SkeletonHunter: Diagnosing and Localizing Network Failures in Containerized Large Model Training

Wei Liu[1,2], Kun Qian[2], Zhenhua Li[1*], Tianyin Xu[3], Yunhao Liu[1], Weicheng Wang[2]
Yun Zhang[2], Jiakang Li[2], Shuhong Zhu[2], Xue Li[2], Hongfei Xu[2], Fei Feng[2], Ennan Zhai[2*]
[1]Tsinghua University [2]Alibaba Cloud [3]UIUC

Abstract

The flexibility and portability characteristics have made containers a popular serverless environment for large model training in recent years. Unfortunately, these advantages render the network support for containerized large model training extremely challenging, due to the high dynamics of containers, the complex interplay between underlay and overlay networks, and the stringent requirements on failure detection and localization. Existing data center network debugging tools, which rely on comprehensive or opportunistic monitoring, are either inefficient or inaccurate in this setting.

This paper presents SkeletonHunter, a container network monitoring and diagnosis system that leverages the intrinsic and regular sparsity of the network traffic incurred by large model training. Its key idea is to reason about the *traffic skeleton*, which comprises a crucial set of network paths consistently traversed by the training traffic, so as to reliably detect and localize network failures in short time. We deployed it in production for six months, uncovering 4,816 network failures with 98.2% precision and 99.3% recall, and localizing them with a high accuracy of 95.7%. After fixing 98% problematic network components, the monthly network failure rate has significantly dropped by 99.1%.

CCS Concepts

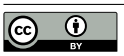
• **Networks** → **Data center networks**; **Network monitoring**; **Error detection and error correction**; Network dynamics.

Keywords

Containerized Large Model Training; AI Infrastructure; Serverless Computing; Large-scale Network Monitoring and Troubleshooting; Network Reliability; Traffic Pattern Analysis

ACM Reference Format:
Wei Liu, Kun Qian, Zhenhua Li, Tianyin Xu, Yunhao Liu, Weicheng Wang, Yun Zhang, Jiakang Li, Shuhong Zhu, Xue Li, Hongfei Xu, Fei Feng, Ennan Zhai. 2025. SkeletonHunter: Diagnosing and Localizing Network Failures in Containerized Large Model Training. In *ACM SIGCOMM 2025 Conference (SIGCOMM '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3718958.3750513

*Corresponding authors.



SIGCOMM '25, Coimbra, Portugal

ACM ISBN 979-8-4007-1524-2/2025/09
https://doi.org/10.1145/3718958.3750513

1 Introduction

Large model training is emerging as an important business for cloud service providers (CSPs) [45, 50, 67]. It is typically launched in either physical clusters or flexible containers in modern data centers. While physical clusters can be fully utilized by professional customers, they are not suited to common users for technical difficulties and resource wastes. In contrast, containerized model training [10, 11, 15] has become popular in recent years together with the advancement of serverless computing [74], due to its lightweight, portability, and isolation characteristics [57]. Specifically, containerized model training allows users to launch model training tasks over shared RDMA NICs (RNICs) and GPUs through containers that bundle training code, data, and dependencies. As a mainstream CSP, we have been operating a large-scale containerized model training cloud with 40K+ RNICs and 40K+ GPUs for over three years, serving ~5M training tasks from users.

At such a large scale, the reliability of the network infrastructure, especially the connectivity among training containers, is crucial to the service quality. In a typical setting, an RDMA over Converged Ethernet (RoCE) network is expected to have a round trip time (RTT) of less than 20μs [35, 46, 51] with zero packet loss for high-performance training. This is because the training tasks are highly synchronous—even a 10μs increase in RTT can lead to a ~20% slowdown in the training process [67]. In the worst case, when a connectivity issue lasts for longer than 4s, the collective communication will time out and thus fail the entire training task [16].

Unfortunately, our long-term operational experience shows that accurately and timely pinpointing connectivity issues for large-scale containerized model training infrastructure is confronted with three-fold major challenges as follows.

- **High Dynamics of Containers.** Most containers have short life cycles, and grouped containers (that belong to the same task) have asynchronous running states. Our statistics show that over 50% of the containers have a lifetime of less than 60 minutes; in contrast, the lifetime of a physical host is usually as long as months to years. Worse still, even grouped containers suffer a couple of minutes of time lag in state synchronization. Both factors make the container network structure frequently change.
- **Endpoint-Induced Complexity.** A container in production can bind to multiple (e.g., eight) RNICs for adapting to different patterns of training workloads, i.e., adaptive parallelism. The bound pair of a container and an RNIC is termed as an *endpoint*. This induces more complexity to connectivity monitoring, since all endpoints now should be covered to ensure a reliable network.
- **Interplay between Overlay and Underlay.** Containerized model training infrastructure is typically shared among numerous tenants to concurrently train models, thus involving an

additional *overlay* network atop the physical network for training resource/performance isolation. This, however, renders the instances of virtual network components (e.g., container network interface [6], virtual switches [66], and RNIC flow tables [54]) significantly more than those in traditional data centers.

Most importantly, these unique features of containerized model training bring a *multiplicative effect* to the difficulty of troubleshooting connectivity issues in the network. Suppose there are X containers involved in a training task, each container is bound to an average of Y RNICs, and each RNIC is associated with an average of Z virtual network components. Then, we need to examine $X \times Y \times Z$ (e.g., 1K × 8 × 16 = 128K) network components in each training round (i.e., an iteration of model training, typically ~30s), which is impossible to achieve in practice.

Existing solutions for diagnosing data center networks either *comprehensively* monitor all network packets [34, 87, 89], or *opportunistically* sample the network traffic [32, 58, 71]. Neither applies to our scenario. The former requires heavy modifications to network infrastructure—although today's commodity switches provide some hardware supports for full-traffic monitoring (e.g., packet mirroring [69]), monitoring virtual network components such as software switches is heavyweight and incurs nontrivial performance degradation. The latter uses various strategies to sketch the network, but are not guaranteed to cover all the critical paths leading to connectivity issues, and thus incur false negatives.

In this paper, we address the above challenges based on a key insight—there is intrinsic and regular *sparsity* hidden in the network traffic incurred by large model training. This is attributed to the wide usage of the *collective communication* paradigm in large model training. Supported by sophisticated libraries like UCC [21], NCCL [20], MSCCL [18], and MPI [44], this paradigm has now become a de facto standard in industry due to its effectiveness and efficiency. As a result, the actual connectivity only exists among the endpoints located at the same parallelism group [67], therefore substantially and safely reducing the scrutinizing scope.

As a CSP, however, practically using the insight is hindered by our invisibility into the tenants' model composition, so we cannot directly figure out the desired sparsity information. Instead, we resort to a more pragmatic approach by *inferring* the *traffic skeleton* that essentially connects all the active endpoints during a large model training task. In other words, the skeleton comprises a crucial set of network paths that the training traffic consistently traverses.

Specifically, we identify each crucial network path based on the unique patterns of periodic traffic bursts that manifest on active endpoints of the training containers. This phenomenon originates from an imperative operation during large model training, i.e., the iterative synchronization of enormous parameters in each training round. With the inferred traffic skeleton information, we build an optimal probing matrix among the endpoints to effectively detect network failures with minimum resource utilization.

