error. These intra-host network problems are very rare in general cloud computing scenarios.

More link failures in large-scale model training cluster. To maximally utilize the high bandwidth provided by the intra-host network, rail-optimized networking [45] is widely deployed in construction training clusters. While this approach improves performance, as shown in Figure 4, compared with the traditional single-ToR topology, the rail-optimized connections require long-distance links. It necessitates using the optic modules and optic fiber to overcome the distance limitations of copper cables. However, optic modules and fiber have a higher failure ratio [18]. Statistics from our production reveal that optic modules and fiber can introduce a $1.2 \sim 10\times$ higher failure ratio compared with DAC, which varies according to different vendors and link speeds. Although we have implemented the dual-ToR design proposed in HPN [45] to mitigate these issues, the inherently higher failure rates still introduce significant operational challenges. While one of the two dual-ToR links fails, the entire training task may not crash, but may encounter significant performance degradation.

2.1.2 Lacking direct root cause indicators

Besides the high failure ratio, the specific workload of large-scale model training differs from general cloud computing



(a) Single-ToR access (b) Rail-optimized access

Figure 4: Different types of host accessing topology.

tasks, further introducing difficulties in diagnosing the root cause of failures. The limited diagnostic information provided by mainstream training frameworks adds additional difficulty to the failure diagnosis. In traditional general cloud computing scenarios, a failure only affects several specific hosts, with clear fault localization indicators (e.g., the specific 5-tuple connection encounters high RPC completion time). In contrast, in the model training, task failures spread from a single point (e.g., a host or a link) to the entire cluster, causing the culprit to be hidden among all the faulty tasks.

In addition, the entire large-scale model training procedure heavily relies on collective communication to cooperate with all GPUs. Usually, the failure is observed throughout the entire training task crash (e.g., most hosts concurrently encounter the same CCL timeout error). While there are many potential root causes, the CCL timeout is often the first signal to identify the occurrence of a failure. As a result, diagnosing the problem typically begins with the network, but the root cause may not be in the network.

2.2 Why Our Existing Systems Do Not Help?

We have three diagnosis tools used in our data center networks for general cloud computing.

Tool 1: Network monitoring and analysis. During the running of training tasks, each NIC and switch continuously generates logs recording their runtime status, especially warning and error messages. Furthermore, a series of statistic-based monitors are deployed in hosts and switches (e.g., RX/TX throughput, Out-of-order number, ECN mark number on each port). This diagnosis system is primarily designed to collect and analyze logs and statistical data from NICs and switches. By matching specific network failure patterns (e.g., critical errors shown in the log and abnormal statistics), it can automatically identify the failure node and isolate it.

Tool 2: RDMA Pingmesh. In the general cloud computing scenario, we have deployed TCP Pingmesh service on a large scale, and we have transformed it into RDMA Pingmesh during the support of high-performance block storage service, which is similar to R-Pingmesh [41]. RDMA Pingmesh can actively probe the network to diagnose connectivity and high latency issues.

Tool 3: Inband network diagnosis. The above two systems are responsible for end-to-end information to diagnose the faulty device in the training cluster. However, in some complex failure scenarios, we also need hop-by-hop statistics,

rather than end-to-end information, to discover the root causes of failures. We build such a system, which can conduct coloring on specific packets and trace their forwarding status among each single switch.

The procedure: Tool1+Tool2+Tool3. The entire diagnosing procedure (combining the above three tools) works as follows. When application monitors report an abnormal event, this abnormal event points to a clear source-destination pair. Our network monitoring and analysis system is triggered to analyze whether critical errors occur among the source-destination pair. If so, devices with critical errors would be isolated. If there is no such clear conclusion, Pingmesh results are used to verify whether abnormal packet loss and high latency occur in the network. If so, the in-band diagnosis system is triggered to color specific traffic that passes through the source-destination path hop-by-hop to determine the location of the failure.

Limitations. This diagnosis system works well in general cloud computing scenarios; however, it is not suitable in the model training scenario due to the following two reasons.

Reason 1: Focusing on the network itself. In general cloud computing scenarios, a failure notification points to a clear source-destination path. Therefore, existing diagnosis systems only need to focus on identifying faults inside the network. However, serving model training in the cloud is quite different. It heavily relies on collective communication to cooperate with all GPU hosts. Any single failure would cause a cascading task crash across all participating hosts. Therefore, a large number of secondary issue errors may overshadow the root cause error. We encounter numerous false positives in the diagnostic results by directly employing existing diagnosis systems in model training production.

Reason 2: Focusing on single request/response. Our traditional diagnosing systems diagnose faults in the request-response way, focusing on the issues of individual connections. In contrast, large-scale model training requires correlating information from multiple devices to accurately diagnose faults, necessitating an automated localization system that is vacant in the previous procedure.

2.3 Limitations of State-of-the-Art Efforts

State-of-the-art general-purpose diagnosis systems [23–26, 28, 29, 31, 32, 34, 35, 37–43, 46–48, 50–55, 57–59, 61–64] move a step towards handling failures beyond the network. Some of them further consider the correlation between different devices to improve the analysis accuracy [29, 30, 33, 35, 44]. However, due to the lack of taking into account model training-specific features such as collective communication, they are unable to cover many failures in the model training scenario. For example, in LLM training, each single collective communication forces dozens or even hundreds of GPUs in synchrony. Simply extending these correlation-analyzing methods to such a large scale forces the comparison among many nodes. According to our experience, it leads to an increase in unacceptable diagnosis latency and accuracy loss. We, therefore, turn to investigate whether there is any diagnosis system specifically designed for the model training scenario. There are two representative diagnosis solutions built for model training clusters and large-scale deployed in production (i.e., SuperBench [60] deployed in Microsoft and MegaScale [36] deployed in ByteDance). Unfortunately, as shown in Figure 5, both cannot perfectly fit the demands in the model training cloud scenario due to the following reasons.

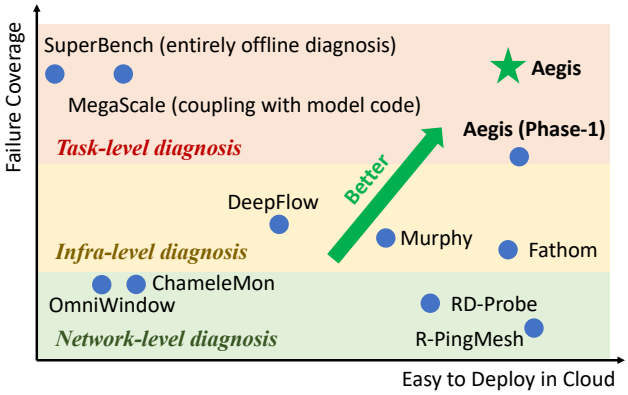


Figure 5: Limitations of state-of-the-art solutions.

Time-consuming offline diagnosis is insufficient. SuperBench constructs a comprehensive benchmark suite, including computation/communication microbenchmarks and typical end-to-end model training benchmarks. By executing this benchmark suite before delivering the training cluster to production, SuperBench can proactively figure out possible issues. We have deployed a similar procedure during each host's online procedure. However, failures may happen during the entire life cycle of the cluster, not only before the online procedure.

It is also impractical to execute SuperBench after any failures. Initializing a single model training benchmark would take tens of minutes, making the entire SuperBench execution lasting hours. Employing this system as the only diagnosis method would introduce a great waste of computation resources and a bad user experience (i.e., prolong waiting time after each failed training).

Deep coupling with customers' code is inappropriate. MegaScale, on the contrary, proposes a fully runtime method to execute failure diagnosis. It monitors the executions of CUDA events in "critical code segments" to diagnose training issues. It is a proper solution for companies that train models for their own use (e.g., ByteDance and Meta). However, it is not suitable for us who offer public AI training services for customers. The reasons are twofold.

First, MegaScale needs a clear definition of "critical code segments". However, as a cloud service provider, we offer training services for various customized models from different customers. Each model may have different architectures and implementations, leading to varying definitions of "critical code segments". Requiring clients to explicitly define

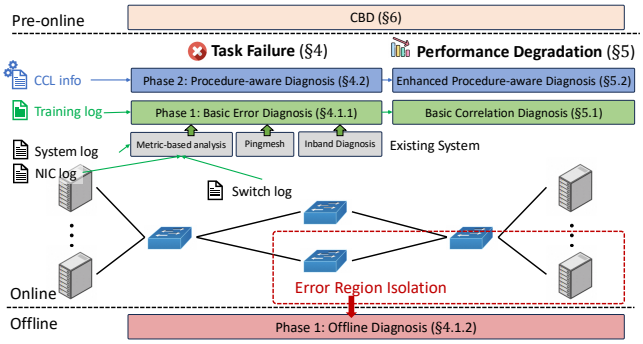


Figure 6: Aegis overview.

the "critical code segments" they use is also impractical, as many model details are highly confidential. Therefore, we need a generic and efficient solution that introduces minimal intrusion on customers.

Second, monitoring CUDA events requires initializing a customized monitor module exactly in the same thread these CUDA events are called. It means this initialization can only be explicitly called in customers' model code. Deploying a new tool that forces modifying customers' model code is hard. The negotiation between our salesman and customers mainly bottlenecks the entire deploying procedure.

3 Aegis Overview

We decided to build a diagnosis system (named Aegis) for the AI model training scenario. Aegis achieves our key goal: diagnosing the culprit of failures in service runtime without modifying customer code. Notice that, in production, when a failure happens, the most urgent issue is timely locating which device is introducing this failure/degradation. Then, we can isolate this device and go on the training. The root cause analysis is conducted offline (i.e., after the isolation), which is another big topic and not covered in Aegis. Figure 6 shows the overview of Aegis. Aegis focuses on two types of abnormalities: training failure and performance degradation. Training failure is the first critical disease we need to cure. Aegis has experienced a two-phase evolution for handling training failure. Considering the limitations of our existing diagnosis system as discussed in §2.2, in Aegis Phase-1, we further involve the error information from the training log and construct a training-specific runtime diagnosis procedure (§4.1.1). We also design a comprehensive offline diagnosis backstop for hardcore cases that cannot be handled in runtime diagnosis (§4.1.2). During the operation, we find that a non-negligible number of failure cases need an offline diagnosis, which leads to unsatisfied GPU utilization. Therefore, we evolve Aegis Phase-2 to get more training-specific information through customizing CCL to conduct the training procedure-aware diagnosis (§4.2). For handling performance degradation, based on previous experience, we design both metric correlation diagnosis (§5.1) and enhanced procedure-aware degradation diagnosis (§5.2). To further improve user experience, we add a new "pre-online" process conducting efficient cluster checking before delivering it to the customer (§6).

4 Task Failure Diagnosis

This section illustrates the evolution of Aegis. We start by enhancing existing diagnosis systems and then move to constructing a better runtime diagnosis system.

4.1 Phase-1: Enhancing Existing Systems

4.1.1 Basic error diagnosis

In most failure cases, there are no clear indicators even to determine the scope of the failure. Therefore, diagnosis begins with identifying whether the current issue originates from the endpoint or the network. At the very beginning of our diagnosis practice, most failures are located through manual diagnosis. Our engineers filter error and warning logs on each host and switch through different sources (e.g., OS dmesg [9], training log, CCL log, NIC driver, switch syslog [5] and customized counters) and analyze the time-sequence relationship between these abnormal reporters with the actual failure. After dealing with hundreds of online failure cases, a series of critical errors are summarized and transformed into our initial automated task failure diagnosing system. Two main challenges are faced and conquered in this procedure.

Not all reported errors are critical. Error logs are widespread during the entire training process. Not all errors deterministically lead to task failures or performance degradation. Considering ECC errors as an example, there are multiple types of ECC errors. Only double-bit ECC errors (e.g., XID 48 Error [21]) and uncorrectable ECC errors (e.g., XID 94/95 Error) are root causes of failures. Other single-bit ECC errors (e.g., XID 92 Error) and correctable ECC errors (e.g., XID 63/64 Error) indicate HBM memory errors, but do not trigger failures. However, widespread errors are reported from different hosts near the task's crash time. If all these hosts were directly isolated, the entire utilization of the training cluster would be greatly damaged.

Our solution first involves identifying critical errors that intrinsically lead to task anomalies, such as GPU missing, PCIe lane drops, or NVLink issues. When such issues are detected, the corresponding host is directly isolated. This method covers only a portion of potential anomalies.

In the early stages of operating LLM training clusters, our engineers devoted much effort to handling online training failures. They summarized other error patterns that certainly lead to task failure. This experience formed the foundation for our critical error diagnosis: $CriticalError()$. Critical errors contain several categories in production: (1) hardware failure (e.g., double-bit ECC error, link down, GPU/NVLink/NIC missing, fan error, and power error) (2) unrecoverable software failure (e.g., GPU/NIC driver error) and (3) unbearable performance degradation (e.g., GPU/host overheat).

Not all critical errors point to a clear location. Besides the clear critical errors, which can directly indicate failure location, many other errors do not point to specific nodes. A

widely encountered example is the crash of network connections (e.g., "connection reset by peer"). These failures can trigger NCCL error handlers, causing the corresponding training thread to exit and leading to cascading thread crashes in other hosts. We also construct a distributed error list to record these errors: *DistError()*.

We enhance our fault diagnosis process with the above experience, as shown in Appendix A (Algorithm 1).

• If a machine encounters critical errors, it is isolated, and the task is restarted.

• If distributed errors appear on only two hosts, it means errors occur on these hosts. They are isolated, and the task is restarted. There is a trade-off behind this process logic. When the potential culprit host set is small enough, it is more efficient to directly isolate all nodes in this list to accelerate the diagnosis procedure. However, the side-effect is that some normal hosts in this set are also isolated, introducing resource wastage. In production, we take the size of this potential culprit host set to 2 to achieve the sweet point.

• For distributed errors across multiple machines, *RootDiag()* analyzes the reported errors to identify if they can be clustered by the source or destination. If GPU G_j is the root cause of the failure, then connections from G_j and connections to G_j crash the first time. Therefore, *RootDiag()* can precisely determine the faulty G_j. If not, it means that the root cause is not on the host-side. Then, network components need to be further checked.

• If errors are distributed without a clear pattern, systemic issues are most likely, such as network or configuration problems. We develop *ConfigCheck()* and *NetDiag()* to conduct further diagnosis. *ConfigCheck()* maintains a checklist and corresponding scripts to identify various causes of configuration errors. *NetDiag()* is constructed by the existing DCN diagnosing system illustrated in §2.2.

• If all the above procedures cannot pinpoint the root cause of the failure, then all hosts used in this training task need to be isolated and conduct offline diagnosis (details in §4.1.2).

Lesson: Exhausting host-side critical failures first is the most efficient way to diagnose. In large-scale model training, host-side issues may be misinterpreted as network issues. In practice, 71% distributed failures turn out to be irrelevant to the network. Therefore, in environments with mixed network-side and host-side faults, solving host-side issues first is important and efficient.

4.1.2 Offline failure diagnosis

After the entire diagnosis procedure, some issues still cannot be directly diagnosed with the available runtime information. To locate the root cause of these failures, we have designed an offline failure localization mechanism. It isolates and diagnoses all suspicious hosts used in the current training task. Unlike SuperBench [60] monopolizes the entire cluster for multiple hours of testing, our offline system diagnoses failures in parallel for targeting hosts. After ensuring each host is problem-free, this host is returned to online service.