For a detected failure between a pair of endpoints, we localize the problematic network component(s) through *optimistic overlay-underlay disentanglement*, which separately examines the network components along the path between the two endpoints at each layer, assuming that the other layer is healthy. The optimism comes from our observation that the root causes at overlay and underlay layers are usually software- and hardware-related respectively, which will not propagate to each other. If we cannot find any problematic components at each layer, we fine-check the RNICs that connect the two layers, which requires a lot of manual efforts and may induce temporary network performance degradation.

We implement the above design into a practical system dubbed SkeletonHunter, which has been deployed in our production containerized model training cloud for 10+ months (since Mar. 2024). Using only two middle-end backend servers, it continuously collects, filters, and aggregates connectivity data from 40K+ endpoints that belong to ~2K concurrent training tasks, based on which it reliably detects and localizes network failures in a short time (8s on average, which is essentially shorter than 30s, the typical duration of a training round).

SkeletonHunter helped us discover 4,816 network failures during Mar.–Aug. 2024 with 98.2% precision and 99.3% recall, and localize them to 1,302 problematic network components with a high accuracy of 95.7%. We carefully analyze the false cases, finding that they mainly derive from intra-host connectivity issues like the GPU-to-NIC NVLink and PCIe buses. Such issues can only be detected using heavyweight hardware monitoring tools on each host, which is orthogonal to the scope of SkeletonHunter.

We fixed 98% of the 1,302 problematic network components in Sep. 2024; the remaining 2% cannot be fixed due to their relevance to hardware switches and RNICs whose internal implementations are invisible to CSPs like us [57, 87]. After that, the monthly network failure rate of our production system during Oct.–Dec. 2024 has been reduced by 99.1%, paving a dependable and generic network runtime for stateful/stateless functions in model training tasks.

In summary, this paper makes the following contributions.

- We are the first to point out the real-world challenges against reliable network support for large-scale containerized model training, as well as their multiplicative effect on troubleshooting the connectivity issues.
- We propose SkeletonHunter, a container network monitoring and diagnosis system that leverages the unique traffic patterns of large model training to accurately and efficiently pinpoint the connectivity issues.
- SkeletonHunter has been deployed in our production container network and has helped discover diverse network failures that derive from the problems of different network components. We have fixed most problematic network components and greatly reduced the monthly failure rate.

This work does not raise any ethical issues.

2 Background

The rapid advancement of deep learning models, especially large language models (LLMs) such as GPT 3 [8], Llama 3 [17], Qwen 3 [22], and DeepSeek V3 [14] have driven the demand for massive computing and networking resources for model training. For example, Llama 3 405B is trained with 16K NVIDIA H100 GPUs, lasting for over three months [37]. This unprecedented demand for computational power has created a significant barrier for ordinary users and developers, due to the high cost of acquiring and maintaining such vast computing and networking infrastructure.

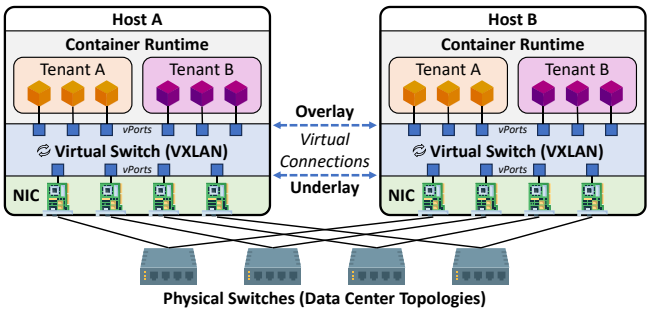


Figure 1: Architectural overview of our in-production multi-tenant large model training infrastructure built on top of the RDMA NIC- and VXLAN-based container network.

To facilitate users' rapid deployment and evolution of models, today's CSPs all provide containerized large model training services [10, 11, 15]. This allows users to submit and run their training tasks on shared infrastructure with a high degree of flexibility and cost-effectiveness. The specific number and configuration of containers, as well as the number of bound RNICs, can all be determined by the users based on their training requirements, although CSPs will also provide a recommendation for their reference. In practice, containerized large model training has become a popular paradigm for common users to deploy training tasks on the cloud. As a major CSP, we have been operating a large-scale containerized model training cloud for over three years. It is equipped with 40K+ RNICs and 40K+ GPUs, and has served 5M+ model training tasks.

Figure 1 shows the architectural overview of our containerized large model training infrastructure. Each host is connected to the underlay network [9] and is equipped with one to eight GPUs for training, along with RNICs for high-performance RDMA communication. Also, the host runs a container runtime for managing multiple containers from different users. When a user creates a training task, the control plane initializes the corresponding containers (i.e., the training nodes) on one or more hosts, and binds the RNICs and GPUs requested by the user to the training nodes.

An RDMA-based container network provides high-performance data communications among containers within the same tenant while isolating the traffic from different tenants [57]. As shown in Figure 1, our RDMA-based container network is built as an overlay network through the VXLAN [1] technique. A software-based virtual switch (i.e., Open vSwitch [66], or OVS for short) is employed to control the packet forwarding across the overlay and underlay. Note that most of the packet en-/de-capsulation and forwarding tasks are offloaded to RNICs, which significantly reduces the CPU overhead and improves the network performance.

3 Motivation

Reliable networking among containers is crucial for containerized large model training. Unlike traditional public cloud services (e.g., web services) that are stateless and can tolerate a certain degree of packet loss, large model training is stateful and highly synchronized, where training containers iteratively exchange the parameters of the model [13]. A single-point connectivity failure can fail the entire training procedure, leading to significant financial loss to end users.

During the three-year operation, we observe that ensuring the network reliability is challenged by the aforementioned *multiplicative effect*: the number of network components to be examined is the multiplication of the numbers of containers, RNICs, and virtual network components involved in a training task. In production, this number can reach up to 128K (e.g., 1K containers × 8 RNICs per container × 16 virtual network components per container, as we have mentioned in §1). This compels us to make efforts to scrutinize the whole network stack to understand and enhance reliability.

3.1 Observations and Challenges

To understand the fundamental challenges in accurately and efficiently capturing connectivity issues of the container network for large model training, we perform a comprehensive analysis on the production data. Our analysis shows that 1) the high dynamics of containers, 2) the endpoint-induced complexity, and 3) the interplay between the overlay and underlay networks are the major obstacles that make existing solutions inapplicable.

High Dynamics of Containers. The high flexibility and multi-tenancy of our containerized training infrastructure lead to extremely frequent training task initializations and thus container creations. Our data show that at peak hours, we need to handle the creation of ~2,000 containers *every minute*. Even in non-peak hours, there are hundreds of containers created every minute.

In addition to the large number of container creations in a short time, we observe a skewed distribution in the lifetime of containers. Figure 2 shows the lifetime of containers in different training tasks with different sizes in production (we measure the *size* of a task by its utilized number of containers). A large portion of the containers have a short lifetime of less than 60 minutes (e.g., ~50% containers for training tasks whose size is ≤256), while the vast majority (70%) of the training containers have a lifetime of less than 100 minutes. The lifecycle of a container network is much shorter than that of traditional bare metal or virtual machine (VM) networks.