Parallelized offline failure localization. To expedite the offline localization process, we design a parallelized approach to maximize efficiency. In the offline localization procedure, all hosts undergo a series of self-checks, including stress testing for CPU, GPU, PCIe, and NVLink. This part is fully parallelized (i.e., each host runs self-checks independently). If an issue is identified during the single-host self-checks, that host is marked as faulty. If no issues are detected in the single-node tests, further multi-host failure diagnosis is required.

The next step is a multi-host failure diagnosis. We select typical models similar in SuperBench [60], and we include more emerging typical models (e.g., MoE models [12] and Multimodal models [4, 20]). The basic idea is that the end-to-end training failure is triggered by a specific combination of computing and communication in the customer's model. Traversing our selected typical models, we can determine which model covers this combination of computing and communication and use it as the reference model in the following multi-host diagnosis. The cluster is then divided into smaller segments, and the reference model is trained on different segments independently, ultimately pinpointing the problematic host. Once a subset of the cluster is confirmed normal, these hosts are promptly returned to the resource pool, minimizing resource wastage.

Topology-aware parallel localization. During parallel localization, the partitioning of different subsets is crucial. Since the network is shared among different hosts, splitting hosts indiscriminately may lead to parallel training tasks competing with the same network links. This can result in two side effects. (1) If the failure occurs within the network (e.g., silent packet loss) and the parallel diagnosing tasks use the same problematic link, both tasks will be affected, making the parallel diagnosing procedure fail to locate the root cause. (2) If the fault is not in the network but on the host, the current running of training tasks may share the same network link and cause congestion, leading to inaccurate diagnosis results.

To handle this issue, hosts are not evenly split into different subsets. We split hosts into two subsets according to their location in the physical network topology. We count the number of hosts in different Pods and ToR groups (in rail-optimized topology, multiple ToRs are used to serve the same host, called a ToR group), respectively. Hosts are split according to whether they belong to the same Pods and ToR groups. As a result, during the parallel diagnosis, traffic from different diagnosing tasks would not interfere with each other in the network.

After training the reference model on two subsets, the subset encountering failure is chosen for further diagnosis. Hosts in the other subset can be returned to online service. With the proceeding of the parallel diagnosis, left hosts would gradually converge to be in the same ToR groups. In this case, the splitting could be arbitrary since any splitting would not lead to network congestion. If both subsets encounter failures, the

root cause exists in both subsets, and then all hosts in these two subsets need to be isolated.

Transforming the missing piece into a new clue. The above procedure can cover the diagnosis of most failure cases. However, there is a missing piece: if the root cause is the misbehavior of Core (Tier-3) switches or Aggregation (Tier-2) switches, this independent parallel diagnosis would miss this root cause of failure (since we delicately minimize traffic passing Core and Aggregation switches).

We indeed encounter this issue in production. There was a failure in a training task occupying 1.5K GPUs. In the offline failure diagnosis, we successfully find a reference model to reproduce the failure. However, when we employ the parallel diagnosis, the failure cannot be reproduced in any subsets of hosts. We are surprised at first since there is a missing piece that the default diagnosis procedure cannot handle. After several times of reproduction and further analysis, we derived insight from this missing piece and concluded that there must be some misbehavior in the Aggregation switches. Through succeeding switch-specific diagnosis, the root cause is clear: silent packet loss happens on one Aggregation switch. But why is this switch error not detected by the $NetDiag()$ in the online diagnosis? We encounter an exceptional kind of silent packet loss, which only drops packets larger than 1KB. As a result, the RDMA Pingmesh system does not throw any error since the size of all probing packets is 64B.

After comprehensive reviews of this failure case, we make several enhancements to our system. (1) We supplement the offline diagnosis to handle this case automatically. (2) We enhance RDMA Pingmesh to cover varied lengths of probes.

4.2 Phase-2: Procedure-aware Diagnosis

Despite significant improvements in failure localization through enhanced legacy systems, we still encounter many cases that need to trigger the offline diagnosis to finally locate the root cause, inevitably leading to considerable computation resource waste. Furthermore, even though most cases can be handled with Aegis Phase-1, we encountered several rare cases where typical models could not reproduce the failures encountered in production. In these cases, to finally solve customers' problems, we had to cooperate deeply with them to reproduce failure cases online and collect more diagnosis information. These failures are caused by systematic reasons, requiring not just basic error information but also training-procedure-specific information to find out the culprit. Therefore, we upgrade our system to diagnose runtime failures using a procedure-aware approach.

4.2.1 What is the ideal solution?

The main difficulty of fully online diagnosis is lacking precise information. Therefore, we enhance our online task monitoring capabilities to provide more valuable runtime information. However, several practical constraints must be considered before implementing such improvements:

High confidentiality. LLM training is a synchronous process, which is also the main reason for the difficulties in failure diagnosis. Accurate localization of such issues requires detailed outputs, and since the root causes vary across different cases, the necessary data for diagnosis also differs. Choosing the right metrics for high-confidence diagnosis is crucial.

Minimal customer modifications. Comprehensive fault localization typically requires extensive metrics collection, which demands tight fusion with customers' code or training frameworks. However, as a model training cloud service provider, extensive modifications to customer code or training framework are impossible in production. The ideal solution is fully transparent to customers.

Low overhead. Adding new information collection and processing should introduce minimal overhead to avoid impacting the main training tasks.

4.2.2 Customizing CCL is the bridge

After comprehensive consideration, we employ a customized CCL as the bridge for enhanced runtime diagnosis. This decision is based on several valuable characteristics of CCL:

First, in mainstream training frameworks (e.g., Megatron and DeepSpeed), collective communication is a modularized component that can be replaced independently. By replacing the CCL, we can collect customized diagnostic information without any change in customers' model codes. Unlike higher layers, which may involve extensive modifications, customizing the CCL offers a more practical solution for diagnostics.

Furthermore, collective communication sits at the boundary of computation and communication. Precise runtime information from this layer can provide clear information about the host-side processing time (computation) and network-side processing time (communication). This information is vital for the localization of faulty devices.

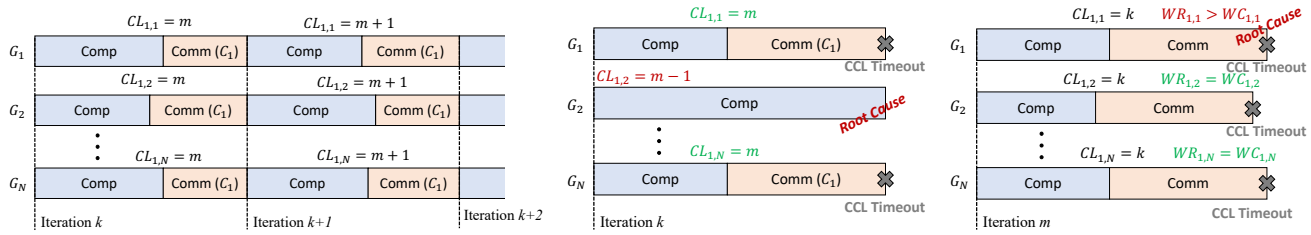
Information collection. During the training, our customized CCL records several statistics of each communication operator (C_i) in each GPU (G_j).

- Collective launch count($CL_{i,j}$) records how many times C_i is launched by G_j.
- Work request count ($WR_{i,j}$) records how many work requests in C_i are launched by G_j.
- Work completion count ($WC_{i,j}$) records how many work requests in C_i are finished by G_j.

We have tried other different metrics in our testbed and found that the information mentioned in our paper is both sufficient and necessary. Keeping collected metrics lightweight and easy to deploy is important.

As shown in Figure 7a, all GPUs alternately execute computation and collective communication in a synchronized way. In normal cases, $CL_{i,j}$ from different G_j keep the synchronized increasing in each iteration. We further present how to localize failures with the above statistics.

Scenario-1: failure in computation. If a failure occurs in the computation phase, as shown in Figure 7b, G_n ($n = 2$ in this case) fail in launching the succeeding C_i ($i = 1$ in this



(a) Normal training iterations. (b) Failure in computation. (c) Failure in communication.

Figure 7: Customizing CCL for failure diagnosis.

case). As a result, all other workers in the same communicator would stall at the C_i and crash owing to CCL timeout. In this case, $CL_{i,n} < CL_{i,j\neq n}$ in the same group. As a result, G_n is pinpointed as the root cause of the failure.

Scenario-2: failure in communication. If the failure is in communication, then the transmission of a specific work request in C_i would fail, causing all GPUs in this group to endure CCL timeout as shown in Figure 7c. More detailed statistics $WR_{i,j}$ and $WC_{i,j}$ are used for further diagnosis. In normal GPUs, $WR_{i,j} = WC_{i,j}$. If $WR_{i,n} < WC_{i,n}$ ($n = 1$ in this case), it means G_n is related to the root cause. We further conduct *NetDiag()* on all sources and destinations related to these abnormal work requests.

After deploying deep runtime diagnosis in production, almost all training failure cases can be pinpointed in runtime (more details in §7).

Limitations. Leveraging information from collective communication is a compromise that is easy to deploy. Actually, just using collective communication information is not enough to find the root cause of failures. However, as aforementioned, collective communication is located at the boundary between computation and communication. This intrinsic characteristic is important for our primary goal: locating the culprit.

As a cloud service provider, our customers may employ various versions of official or self-constructed images with differing CUDA, drivers, and CCL versions. To provide consistent diagnosis ability for all customers, we need to make sure (1) our diagnosis system can seamlessly deploy in different images and model training tasks and (2) solutions deployed in different environments should deliver the same diagnosis information for delivering the consistent diagnosis performance. Therefore, we need to provide our corresponding customized version based on all CCL-released versions. Although the above limitations exist in customizing CCL, it is still easy to deploy compared with other alternatives (e.g., customizing the training framework or modifying customers' model code).

5 Performance Degradation Diagnosis

Besides complete training task failure, some device abnormalities may not crash the entire training but lead to significant performance degradation. These abnormalities should also be diagnosed on time. Since the training task is still running when a performance degradation occurs, we cannot use *OfflineDiag()* as the final fallback solution. Therefore, we design a degradation diagnosis system.

5.1 Basic Correlating Diagnosis

Similar to Phase-1 in §4.1, we first leverage the existing runtime statistics to detect the potential performance degradation.

Key metric selection. The first challenge is to determine precise metrics that can help recognize and diagnose the fault. After solving various cases in production, we notice that most performance degradation is triggered by one single abnormal device, where two categories of metrics can indicate it.

(1) Abnormal operating metrics. These metrics are designed to directly indicate that some components are running at abnormal conditions. For example, the Retran metric denotes how many packets are retransmitted per second. In ordinary cases, this Retran metric should always be zero. The high Retran metric denotes misbehavior in the network.

(2) Performance metrics. These metrics are designed to reflect the execution efficiency of specific components. The abnormal evolution of one component would lead to overall performance degradation. For example, the Actual TensorFLOPS metric denotes how many tensor float-point calculations are completed each second.

We select 20+ metrics according to operating in production including host metrics (e.g., CPU utilization, GPU utilization, GPU temperature, and PCIe utilization) and network metrics (e.g., Bandwidth utilization, Retransmission count, Switch port queue length, and ECN count). Owing to the confidential policy, we cannot release a detailed list of all metrics used. Intuitively, if one metric (especially abnormal operating metrics) is running in a faulty range (e.g., continuously high Retransmission metric), then this value should lead to performance degradation. To filter these clear signals, we set corresponding thresholds for these metrics, respectively.

However, simply using static thresholds cannot fit various training scenarios. Simply setting thresholds for every single metric would lead to numerous misjudgments since resource utilization varies greatly, even in the normal training procedure. Considering that the entire training procedure is well-organized, we, therefore, leverage the synchronizing training characteristics for further correlating diagnosis.

Cross-host correlating diagnosis. The monitored results of the same metric from different hosts should follow the same

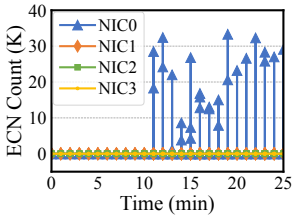
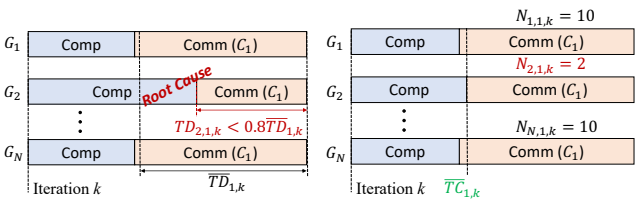


Figure 8: Abnormal ECN metric evolution.



(a) Computation degradation. (b) Communication degradation.

Figure 9: Customizing CCL for performance diagnosis.

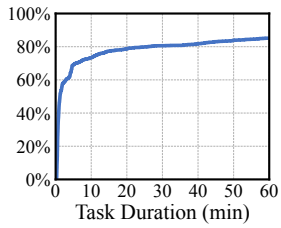


Figure 10: Durations of training tasks in production.

changing pattern among different training iterations. The basic idea is that a small part of nodes causes the entire performance degradation, and performance degradation usually occurs along with abnormal changes in some metrics, which can indicate the root cause. Therefore, we design a Z-Score [22] outlier analyzer for different metrics.

For each selected metric, the outlier analyzer calculates the average value λ and standard deviation δ in the period T. If the metric value from a single host is higher than $\lambda + 2\delta$ for the lasting T, then this host is defined as an outlier. T is set to be ten minutes in production. In practice, the abnormal node usually generates significantly different values compared with others, which makes $\lambda + 2\delta$ a simple and good enough threshold for diagnosis in production.

Actually, we have tried a series of other outlier analysis mechanisms (e.g., LOF [27], Isolation Forest [3] and DBSCAN [7]). They lead to similar precision and recall ratios. Considering that this entire process needs to run in a stream processing way, the calculation cannot be too complex. We finally choose this simple but efficient outlier analysis mechanism. This correlating diagnosis helps us troubleshoot various malfunctioning devices, including decelerating GPU/CPU/PCIe/NIC/link/switch, which covers a large proportion of failures in production.

Case study. We use an actual case met in production to further illustrate how the cross-host correlation diagnosis runs. During our in-house LLM model's training, as shown in Figure 8, the ECN statistic from one NIC increases from zero to 10-30K per second. At the same time, our model training team reports a 26% training iteration time increase. The correlating diagnosis mechanism immediately identifies this abnormal situation, since this abnormal NIC exceeds $\lambda + 2\delta$ of the ECN metric. The root cause is that one of the links connecting to this NIC drops packets silently. It triggers all traffic forward to this NIC through another link, introducing network congestion at the last hop and finally delaying the entire training iteration. Correlating diagnosis immediately pinpoints the root cause of this performance degradation. After isolating the host containing this abnormal NIC and restarting the training task, the training performance returns to normal.

Limitation. This correlating diagnosis can work out the root cause for performance degradation cases in which one or several hosts generate significantly different metrics compared with others. However, this is not always true. It is also common that when performance degradation happens, several metrics change on all hosts, and we are unable to pinpoint the root cause. Therefore, we need to acquire more information and a new mechanism to solve these hard cases.

5.2 Enhancing Procedure-aware Diagnosis

Inspired by the design choice in §4.2, we choose to further customize CCL for more information helping degradation diagnosis. We further record the following statistics of each collective operator (C_i) for each GPU (G_j) in iteration I_k.

- In I_k, the duration of C_i in G_j is $TD_{i,j,k}$.
- In I_k, average duration of C_i is $\overline{TD}_{i,k}$.
- In I_k, the network throughput for the last L ($L = 5$ in practice) work requests of C_i in G_j is $N_{i,j,k}$.
- In I_k, average network throughput of C_i is $\overline{N}_{i,k}$.