Besides, we notice that containers with a higher-end configuration (typically in terms of the number and type of GPUs) tend to have a longer lifetime, as illustrated in Figure 3. This is because containers with lower-end hardware configurations are usually used for debugging or testing during the training process, which are more likely to be short-lived. After the debugging or testing, users will use higher-end containers for actual training tasks.

Moreover, although all containers in the same training task are supposed to have the same lifetime, the state transitions of different containers are highly uncoordinated in practice. This is due to the inherent differences between the host environments of different containers (recall that containers in the same training task are distributed across different physical hosts). Different hosts can have different workloads and caching states, which makes the container orchestration system create/destroy containers with distinct time durations. Figure 4 profiles the different container startup time of different training tasks. Most training tasks require a couple of minutes to initialize all the containers, which presents a phased pattern. Also, larger tasks bear a higher tail delay, where the longest delay can reach up to 10 minutes. The deletion time of containers exhibits a similar situation (and thus is not depicted).

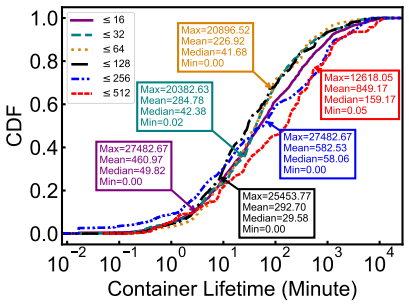


Figure 2: Lifetime distribution of different sizes of training tasks.

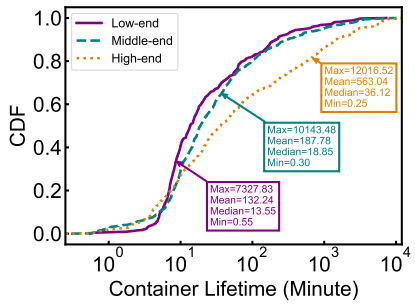


Figure 3: Lifetime distribution of different container configurations.

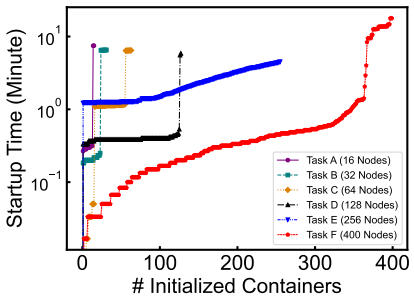


Figure 4: Startup time of the containers in six different training tasks.

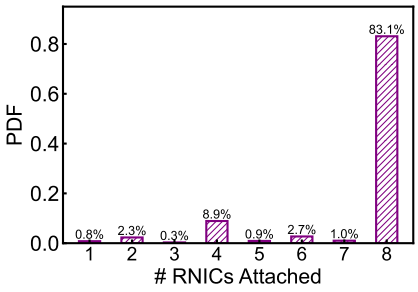


Figure 5: Number distribution of allocated RNICs to each container.

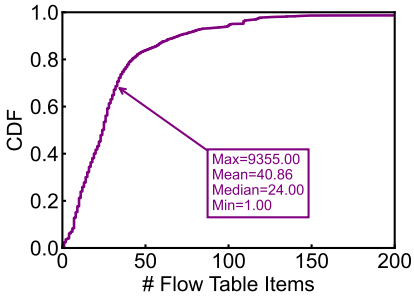


Figure 6: Distribution of the number of flow table items on each host.

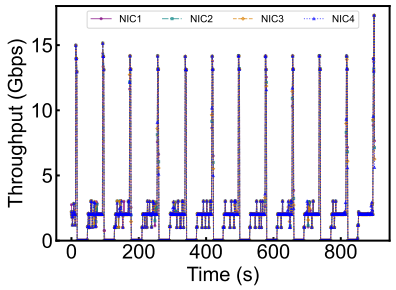


Figure 7: Traffic burst cycles of RNICs in a typical training container.

▹ *Challenge 1: Requiring fast connectivity probing on the highly dynamic network topologies.* The containerized large model training scenario brings short lifetimes to most containers, as well as asynchronous running states to grouped containers (that belong to the same task). Both are different from the situations in traditional long-existing and stable physical networks, rendering existing solutions based on comprehensive or opportunistic monitoring (c.f. §9) inefficient or inaccurate. This makes it even harder for failure detection of problematic short-lived training tasks, where the containers also present short lifetimes like the ones in normal tasks.

Endpoint-Induced Complexity. We note that the multiple RNIC endpoints attached to each container further complicate connectivity monitoring. In our production environment, each container is allowed to be bound to at most eight RNICs in hopes of maximizing the GPU utilization. Figure 5 shows the distribution of the number of RNICs allocated to each container in different training tasks. The vast majority of containers are each bound to eight RNICs, while a nontrivial portion of containers are each bound to four RNICs. This is easy to understand since these numbers are the most common configurations for mainstream model training frameworks to enable efficient data exchange among the GPUs within a training node (e.g., through NVLink [55] and NVSwitch [90]).

In detail, today's data center architectures tailored for LLM workloads typically assign a dedicated RNIC for each GPU [45, 67], so that each GPU can fully utilize the throughput of the RDMA network. As a result, the number of RNICs bound to each container is generally equal to the number of GPUs requested by the container, which is usually the aforementioned four or eight.

▹ *Challenge 2: Requiring efficient coverage of the endpoint-induced complexity.* The multiple RNICs attached to each container additionally introduce a large number of source-destination pairs (corresponding to a wider variety of network paths) that need to be monitored in real time.

Interplay between Overlay and Underlay Networks. To provide the multi-tenancy support, CSPs must isolate the resources and performance for each training task. To this end, an overlay network is constructed to separate the network traffic of different training tasks. However, this introduces a significant number of virtual network components in the network stack compared with the traditional physical network or the VM overlay network. For example, each RNIC should enable SR-IOV [40] and eSwitch [73] to support packet en-/de-capsulation. Also, a virtual switch is deployed on each host to determine how packets should be forwarded in the data plane. This results in a great number of flow tables on each host, making the network stack more error-prone.

Figure 6 illustrates the distribution of the number of flow table items on each host in production, which is in fact only one kind of virtual components in the network stack. As shown, the average number of flow table items on each host is over 40, and the maximum number can reach as large as 9.3K on a host.

▹ *Challenge 3: Requiring effective disentanglement of the overlay-underlay interplay.* The interplay between the overlay and underlay networks incurs a large number of virtual components in the network stack, leading to essentially more difficulty in locating the root causes of connectivity failures compared with the traditional physical network or the VM overlay network.

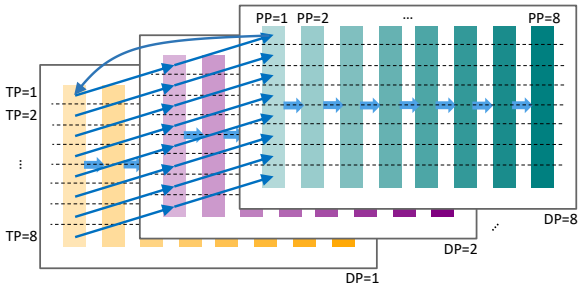
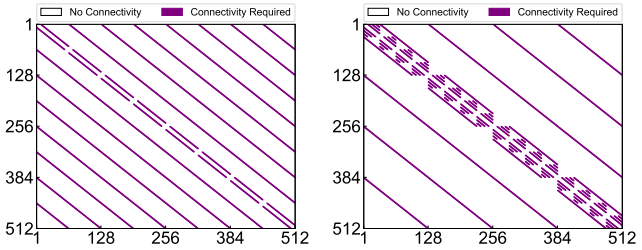


Figure 8: Parallelisms in a dense model training task, where each cell represents a GPU along with its bound RNIC, and each column represents a container.