Figure 9a represents the computation degradation case, where the unexpectedly long duration of the computation is the culprit of performance degradation. Since the end of each collective operation is synchronized, we can utilize the duration of communication time to deduce the computation time. If the $TD_{i,j,k} < \alpha\overline{TC}_{i,k}$ ($\alpha = 0.8$ in practice), then G_j is the root cause of the computation degradation. Figure 9b represents the communication degradation case, where the long duration of the communication is the culprit of performance degradation. If $N_{i,j,k} > \beta\overline{N}_{i,k}$ ($\beta = 1.5$ in practice), there is a communication degradation. We use a slack threshold here to resist noise caused by possible temporary network congestion. Based on this information, we filter out GPUs group $\mathbb{G}$ directly suffering degradation. $\mathbb{G}$ is further used to determine which source or destination is the root cause of this communication degradation. The principle of this procedure is similar to *RootDiag()* in Algorithm 1 in Appendix A. We omit the details due to limited space.

6 Solving Problems Before Delivery

We analyze the runtime duration of all failed training tasks. As shown in Figure 10, 73% of tasks failed within the first 10 minutes, which is unexpected since the initialization phase of a training task usually takes between 5 to 20 minutes. It indicates that many tasks are failing during the initialization process. It is also in alignment with our experience: many components (both software and hardware) may already endure errors before engaging in the new training task. We have reviewed all failures that occurred during the initialization phase, and there are two main reasons for these failures.

Table 1: CBD task list

Phase	Tasks	Time
Configuration check in parallel	Host configuration check	<1min
	GPU configuration check	
	NIC configuration check	
Single-host test in parallel	GPU kernels test	3min
	NVLink test	
	HBM test	
	PCIe test	
	CPU execution test	
	Dataset/Model/Checkpoint load test	
Multi-hosts test in parallel	Collective communication test	6min
	Comput./Comm. overlap test	

Frequent component updates. Components such as training frameworks, CCL, container networks, NIC drivers, and switches are updated frequently. For example, as released by Meta [19], during the training of LLaMa3, 47 planned updates are executed during a 54-day training snapshot (i.e., 26 times per month). In our model training cloud service, we have a bunch of update requirements for many purposes (including bugfix, safety reinforcements, version unifying and performance optimizations) from many components. To keep the entire service steady, we do not execute updates at such a high frequency. Critical updates, including bugfix and safety reinforcements, are merged to be released weekly. Other updates are released monthly. Even so, updates still trigger many failures in production.

Post-usage failures. If a host encounters a fault after its last use, it will trigger the failure of the new training task during the initialization phase. This issue is especially challenging in cloud environments where hosts are dynamically allocated from a shared resource pool, making failure reproducing much harder. Additionally, once a host is delivered to the customer, the cloud provider cannot run diagnostics on it arbitrarily, making diagnosis much more difficult.

To address these issues, we introduce the Check Before Delivery (CBD) procedure. This check is done right before the resources are handed over to the customers. It brings two main advantages. First, this check process is added at the final stage of resource delivery, so it does not disrupt the existing workflow. Second, by running the validation after the entire environment is set up, we can catch more issues. For example, a connectivity test executed on the physical host will miss connectivity problems caused by incorrect routing configurations in the container network. These issues can only be detected after the container is completely created.

However, CBD has a drawback. Since CBD is called after the environment is fully prepared, it must be in the last phase of delivery. This means CBD needs to be efficient to avoid impacting user experience. Taking these factors into consideration, we designed a set of CBD operations (as outlined in Table 1). We comprehensively select representative tests that cover the main components and organize them to execute in parallel. The entire execution of CBD is less than 10 minutes. If a large number of machines fail in the CBD procedure

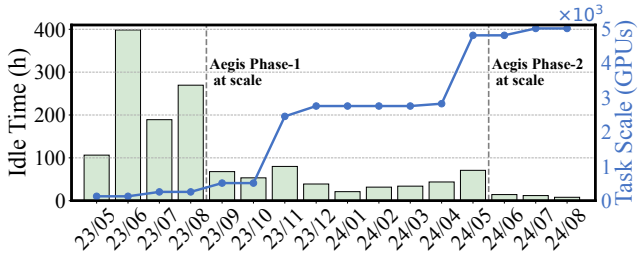


Figure 11: Evolution of idle time in production.

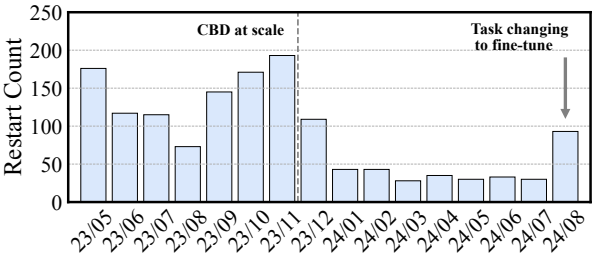


Figure 12: Evolution of restart counts in production.

(reaching a certain threshold), we roll back recent updates to prevent widespread service disruption. There are several different sale modes of the model training service. For the complete PaaS mode, introducing an extra 10 minutes before the startup of each single training task is still unbearable. We also deliver a lightweight CBD version, which only contains parallelized configuration checks and other critical quick local host tests. This lightweight CBD can be completed within 1 minute and can cover most fundamental failures.

With the deployment of CBD, we have intercepted 1-2% problematic hosts before final delivery. If not detected on time, these problematic hosts would lead to training task failures. Considering this significant benefit, we have made CBD a mandatory procedure in delivery.

7 Evaluation: Aegis in Production

The first version of Aegis was online in September 2023, and it immediately became one of the most fundamental components in all succeeding delivery of training services. Aegis has served dozens of large-scale training clusters for more than one year. Due to confidential reasons, we cannot release statistics from our external customers. Therefore, we statistic training task information from our inner model training team, which works to train one of the top-tier LLMs (i.e., large language model) in the world.

7.1 Evolution of Training Stability

As shown in Figure 11, the line records the scale of the training tasks from our inner model training team, which increases by more than 40× during the last 16 months. The bars represent the monthly accumulated idle time of the training tasks owing to the waiting for the failure diagnosis. After Aegis Phase-1 is online, the idle time (time duration where no task running in the training cluster) is decreased by 71% in the next month. This result is impressive, considering that we even doubled the training scale in September 2023. There is

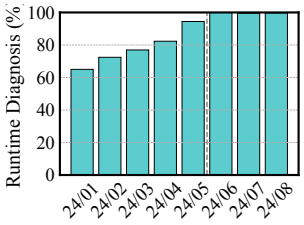


Figure 13: Runtime diagnosis percentage.

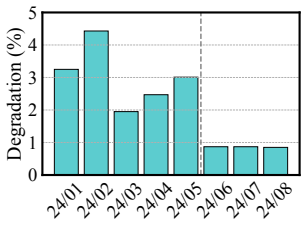


Figure 14: Performance degradation percentage.

an increase of idle time in November 2023, which is owning to a 4× boosting of training scale. The increase in the training scale introduces several unexpected corner case issues, which consume a long time for diagnosis and affect idle time. Aegis Phase-2 is deployed in June 2024, directly leading to 91% save of the training idle time. This improvement mainly comes from the fact that more failures can be solved without involving offline diagnosis.

Figure 12 shows the restart counter of the training task. The increase of the training task scale in November 2023 also triggers more training restart, and a large amount of failures during the initialization phase. We, therefore, speed up the development of CBD and put it into online service in December 2023. It contributes 44.8% decrease in the restart counter in the next month. Through continuously handling more cases and optimizing the checklist, it finally decreased 84.6% of the restart counter. With CBD fully deployed and comprehensively optimized, it discovers around 1-2% problematic hosts before final delivery. Note that the restart number increases in August 2024, the reason is that our model training team switched the task from pre-train to fine-tuning, introducing planned experiments and tests. The root causes of these errors are diverse (e.g., device aging, incorrect configurations, wiring mistakes caused by repairs, complex serving mode switching, etc.). Moreover, since we serve model training requests from multiple tenants, not all tasks are mature and steady-running. Many failures are caused by unoptimized LLM designs or incorrect use of the training infrastructure.