(a) Dense model. **(b) MoE model.**

Figure 9: RNIC traffic patterns of a 512-GPU task.

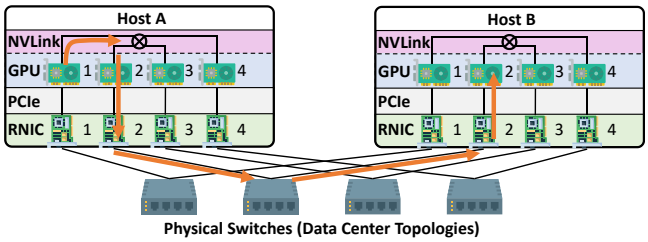


Figure 10: GPU communications in a rail-optimized data center topology.

3.2 Opportunities

Sparse Spatial Distribution. To address the above-described challenges, our first insight is that today's large model training traffic exhibits a sparse yet regular spatial distribution, where the majority of network traffic is exchanged only between the RNICs of containers that have data dependencies. This phenomenon comes from the intrinsic characteristics of mainstream model training frameworks (e.g., TensorFlow [26], Megatron [61], DeepSpeed [68]): various parallelism strategies, such as data parallelisms (DP), tensor parallelisms (TP), and pipeline parallelisms (PP), are used to coordinate multiple GPUs to efficiently train a single large model. Thus, the entire training task is divided into multiple parts according to the employed parallelism strategies assigned to different GPUs. As a result, each GPU only needs to communicate with others in the same parallelism group [45, 67].

Figure 8 shows the three types of parallelisms in a typical training process of a dense model [80] using 512 GPUs. This task is configured with TP=8, PP=8, and DP=8. Each DP takes a different batch of data for training. Within a DP, the involved tensors are split into eight GPUs, and the entire model is divided into eight levels of pipelines for forward and backward propagation. The intermediate training results are exchanged among GPUs in the same stage or neighboring stages of the pipelines. After each training iteration is finished, the parameters across different DPs are exchanged among the counterpart GPUs in the same stage of the pipeline. In particular, within each host (and its hosted containers), the communication among GPUs is conducted by high-bandwidth intra-host links like NVLink, while the inter-host communications are over RNICs. Thus, network transmission only happens in certain RNICs.

Figure 9a shows the corresponding traffic matrix among the RNICs of the example training task, which is highly sparse. This offers us the opportunity to efficiently monitor the network connectivity by focusing on source-destination pairs that actually have the connectivity (rather than all pairs). Besides dense models, other emerging models such as Multiple-of-Expert (MoE) models [19] introduce new parallelism strategies like expert parallelism (EP). As shown in Figure 9b, new parallelism strategies may have different traffic patterns, but the sparse spatial distribution still holds.

This sparse traffic distribution is caused by the combination of two factors. First, today's model training in data centers widely uses rail-optimized topology [5, 67], which can fully utilize high-bandwidth NVLink for maximizing the cluster scale. As shown in Figure 10, in a rail-optimized topology, different RNICs on the same host belong to different rails and connect to different top-of-rack (ToR) switches, making in-rail transmission the best choice for any inter-host communications. Second, collective communication libraries like UCC [21], NCCL [20], MSCCL [18], and MPI [44], will automatically transform any cross-rail transmissions to a combination of intra-host NVLink transmission and inter-host in-rail transmission to maximize the performance. Thus, traffic patterns are shaped to be sparse as in Figure 9.

Temporal Burst Cycles. Apart from the traffic sparsity in the spatial dimension, the training traffic also presents a strong periodic and seasonal pattern in the temporal dimension. Figure 7 shows the throughput of RNICs in a typical training container. As shown, during the 900-second period, there are multiple periodic traffic peaks, where the throughput reaches up to 15 Gbps[1]. Between two adjacent peaks, the throughput of the RNICs is low (even idle). Such burst cycles are brought by the collective communications of different phases in model training, where each training task is divided into multiple iterations. In each iteration, network transmission is trivial across model layers (i.e., the idle period). However, after the gradients are calculated at the end of an iteration, the network traffic thus becomes bursty as all the DPs need to exchange a large number of parameters to synchronize the model (usually with an all-reduce operation [43]).

The burst cycles of the model training traffic provide the opportunity to distinguish the "role" of each container. Specifically, by comparing the time series of the traffic throughput of the RNICs of the training containers, we can determine in which parallelism group the current training container is working. Therefore, we can

[1]The actual peak value is much higher (close to the line rate). However, due to limited monitoring granularity in production (1 second), we can only present the average value in each second.

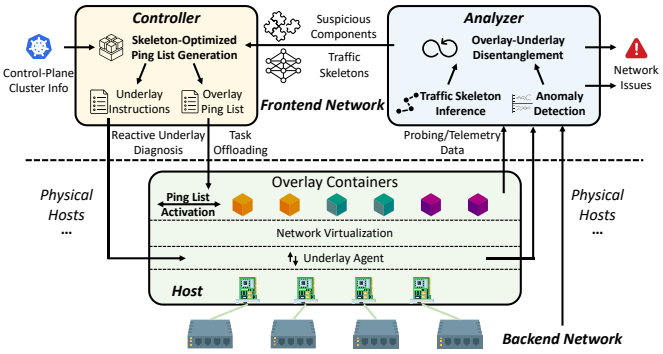


Figure 11: Architectural overview of SkeletonHunter.

precisely calculate the minimal source-destination pairs that are necessary for connectivity monitoring, and remove the unnecessary ones to significantly reduce the monitoring cost.

4 System Overview

We present SkeletonHunter, a system that enables efficient and accurate detection of network issues for our in-production containerized large model training services. Our key idea is to use the unique sparse traffic patterns of large model training to reduce the overhead of network monitoring and improve the accuracy of problematic network component localization. As a CSP, however, we have very limited visibility into the tenants' model composition for privacy reasons and thus we cannot directly exploit the sparsity.

To address this, we propose a more pragmatic approach by inferring the *traffic skeleton* that essentially connects all the active endpoints in a training task. As shown in Figure 11, SkeletonHunter follows the traditional Pingmesh [47] architecture and consists of three major components including the controller, the agent, and the analyzer to realize the methodology. The controller is responsible for generating probing tasks (a.k.a., ping lists) and instructing the agents for each container to perform actual probing. The agents report the probing results (including end-to-end latency and packet loss rate) to the analyzer for failure detection and localization. It employs the following techniques to meet our goals.

- *Traffic Skeleton Inference* (§5.1). To enable fast ping task execution, SkeletonHunter infers the sparsity of the training traffic (i.e., the traffic skeleton) by exploiting the traffic bursts of the RNICs. In particular, SkeletonHunter distinguishes parallelism configurations such as DPs, TPs, and PPs for constructing the concrete traffic skeleton and the corresponding RDMA probing matrix. Besides, to accommodate the high dynamics of container state transitions, the agents execute probings in a phased manner.
- *Connectivity Anomaly Detection* (§5.2). After collecting probing results from agents, the analyzer applies temporal windows (i.e., a 30-second short-term window and a 30-minute long-term window) over these data to differentiate the latency patterns among endpoints. It then identifies abnormal patterns that are prone to have connectivity issues through statistical distribution testing. In this way, the analyzer discovers anomalous end-to-end latency increase and packet loss, as well as the corresponding suspicious network components that lead to the anomalies.
- *Optimistic Overlay-Underlay Disentanglement* (§5.3). Once detecting a connectivity issue, SkeletonHunter localizes it by eliminating the interplay between the overlay and underlay networks. It examines the root causes at the overlay and underlay layers separately with an optimistic (and reasonable) assumption that the root causes at the two layers are mostly software- and hardware-related respectively, so they will not propagate to each other. If the root cause is not found at either layer, SkeletonHunter will validate the RNICs that connect the two layers.

5 Component Design

5.1 Traffic Skeleton Inference

The frequent container creation and state transitions require extremely efficient ping list generation and connectivity probing. To achieve this, SkeletonHunter generates the ping list for the agents in a phased manner, which adapts the ping task in $O(1)$ time complexity on scaling. First, considering pervasive sparse traffic patterns, SkeletonHunter constructs the basic pruned ping list with 87.5% scale reduction. Second, to conquer false probing incurred by different startup times of containers on training task initialization, the above ping list is incrementally activated in the data plane. Finally, at the runtime of the training task, the ping list is further optimized based on deduced traffic skeletons.

Preload: Basic Ping List Generation. For a real-world training task, as shown in Figure 10, commutations only occur in the same rail. When the training data in GPU 1 of Host A are transmitted to GPU 2 of Host B, the data are first transmitted to the GPU 2 of the Host A through the intra-host NVLink, and then forwarded to the GPU 2 of the Host B through network links. This leaves the opportunity for the basic pruned ping list generation. Specifically, cross-rail communications would be automatically optimized to the combination of intra-host NVLink transmission and same-rail inter-host transmission [4]. Therefore, the network communication is only conducted within the same rail of the network topology (i.e., the same rank of the RNICs among different hosts).

Based on the above property, SkeletonHunter first removes source-destination pairs from a full-mesh ping list (i.e., all RNICs should ping the other RNICs in the same training task) that are not in the same rail. For typical training-oriented hosts in production, the number of equipped GPUs and RNICs on each host is eight [45, 59, 67]. Therefore, we can reduce the scale of the basic ping list by 8×. This process is done immediately at the controller when the user submits the training task (even before the container initialization), so we call it as the preload phase.

Initialization: Incremental Ping List Activation. If the containers directly execute the basic ping list after it is created, we may encounter a large number of false positives on connectivity issues derived from container state transitions. This is because the containers that are under initialization may not have finished their network stack initialization, and thus are reachable from the already created containers, causing packet loss or high latency.

To avoid false positives, we choose to incrementally activate the basic ping list on agents based on the container status (only running containers are selected as destinations). However, if these updates are all done by the controller as in traditional Pingmesh [47] solutions, the controller will become the bottleneck (recall that the

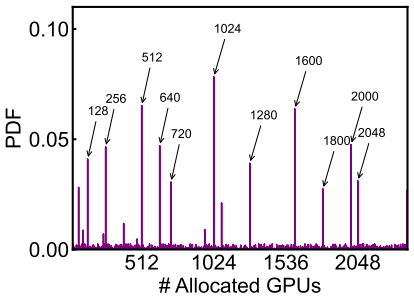


Figure 12: Distribution of the number of GPUs in a training job.

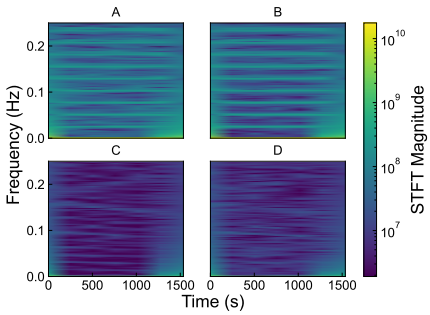


Figure 13: Frequency-domain features of two kinds of burst cycles.

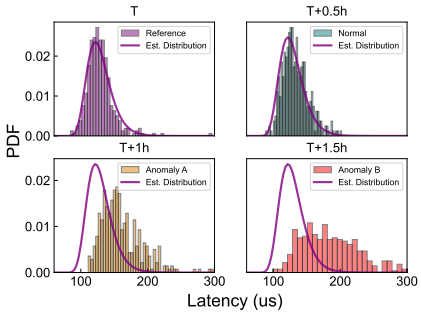


Figure 14: Long-term latency distribution tracking.

state transitions of containers are highly frequent). Instead, our key insight is that the high dynamics of the data plane should be handled by the data plane itself. In SkeletonHunter, the controller offloads the ping list activation to data-plane containers.

Concretely, when a container is created, its agent will first get the basic ping list from the controller but does not start the actual connectivity probing until other containers *register* themselves to *activate* the corresponding ping target recorded in the already created source container. This registration process is done once a container is created and ready to be pinged. In this way, we avoid false alarms on container initialization.

Runtime: Optimization with Inferred Traffic Skeletons. Even with the optimized basic ping list, there are still many inactive source-destination pairs in it. As exemplified in Figure 9a, the basic ping list for a single GPU in a 512-GPU task should have 64 destinations. Actually, only nine destinations are connected in practice, leaving 85.9% space for further optimization. However, achieving this optimization requires a deep understanding of users' model parallelism strategies, which is impossible for a CSP like us. Existing solutions [56] that capture detailed five-tuple information of the network connections of the service links are also not applicable to the containerized training scenario, as we cannot modify relevant RDMA APIs of the network stacks inside the users' containers for privacy, security, and stability reasons.

To address this, we propose a more pragmatic yet effective approach: *inferring* the traffic skeletons (i.e., a crucial set of network paths that the training traffic consistently traverses) through the easy-to-obtained throughput burst cycles (cf. §3.2) over containers' RNICs, together with the common parallelism patterns used in production. In detail, Figure 12 illustrates the scale of training tasks in production. First, the number of requested GPUs (and the corresponding RNICs) in a training task is only confined to a limited set of values (e.g., 128, 512, and 1,024), which are often multiples of eight. This is easy to understand since if the users want to fully utilize GPUs' computation capacity, they must request a fixed number of GPUs so that the GPUs can be divided into different groups for efficient parallelism (e.g., $\#GPUs = TP \times PP \times DP$).

Next, we notice that the temporal throughput burst cycles are similar for RNICs (and GPUs) in the same position across different parallelism groups. For example, in Figure 8, the upper left allocated RNICs in the same position across the eight different DP groups have the same burst cycles. This is because the DP only divides the input training data into different chunks, while the training processes for these chunks are the same. With the above two key observations, we devise traffic skeleton inference as follows.