7.2 Runtime Failure Diagnosis

We further answer how many failure cases are diagnosed in runtime (rather than offline) in Figure 13. This metric is important since each offline diagnosis would introduce great damage to the overall GPU utilization and user experience. With the deployment of Aegis Phase-2, the runtime diagnose percentage gradually converges to near 100%. It means that a training task can automatically recover from almost all types of failures without human interference.

7.3 Handling Performance Degradation

To quantify the efficiency of performance degradation diagnosis of Aegis, we deeply cooperate with our model training team to acquire records of iteration time in all training tasks. The iteration time T_k is measured from our model training log. The standard iteration time is calculated as $T_S = 1.2 \times \overline{T_k}$. Performance degradation is calculated as $\frac{\sum_{k:T_k>T_S}(T_k - T_S)}{\sum_{all} T_k}$. The statistic results are shown in Figure 14. Performance degradation diagnosis of Aegis is deployed in June 2024. It significantly eliminates performance degradation by 71%.

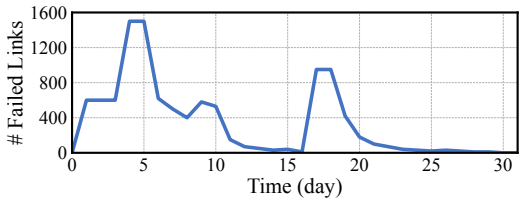


Figure 15: Number of failed links during the batch failure.

8 Experience and Lessons

We share key experiences and lessons we learned during the evolution of Aegis as well as the entire reliability guarantee for model training cloud service (additional experience can be found in Appendix B).

Handling batch link failure cases. We have built and delivered dozens of large-scale training clusters. At the delivery of one cluster in them, we encounter an unbearable increase in the link failure ratio ($10-20\times$ higher than the normal link failure ratio). Our investigation revealed that the primary cause was overlapping construction timelines for the data center's building infrastructure and the installation of servers and network wiring. This overlap resulted in significant contamination of optical modules and fibers.

Figure 15 illustrates the change in failure rates over a month. With new machines delivered in batches, the number of failed links increases swiftly due to contaminated links. As cleaning efforts progressed, the failure rate decreased gradually. It takes tens of deep cleaning to completely solve this contamination problem. After identifying this problem, we implemented stricter guidelines for data center construction and delivery to prevent similar issues in the future.

Multiple sale modes and heterogeneous devices in production. Another significant factor affecting the stability of training clusters is the complexity of delivery scenarios. The constant updates to top-tier GPUs (e.g., NVIDIA A100 [13], H100 [16]), NICs (e.g., NVIDIA CX-6 [6], CX-7 [15], BF3 [14]), and switches have led to the deployment of a wide variety combinations of heterogeneous devices in our clusters. This diversity in hardware configurations adds complexity to both the testing and delivery processes, as well as to identifying the root causes of stability issues. Furthermore, to satisfy the diverse needs of our customers, we offer different sales models. For customers focused solely on model optimization, we provide the Platform-as-a-Service (PaaS) sale mode. Therefore, these customers only need to care about constructing novel models and managing their Docker images. On the other hand, for customers who seek further optimization that combines models, training frameworks, and infrastructure, we offer a basic Infrastructure-as-a-Service (IaaS) sale mode. These sale modes involve significant differences in the technology stacks, such as how devices are virtualized and

how the entire cluster is managed. Each mode has its unique set of configurations and delivering procedures, greatly impacting the overall reliability.

Furthermore, in practical operations, we often need to transfer machines originally used in one environment to another, such as relocating servers from one cluster to another with a completely different network architecture or converting a cluster from IaaS to PaaS. These transitions involve numerous configuration changes. Initially, we directly reconfigure all hosts according to the new scenario. However, we encounter several issues where the configurations from the previous setup are not fully compatible with the new one. For example, some modules lack initialization options and only have the overwrite function (e.g., NIC firmware). If the new configurations did not completely overwrite the old ones, residual configurations could remain active, leading to potential issues.

To address these problems, we devote significant effort to thoroughly reviewing and rewriting all configurations to ensure compatibility under different scenarios. By adding a corresponding configuration check in CBD, we ultimately avoid such problems from occurring again.

Traffic pattern evolution triggers the congestion control issue. During the continuous delivery of new training clusters, we encounter a significant issue, where the iteration time increased significantly during the training. Through comprehensive correlation analysis, Aegis identifies the problem but does not find out any abnormal indicators on individual hosts. Further Aegis's runtime analysis based on collective communication information reveals that the performance degradation was due to communication delays caused by many hosts. Despite isolating the abnormal hosts, the training performance can recover, but soon, a similar issue occurs again.

Initially, we suspect that the issue might be related to some faulty switches, since network problems are typically the cause of persistent communication performance issues. However, our thorough investigations show no abnormalities in switch behavior. To resolve this, we conduct extensive offline reproduction and testing. We eventually pinpointed the cause: a bug in the congestion control implementation of NICs. The bug causes an issue where a small number of continuous ECN signals could cause the NIC to enter a preset maximum rate limit. This rate limit is set very low, resulting in significant communication slowdowns.

We report this issue to the NIC vendor, who also confirms this bug. Given that a firmware update to fix the issue would take a considerably long time, we implement two immediate patches in production. (1) We raise the maximum speed limit to alleviate the performance impact. (2) We employ periodic resets of the congestion control state to prevent prolonged performance degradation. We later fix this issue permanently by updating the official bugfix firmware.

Protecting customers' privacy. It is always an important topic for cloud service providers to keep customers' privacy (e.g., training dataset, customized models, training strategies and model checkpoints) well protected. Aegis Phase-1 builds the fault diagnosis ability fully based on existing statistics, introducing no privacy risk. During the design of Aegis Phase-2, to achieve further runtime information for better diagnosis, we investigate and attempt a series of different methods. We have had multiple times of conversations with our product development, solution architect and solution sales teams to figure out the best solution path for Aegis Phase-2. Although for those heavy solutions (e.g., encoding specific statistic functions in customers' models), we can force the deployment of these solutions by re-signing authorization agreements with customers, these solutions would be rejected by most of our customers. After extensive design deliberation, we ultimately opt to enhance CCL.

9 Related Work

Large-scale AI model training diagnosis systems. SuperBench [60] provides a comprehensive benchmark suite for diagnosing issues before deployment, which cannot cover runtime failures. MegaScale [36] monitors CUDA events of "critical code segments" in the model code, which is impractical from the perspective of the cloud service provider. There are some other diagnosis solutions based on infrastructure logs and statistics like monitor_train_log [2] and SageMaker [11], which can only cover a limited range of failures. Dynolog [10] integrates PyTorch profiler to further get code-block level tracing, which however cannot locating root causes in the GPU kernel executions. We have tried this technology roadmap before, but owing to the limited failure coverage, we eventually chose to not deploy them in production.

General abnormality diagnosis systems. In the general cloud computing scenario, various efforts have been devoted to anomaly diagnosis in the host [23–25, 28, 31, 34, 35, 39, 42, 43, 46–48, 50, 52, 55, 57] or in the network [26, 29, 32, 35, 37, 38, 40, 41, 51, 53, 54, 56, 58, 59, 61, 63, 64]. However, as illustrated in §2, without specific optimization focusing model training scenarios, these solutions cannot provide satisfied diagnosing precision for model training specific failure cases.

10 Conclusion

We present the evolution of Aegis, a large-scale model training diagnosis system, in our production. Keeping easy-to-deploy in the cloud as the first principle, Aegis solves training failures and performance degradation cases. Aegis decreases 97% of the idle time wasted by failure diagnosis, 84% of the training task restarts, and 71% of the performance degradation.