We extract the frequency-domain features of the throughput burst cycles of containers' RNICs to describe traffic periodicity. We use Short-Time Fourier Transform (STFT) [28] to convert the time-domain throughput burst cycles into the frequency domain. We have tried other feature extraction methods like Wavelet Transform [42] and Discrete Fourier Transform [82], while STFT can capture the time-varying characteristics with the lowest computational complexity, which is crucial for runtime analysis. As shown in Figure 13, after the conversion, the throughput burst cycles of the RNICs A, B, C, and D present two kinds of frequency components, where A and B share similar STFT features, and C and D share similar ones. This similarity indicates that the RNICs A and B (C and D) are in the same position across different DPs.

By applying state-of-the-art clustering algorithms on the extracted STFT features, we can cluster RNICs into different groups, where RNICs in the same group are highly likely to be in the same position across different DPs. In particular, we use the hierarchical clustering algorithm [60] for grouping by measuring the similarity of the traffic burst's STFT features on the RNICs. We further apply the following constraints to the grouping process so that the grouping results are more interpretable based on the number GPUs allocated to the training task:

$$\textbf{\textit{min}} \quad \sigma^2 = \frac{1}{k}\sum_{i=1}^{k}\left(\|c_i\| - \bar{c}\right)^2, \tag{1}$$

$$\textbf{\textit{s.t.}} \quad N \bmod \lfloor \bar{c} \rceil = 0, \tag{2}$$

$$r_1, r_2, \cdots r_x \in H_r \Rightarrow \forall c_i, \|c_i \cap H_r\| \leq 1, \tag{3}$$

where k is the total number of RNIC groups in a training task, c_i represents the i-th group, $\lfloor \bar{c} \rceil$ is the nearest integer of the average number of the RNICs in each group, N is the total number of the RNICs, r_i is the i-th RNIC in host H_r.

The objective function in Equation (1) minimizes the variance of the number of the RNICs in each group. This is particularly important since each individual model training pipeline in a training task should have the same scale, which is $TP \times PP$. Similarly, we add the constraint in Equation (2) to ensure that the number of the RNICs in each group can be evenly divided by the number of DPs. Finally, the constraint in Equation (3) ensures that the RNICs in the same host are not in the same group, since in this case, the RNICs are in the same DP for NVLink acceleration within the host.

The above process helps us infer the DP groups of the training task, whose value is equivalent to $\lfloor \bar{c} \rceil$. We next infer the TPs and PPs of the training jobs based on $TP \times PP = N/\lfloor \bar{c} \rceil$. We further leverage the time shift of the throughput burst cycles to distinguish the level of PPs. For example, the PP in the first layer always experiences the same traffic burst earlier than the PP in the second layer.

As a result, we finally infer the parallelism patterns of training tasks, and we can determine the traffic skeleton (i.e., which pairs of endpoints have the actual connectivity) for each task. The latest new models may introduce extra parallelism strategies (e.g., EP), but can be classified using the same method. At the runtime of the training, SkeletonHunter removes unnecessary targets from the ping list for further optimization. By only enabling the probing along the traffic skeleton in the probing matrix for each container, the ping list can be further reduced by >95% (cf. §7.1).

5.2 Connectivity Anomaly Detection

While a high packet loss rate can be easily classified as a network issue, a sudden high latency can be caused by transient congestion or network resource contention. It is necessary to perform data analysis to filter out these transient latency spikes. To this end, our key idea is to leverage state-of-the-art sequential analysis techniques [79, 86] to coherently evaluate whether the communication patterns have changed over time. Concretely, the analyzer of SkeletonHunter aggregates the collected data, and performs short-term and long-term latency anomaly detections through a statistical testing. The rationale behind this is the law of large numbers [49, 86], which indicates that the average measured data will be closer to the true value if we have sufficient data samples.

Short-term Latency Anomaly Detection. We aggregate the latency data at a fine granularity (30s) for each pair of RNIC endpoints for short-term analysis. In each temporal window, we describe the latency distribution using the 25th percentile, 50th percentile, 75th percentile, minimum, mean, standard deviation, and maximum values. Then, we perform latency anomaly detection based on the local outlier factor (LOF) [33] on the latency distributions on each time window. LOF is a density-based score to measure the local deviation of a data point with respect to its neighbors. We set a look-back window of five minutes to calculate LOF for each time window so as to cover probable burst cycles. If a new five-minute window has a high LOF that cannot be clustered into previous windows, we determine that an anomaly occurs.

Long-term Latency Anomaly Detection. For the long-term analysis, we aggregate and analyze the latency every 30 minutes. This design is to prevent gradual degradation of network performance, which is not easy to detect in the short-term analysis since the gradual anomaly may be cumulatively clustered into the previous short-term windows. As we can gather a huge number of latency data in the long-term analysis, we apply statistical tests to detect the latency anomalies. We find that the latency data in the long term of two RNICs that work properly always follows a log-normal distribution [2], i.e., the logarithm Y of the latency data X follows a normal distribution $Y = \ln(X) \sim N(\mu, \sigma^2)$.

As shown in Figure 14, we perform parameter estimation [31] on the latency data for each source-destination RNIC pair at time T and derive the estimated log-normal distribution. At time $T + 0.5h$, $T + 1h$, $T + 1.5h$, we apply Z-test [62] on the collected latency data respectively to test whether the data still follow the estimated log-normal distribution. In the example, latency data at $T + 0.5h$ still follow the estimated distribution, while those at $T + 1h$ and $T + 1.5h$ deviate from the estimated distribution. For $T + 1h$ and $T + 1.5h$, we determine that a latency anomaly occurs.

Algorithm 1 Failure Disentanglement and Localization

```
1:  global PhyLinkCounter: map[int]
2:  L_O, L_U ← Separate(P_AB)                          ▷ Overlay/underlay links
3:  procedure Separate(P)
4:      for p ← P do
5:          if packets in p encapsulated then P_O ← P_O ∪ p
6:          else P_U ← P_U ∪ p
7:  procedure OverlayReachability(L_O)
8:      Current ← L_O[0]
9:      Visited ← set(Current)
10:     for l ← L_O[1 :] do
11:         Current ← Forward(Current, l)
12:         if Current = null or Current ∈ Visited then
13:             return l                               ▷ Overlay failure point
14:         Visited ← Visited ∪ Current
15:     return null                                    ▷ No overlay failure
16: procedure PhysicalIntersection(L_U)
17:     for l ← L_U do
18:         PhyLinkCounter[l]++
19:     if ∀l ∈ L_U, PhyLinkCounter[l] ≤ 1 then
20:         return null                                ▷ No underlay failure
21:     return MaxCount(L_U)                           ▷ Underlay failure point(s)
```

5.3 Optimistic Overlay-Underlay Disentanglement

After detecting high packet loss or latency distribution anomalies, SkeletonHunter can only determine there is a network issue between two containers, but cannot pinpoint which network component causes this issue so far. To address this, we devise an optimistic overlay-underlay disentanglement mechanism (as shown in Algorithm 1) to locate the network issues under the assumption that the root causes of the overlay and the underlay layers are software- and hardware-related respectively, which will not propagate to the other layer. It first separates the paths between the two containers into distinct overlay and underlay links (lines 1-6). Then, it performs the overlay logical reachability analysis (lines 7-15) and the underlay physical intersection analysis (lines 16-21) to locate the network issues in the two layers, respectively.

Overlay Network Failures. As shown in Algorithm 1, the analyzer of SkeletonHunter examines the logical forwarding chain across the problematic endpoints in the overlay network. It relays the packet forwarding process and checks if the packets are correctly forwarded to the destination or whether there is a loop. Once it detects unreachability (i.e., *Current* is null in Algorithm 1), it can pinpoint the problematic overlay link at the broken point. If the packet is forwarded to a visited component, it determines that overlay forwarding rules are incorrect and a loop exists.

Physical Network Failures. Due to path multiplexing induced by ECMP routing in the underlay network [45], SkeletonHunter leverages the network tomography [65] technique to vote for the physical links that are most likely to be faulty. Besides, we further deploy an agent on each physical host to enable traceroute probing for underlay path intersection, which is similar to the scheme used in R-Pingmesh [56] and 007 [29].

Validating RNICs. If the above two processes cannot locate any problematic network components while the issue does exist, SkeletonHunter further validates the RNICs that connect the overlay and underlay, which involves some manual operations. The host agent dumps the flow tables offloaded from the OVS to RNICs for preliminary detection of the inconsistency between the two layers. This process can cause temporary network performance degradation, but it is necessary to ensure the correctness of the network configurations. If no inconsistency is detected, we manually check the configurations of the RNICs and OVS to locate network issues. With all of the above designs, SkeletonHunter can effectively locate network issues at both overlay and underlay, and classify them into physical switches, RNICs, virtual switches, host configurations, etc.

Rationale behind the Optimism. In practice, the optimistic disentanglement may not persistently work well in all cases, and we have encountered the cases where the issues of the overlay and underlay layers occur at the same time. For example, the underlay RNIC's unexpected behaviors can cause the misconfiguration of the overlay virtual switch [57], which further exacerbated network issues. In this case, we can only manually resolve the problem based on our domain knowledge and experience.

Nevertheless, it should be noted such a situation is rare in our production environment, and designing a system that can handle all possible issues happening in containerized model training is not practical. The optimistic disentanglement can handle most of the cases we have seen in production since it effectively verifies the overlay's logical correctness and the underlay's physical link states.

6 Implementation

Controller. The controller is implemented with 6,350 lines of code in Java. It is responsible for inferring traffic skeletons (with the help of the analyzer) and managing probing tasks for agents, which is deployed on two servers in our frontend network for load balancing and fault tolerance. All the communications between the controller and agents are encrypted so that users cannot "forge" the requests to obtain the information of the training tasks of other users. The controller connects to the database to synchronize the states of the training containers. It receives the analysis results from the analyzer so that it can update the instructions to the agents (e.g., dump RNICs' flow tables).

Agent. We implement the overlay and underlay agents with 5,400 lines of code in golang [36]. For the overlay agent, it is launched through sidecar containers [72] along with training containers. In this way, an agent is isolated from the training container at the process level, while sharing the network namespace with the training container that it monitors. When a training container is launched, the agent automatically queries the controller to get the ping list and then incrementally activates the probings after the registrations of other containers in the same training task. In terms of the underlay agent, it is deployed as a standalone container in the host, which is accessible to all the host's resources. Both agents are highly configurable by the controller.

Analyzer. We use the available services in our cloud environment to construct the analyzer, including log services [24] and real-time computing services [23]. The log service stores both stateless and stateful measurement data from agents, which are then indexed and aggregated based on the training tasks, containers, RNICs, as well as uplink switches. The real-time computing service analyzes the logs and generates analyzing results to inform the controller and trigger alarms in a feedback-loop manner.

7 Evaluation

We have deployed SkeletonHunter in production for over 10 months. We utilize online statistics from one large-scale production cluster with 4K+ physical hosts to evaluate SkeletonHunter. Each host is equipped with eight commodity RNICs (either 200 Gbps or 400 Gbps throughput), 128-core CPUs, and 2 TB memory. Each RNIC runs in SR-IOV mode with 128 virtual functions (VFs)[2]. Users can specify the number of GPUs and training nodes (i.e., containers) on demand. The information of all training tasks is synchronized with the SkeletonHunter controller, so that SkeletonHunter can monitor the connectivity for each task. The evaluation involves six-month (Mar.–Aug. 2024) data with 2M+ tasks.

7.1 Efficiency and Effectiveness

The Scale of Probing Targets. SkeletonHunter significantly reduces the scale of the probing matrix with the help of the traffic skeleton. Figure 15 shows the scale of probing tasks on different numbers of allocated RNICs. At all the RNIC configurations, the number of probing targets (SkeletonHunter-Basic denotes the basic ping list) is always an order of magnitude smaller than that in the full-mesh solution. For example, with 2,048 RNICs, the full-mesh solution requires an average of 60,430.32 probings for each iteration round among all the training containers, while SkeletonHunter only involves 2,593.03 probings. Note that although some of the existing network monitoring solutions like deTector [64] also reduce the scale of probing, they still require a large number of probes in practice (e.g., 15K+ probes for each training iteration round in deTector). This is because these solutions only simplify the probing matrix by considering common data center topologies without the awareness of the training workloads that have a high sparsity and symmetry in their traffic patterns.

Time Cost of Each Probing Round. Given the significantly smaller probing matrix, we evaluate the actual time cost of a probing round in SkeletonHunter. As shown in Figure 16, the full-mesh solution bears a significantly longer probing time on all RNICs. When the number of RNICs in a training cluster is 512, 1,024, and 2,048, the full-mesh solution consumes 560.25, 1,123.43, and 2,034.12 seconds, respectively. In contrast, under the same probing frequency and cluster configuration, a basic ping list in SkeletonHunter only consumes 64.85, 122.54, and 240.54 seconds respectively, while the final ping list with the inferred traffic skeleton consumes 8.23, 16.91, and 25.09 seconds for probing. These results show that the two

[2]We enable 128 VFs per RNIC for faster VF allocation and deallocation across different containers, as well as supporting RNIC sharing in the future.

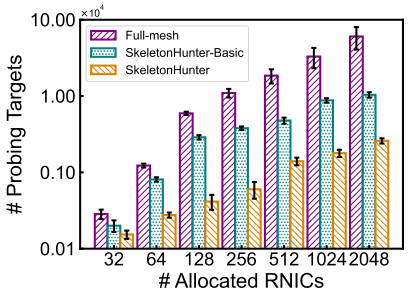


Figure 15: Comparing probing task sizes of different schemes.

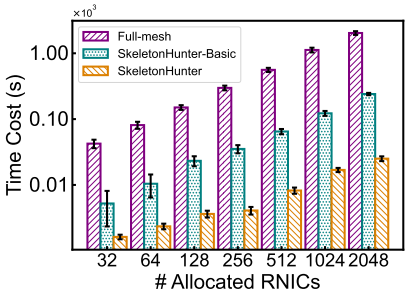


Figure 16: Time cost of probing all endpoints.

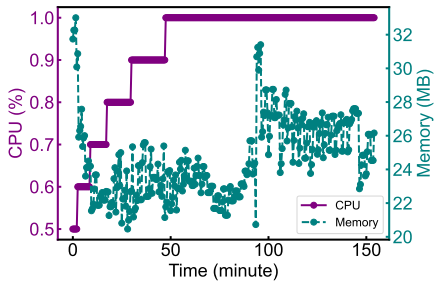


Figure 17: Resource consumptions of SkeletonHunter in production.

phases of probing in SkeletonHunter all significantly reduce the time cost under the same probing frequency. The final ping list can further reduce the probing time by 87.3%, 86.2%, and 89.6% respectively compared with the basic ping list.