Acknowledgements

We acknowledge all teams within Alibaba Cloud that contributed to the success of Aegis, including the High-Performance Network, PAI, Lingjun, Network Automation, Network Operation, Network Systems and Optical Network teams, to name a few. We also thank our shepherd Peng Zhang, and the NSDI reviewers for their insightful comments. Ennan Zhai is the corresponding author.

References

[1] NVIDIA DGX SuperPOD: Next Generation Scalable Infrastructure for AI Leadership. https://docs.nvidia.com/https:/docs.nvidia.com/dgx-superpod-reference-architecture-dgx-h100.pdf, 2023.

[2] OPT-175 Logbook. https://github.com/facebookresearch/metaseq/blob/main/projects/OPT/chronicles/OPT175B_Logbook.pdf, 2023.

[3] Anomaly detection using Isolation Forest – A Complete Guide. https://www.analyticsvidhya.com/blog/2021/07/anomaly-detection-using-isolation-forest-a-complete-guide/, 2024.

[4] CLIP. https://github.com/openai/CLIP, 2024.

[5] Configuring Syslog. https://www.cisco.com/c/en/us/td/docs/switches/metro/me1200/controller/guide/b_nid_controller_book/b_nid_controller_book_chapter_010101.pdf, 2024.

[6] ConnectX-6. https://www.nvidia.com/en-sg/networking/ethernet/connectx-6/, 2024.

[7] DBSCAN vs. K-Means: A Guide in Python. https://www.newhorizons.com/resources/blog/dbscan-vs-kmeans-a-guide-in-python, 2024.

[8] DeepSpeed. https://www.microsoft.com/en-us/research/project/deepspeed/, 2024.

[9] dmesg(1) — Linux manual page. https://man7.org/linux/man-pages/man1/dmesg.1.html, 2024.

[10] Dynolog: a performance monitoring daemon for heterogeneous CPU-GPU systems. https://github.com/facebookincubator/dynolog, 2024.

[11] Machine Learning Service - Amazon SageMaker. https://aws.amazon.com/pm/sagemaker/?nc1=h_ls, 2024.

[12] Mixtral-8x7B. https://huggingface.co/mistralai/Mixtral-8x7B-Instruct-v0.1, 2024.

[13] NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/, 2024.

[14] NVIDIA BLUEFIELD-3 DPU PROGRAMMABLE DATA CENTER INFRASTRUCTURE ON-A-CHIP. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2024.

[15] NVIDIA CONNECTX-7 400G ETHERNET SMART ACCELERATION FOR CLOUD, DATA-CENTER AND EDGE. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf, 2024.

[16] NVIDIA H100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/h100/, 2024.

[17] NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/, 2024.

[18] Ten Reasons to Use Passive DAC Cables in Your Data Center. https://vitextech.com/ten-reasons-to-use-passive-dac-cables-in-your-data-center/, 2024.

[19] The Llama 3 Herd of Models. https://ai.meta.com/research/publications/the-llama-3-herd-of-models/, 2024.

[20] Vision Transformer (ViT). https://huggingface.co/docs/transformers/model_doc/vit, 2024.

[21] Xid Errors. https://docs.nvidia.com/deploy/xid-errors/index.html, 2024.

[22] Z-Score: Meaning and Formula. https://www.investopedia.com/terms/z/zscore.asp#:~:text=Z%2Dscore%20is%20a%20statistical,traders%20to%20help%20determine%20volatility, 2024.

[23] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Liu, Jitu Padhye, Geoff Outhred, and Boon Thau Loo. Closing the network diagnostics gap with vigil. In Proceedings of the SIGCOMM Posters and Demos, SIGCOMM Posters and Demos '17, page 40–42, New York, NY, USA, 2017. Association for Computing Machinery.

[24] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 419–435, Renton, WA, April 2018. USENIX Association.

[25] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpoirot. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, page 440–453, New York, NY, USA, 2016. Association for Computing Machinery.

[26] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and

Protocols for Computer Communication, SIGCOMM '20, page 662–680, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: identifying density-based local outliers. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD '00, page 93–104, New York, NY, USA, 2000. Association for Computing Machinery.

[28] Rui Ding, Xunpeng Liu, Shibo Yang, Qun Huang, Baoshu Xie, Ronghua Sun, Zhi Zhang, and Bolong Cui. Rd-probe: Scalable monitoring with sufficient coverage in complex datacenter networks. In Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM '24, page 258–273, New York, NY, USA, 2024. Association for Computing Machinery.

[29] Chongrong Fang, Haoyu Liu, Mao Miao, Jie Ye, Lei Wang, Wansheng Zhang, Daxiang Kang, Biao Lyv, Peng Cheng, and Jiming Chen. Vtrace: Automatic diagnostic system for persistent packet loss in cloud-scale overlay network. In Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, page 31–43, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Junzhi Gong, Yuliang Li, Bilal Anwer, Aman Shaikh, and Minlan Yu. Microscope: Queue-based performance diagnosis for network functions. SIGCOMM '20, page 390–403, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15, page 139–152, New York, NY, USA, 2015. Association for Computing Machinery.

[32] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15, page 139–152, New York, NY, USA, 2015. Association for Computing Machinery.

[33] Zhenyu Guo, Dong Zhou, Haoxiang Lin, Mao Yang, Fan Long, Chaoqiang Deng, Changshu Liu, and Lidong Zhou. G2: A graph processing system for diagnosing distributed systems. In 2011 USENIX Annual Technical Conference (USENIX ATC 11), Portland, OR, June 2011. USENIX Association.

[34] Roni Haecki, Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Bo Gan, Timothy Merrifield, Sujata Banerjee, and Timothy Roscoe. How to diagnose nanosecond network latencies in rich end-host stacks. In 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), pages 861–877, Renton, WA, April 2022. USENIX Association.

[35] Vipul Harsh, Wenxuan Zhou, Sachin Ashok, Radhika Niranjan Mysore, Brighten Godfrey, and Sujata Banerjee. Murphy: Performance diagnosis of distributed cloud applications. In Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM '23, page 438–451, New York, NY, USA, 2023. Association for Computing Machinery.

[36] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), pages 745–760, Santa Clara, CA, April 2024. USENIX Association.

[37] Pravein Govindan Kannan, Nishant Budhdev, Raj Joshi, and Mun Choon Chan. Debugging transient faults in data centers using synchronized network-wide packet histories. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), pages 253–268. USENIX Association, April 2021.

[38] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 421–436, Boston, MA, February 2019. USENIX Association.

[39] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in RDMA subsystems. In 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), pages 287–305, Renton, WA, April 2022. USENIX Association.

[40] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies, CoNEXT '16, page 481–495, New York, NY, USA, 2016. Association for Computing Machinery.

[41] Kefei Liu, Zhuo Jiang, Jiao Zhang, Shixian Guo, Xuan Zhang, Yangyang Bai, Yongbin Dong, Feng Luo, Zhang Zhang, Lei Wang, Xiang Shi, Haohan Xu, Yang Bai, Dongyang Song, Haoran Wei, Bo Li, Yongchen Pan, Tian Pan, and Tao Huang. R-pingmesh: A service-aware roce network monitoring and diagnostic system. In Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM '24, page 554–567, New York, NY, USA, 2024. Association for Computing Machinery.

[42] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. Hostping: Diagnosing intra-host network bottlenecks in RDMA servers. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 15–29, Boston, MA, April 2023. USENIX Association.

[43] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, page 129–143, New York, NY, USA, 2016. Association for Computing Machinery.

[44] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), pages 353–366, San Jose, CA, April 2012. USENIX Association.

[45] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. Alibaba hpn: A data center network for large language model training. In Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM '24, page 691–706, New York, NY, USA, 2024. Association for Computing Machinery.

[46] Mubashir Adnan Qureshi, Junhua Yan, Yuchung Cheng, Soheil Hassas Yeganeh, Yousuk Seung, Neal Cardwell, Willem De Bruijn, Van Jacobson, Jasleen Kaur, David Wetherall, and Amin Vahdat. Fathom: Understanding datacenter application network performance. In Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM '23, page 394–405, New York, NY, USA, 2023. Association for Computing Machinery.

[47] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren. Passive realtime datacenter fault detection and localization. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 595–612, Boston, MA, March 2017. USENIX Association.

[48] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, Mingwei Xu, Yahui Li, Jiping Yin, Jianchang Song, Zhuofeng Li, and Runjie Nie. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code. In Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM '23, page 420–437, New York, NY, USA, 2023. Association for Computing Machinery.

[49] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. CoRR, abs/1909.08053, 2019.

[50] Ya Su, Youjian Zhao, Chenhao Niu, Rong Liu, Wei Sun, and Dan Pei. Robust anomaly detection for multivariate time series through stochastic recurrent neural network. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD '19, page 2828–2837, New York, NY, USA, 2019. Association for Computing Machinery.

[51] Haifeng Sun, Jiaheng Li, Jintao He, Jie Gui, and Qun Huang. Omniwindow: A general and efficient window mechanism framework for network telemetry. In Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM '23, page 867–880, New York, NY, USA, 2023. Association for Computing Machinery.

[52] Ming Sun, Ya Su, Shenglin Zhang, Yuanpu Cao, Yuqing Liu, Dan Pei, Wenfei Wu, Yongsu Zhang, Xiaozhou Liu, and Junliang Tang. Ctf: Anomaly detection in high-dimensional time series with coarse-to-fine model transfer. In IEEE INFOCOM 2021 - IEEE Conference on Computer Communications, pages 1–10, 2021.

[53] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with PathDump. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pages 233–248, Savannah, GA, November 2016. USENIX Association.

[54] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging

with SwitchPointer. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 453–456, Renton, WA, April 2018. USENIX Association.

[55] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. NetBouncer: Active device and link failure localization in data center networks. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 599–614, Boston, MA, February 2019. USENIX Association.

[56] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. NetBouncer: Active device and link failure localization in data center networks. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 599–614, Boston, MA, February 2019. USENIX Association.

[57] Junxiao Wang, Heng Qi, Yang He, Wenxin Li, Keqiu Li, and Xiaobo Zhou. Flowtracer: An effective flow trajectory detection solution based on probabilistic packet tagging in sdn-enabled networks. IEEE Transactions on Network and Service Management, 16(4):1884–1898, 2019.

[58] Weitao Wang, Xinyu Crystal Wu, Praveen Tammana, Ang Chen, and T. S. Eugene Ng. Closed-loop network performance monitoring and diagnosis with SpiderMon. In 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), pages 267–285, Renton, WA, April 2022. USENIX Association.

[59] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: Diagnosing performance problems with temporal provenance. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 395–420, Boston, MA, February 2019. USENIX Association.

[60] Yifan Xiong, Yuting Jiang, Ziyue Yang, Lei Qu, Guoshuai Zhao, Shuguang Liu, Dong Zhong, Boris Pinzur, Jie Zhang, Yang Wang, Jithin Jose, Hossein Pourreza, Jeff Baxter, Kushal Datta, Prabhat Ram, Luke Melton, Joe Chau, Peng Cheng, Yongqiang Xiong, and Lidong Zhou. SuperBench: Improving cloud AI infrastructure reliability with proactive validation. In 2024 USENIX Annual Technical Conference (USENIX ATC 24), pages 835–850, Santa Clara, CA, July 2024. USENIX Association.

[61] Kaicheng Yang, Yuhan Wu, Ruijie Miao, Tong Yang, Zirui Liu, Zicang Xu, Rui Qiu, Yikai Zhao, Hanglong Lv, Zhigang Ji, and Gaogang Xie. Chamelemon: Shifting measurement attention as network state changes. In Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM '23, page 881–903, New York, NY, USA, 2023. Association for Computing Machinery.

[62] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), pages 293–306, Hollywood, CA, October 2012. USENIX Association.

[63] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow event telemetry on programmable data plane. In Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, page 76–89, New York, NY, USA, 2020. Association for Computing Machinery.

[64] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15, page 479–491, New York, NY, USA, 2015. Association for Computing Machinery.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A Failure Diagnosis Procedure

Algorithm 1 shows the pseudo-code of the entire Failure Diagnosis Procedure.

B Additional Experience

Dual-ToR really helps, but it migrates failure cases to degradation cases. As aforementioned, we deploy a dual-ToR design in production to conquer the high link failure ratio. This design is very effective. We handle dozens of link-repairing tickets every week, but most of them do not lead to the crash of training tasks. Furthermore, thanks to dual-ToR, we can conduct a hotfix for these faulty links without isolating the host (i.e., the training task can train continuously during the hotfix). We deliver a complete Standard Operating Procedure (SOP) for resident staff in the cluster to conduct the hotfix. They need to check (1) the correctness of both ports' links, (2) the status of both links, and (3) the switch ports'

Algorithm 1 Failure Diagnosis Procedure

T //The set containing all hosts used in this training
$L_{CE} = CriticalError(T)$//get critical error locations
$L_{DE} = DistError(T)$//get distributed error locations
if $L_{CE}.size() > 0$ **then**
    $Isolate(L_{CE})$ & Restart the task
else if $L_{DE}.size() <= 2$ **then**
    $Isolate(L_{DE})$ & Restart the task
else
    $N = RootDiag(L_{DE})$
    if $N! = NULL$ **then**
        $CheckError(N)$
        $Isolate(N)$ & Restart the task
    else//distributed problem
        $R_C = ConfigCheck(L_{DE})$
        $R_N = NetDiag(L_{DE})$
        if $R_C || R_N! = NULL$ **then**
            $Repair(R_C, R_N)$
        else
            Isolate L_{DE} & $OfflineDiag(T)$
        end if
    end if
end if

status before conducting the real link replacement. More than $O(10K)$ link hotfixs are successfully executed.

Although dual-ToR can significantly decrease the possibility of training crashes caused by link failure, it transfers these failure cases into performance degradation cases (since the available network bandwidth is halved). As a result, we devote more effort to handling performance-related issues. As the training does not crash, actually customers can independently determine whether to proactively terminate the current training right after a new checkpoint and restart it, greatly improving the overall user experience.

Reboot or repair? As illustrated in Figure 2, most stability issues ultimately trace back to problems with the fault in hosts. Once a faulty host is isolated, a critical next step is determining whether it can be fixed by reboot or repair. Some issues can be fixed with a reboot, but others necessitate sending the host for repair, which can take several weeks or even months. The primary challenge is that error reports alone cannot definitively indicate whether a problem can be resolved by reboot. For example, an ECC Error could be caused by an unexpected bit flip, which might be fixed by a reboot. On the other hand, an ECC Error could indicate a hardware fault in the HBM, which must be repaired by sending the host back to the vendor.

Given the lack of an efficient method to determine whether the host needs reboot or repair, we have developed a SOP for it. After isolating the faulty host, we first perform a reboot and then conduct a series of hardware stress tests. Unlike CBD, since the host is already isolated, we need a comprehensive (rather than fast) test to make sure the host is back to normal. We first check the current installation settings. If any irregularities are found, the machine is sent to execute the complete re-installation. If all settings are correct, the host further undergoes a thorough computing power, CPU, GPU, Memory, PCIe, NVLink and network check. If all checks pass, the host is returned to the online cluster pool. Otherwise, the host is sent for repair.

Despite the above rigorous tests, we still occasionally encounter corner cases where a host that passes all tests still exhibits faulty issues. To address this, we have implemented an additional backstop: if a host passes the reboot tests three times, but continues to cause failures online, it is mandatory to send for repair. We also engage with our vendors to understand the root cause and update our test cases accordingly. Accurately and efficiently determining whether a host requires repair remains an open issue. Our current approach is practical for production but still leaves room for further refinement and improvement.