Agent Overhead. The containerized environment imposes a constraint that the agent should have trivial resource consumption, or the overall overhead will be unacceptable due to the large number of training containers. In practice, the agent of SkeletonHunter incurs negligible overhead on the training containers. As shown in Figure 17, the CPU and memory consumptions of the agent always converge to 1% and 35 MB respectively over the entire lifetime of a container. This is easy to understand since we utilize the unique traffic patterns of the training jobs to minimize the probing matrix, so that redundant probing tasks are largely eliminated.

Detection Accuracy. SkeletonHunter achieves a high accuracy in detecting network reliability issues in production. By manually checking all the detection results, we note that SkeletonHunter achieves a high precision of 98.2% and a high recall[3] of 99.3% in discovering network failures during the operation of our services. In total, it helps us detect 4,816 network failures and correctly localize them to 1,302 problematic components with an accuracy of 95.7%, which significantly accelerates the diagnosis and recovery of network reliability issues for containerized large model training in production. We have fixed 98% of the issues, while the remaining cannot be fixed due to the limited visibility into the hardware of commodity switches and RNICs [57, 87]. After the fixes since Oct. 2024, the monthly network failure rate has reduced by 99.1% compared with the previous period.

7.2 Network Failure Localization

We summarize representative network issues detected by SkeletonHunter in Table 1. All these issues can be categorized to 19 different types, which are mainly related to six components in our model training services (i.e., physical switches, RNICs, host boards, virtual switches, container runtime, and configurations).

Link/Switch Anomalies. For network issues occurring in the inter-host network (Issue 1–4), SkeletonHunter can filter all abnormal probing results and leverage the network tomography to locate the anomalous devices (either links or switches). Most link/switch anomalies can be immediately verified by warning logs on the corresponding switches to quickly determine the root causes. Therefore, we deploy an automatic diagnosis and repairing pipeline based on

[3]We obtain false negatives based on users' feedback.

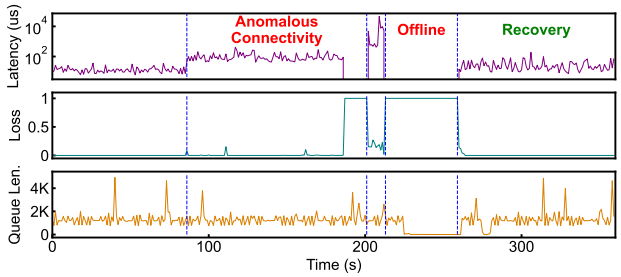


Figure 18: A case study for SkeletonHunter, where the anomalous connectivity behaviors are caused by the flow table inconsistency between the overlay and underlay components.

it to trigger subsequent repairing procedures according to different root causes (e.g., fiber module cleaning for CRC error/port flapping, and fiber module replacement for port down).

Host-related Anomalies. According to our experience, a variety of factors (Issue 5–13) may lead to host-side anomalies. When these issues occur, we immediately isolate problematic hosts/modules to eliminate their effects on model training.

Figure 18 shows a typical case encountered in production. Before the 90th second, the latency between two RNICs of containers is stable at around 16µs. However, after the 90th second, the latency increases to 120µs, and the ping packets bear a small packet loss (<0.1%). With statistical testing, SkeletonHunter determines that such a latency is problematic since it deviates significantly from the history latency distribution. This can be validated through the switch queue length, which hardly increases during the period of anomalous latency spikes and indicates that the actual throughput does not reach the bottleneck. SkeletonHunter in fact did not find any overlay/underlay issues at first, and thus dumps the RNIC flow tables. It then detects an inconsistency in the flow tables for the overlay virtualization, and isolates the RNIC immediately. After that, the RNIC recovers in 60 seconds, and all metrics return to normal. Our further investigation reveals that this issue is because the RNIC did not update flow counters timely, which makes the control plane regard the flow as inactive and invalidates it from the RNIC. As a result, relevant packets are processed at the software stack with a significantly higher latency. With SkeletonHunter, we immediately get alerts and take actions to mitigate this issue.

Virtual Switch/Container Anomalies. In addition to physical links, switches, and hosts, software components (such as virtual switches, containers, and their related configurations) can also become the culprit of reliability issues (Issue 14–19). Nevertheless,

Table 1: Network issues detected by SkeletonHunter in production.

No.	Issues	Components	Symptoms	Detailed Reasons
1	CRC error	Inter-host Network	Packet Loss	Physical fabric causes packet corruption.
2	Switch port down	Inter-host Network	Unconnectivity	The switch port is unreachable.
3	Switch port flapping	Inter-host Network	Packet Loss	The switch port is flapping.
4	Switch offline	Inter-host Network	Unconnectivity	The switch crashes or is manually set to offline for upgrade.
5	RNIC hardware failure	RNIC	Unconnectivity	Hardware components of the RNIC are not working normally.
6	RNIC firmware not responding	RNIC	High Latency	RNIC firmware bugs result in high latency of specific flows.
7	RNIC port down	RNIC	Unconnectivity	The RNIC port is consistently down.
8	RNIC port flapping	RNIC	Packet Loss	The RNIC port is periodically down.
9	Offloading failure	RNIC	High Latency	Packet en-/de-capsulation cannot be offloaded to the RNIC.
10	Bond error	Kernel/RNIC	Unconnectivity	Unable to bond the ports of the RNIC.
11	RNIC GID change	Kernel	Unconnectivity	The network service of the OS is restarted unexpectedly.
12	PCIe-NIC error	Host Board	High Latency	The RNICs in the same host cannot communicate with each other.
13	GPU direct RDMA error	Host Board	High Latency	The GPU cannot directly communicate with the RNIC in the container.
14	Not using RDMA	Virtual Switch	High Latency	Flows that should be transmitted over RDMA are actually using TCP/UDP.
15	Repetitive flow offloading	Virtual Switch	High Latency	Offloaded flows are frequently invalidated in the RNIC.
16	Suboptimal flow offloading	Virtual Switch	High Latency	Flows are offloaded with incorrect orders with high latency of some flows.
17	Container crash	Container Runtime	Unconnectivity	Containers crash shortly after creation due to container runtime defects.
18	Hugepage misconfiguration	Configuration	High Latency	The host's hugepage configuration is not consistent with the RNIC.
19	Congestion control issue	Configuration	High Latency	The congestion control of a specific queue in the switch is not enabled.

these issues can be resolved quickly by restarting or reinitializing the corresponding software components. In this way, SkeletonHunter can truncate usually *hours* of complete testing and directly perform the recovery procedure in *minutes*, which significantly reduces our operational costs.

7.3 Limitations

Users' Uncertain Workloads. SkeletonHunter is built based on a key observation that today's large model training workloads exhibit sparse traffic patterns due to the wide usage of the collective communication primitives. Although this holds true for most of users' workloads, there can still be some users who do not follow this pattern. If the user simply starts up a container cluster for debugging the model or the collective communication libraries, the inferred traffic skeleton might be inaccurate. Moreover, the parallelism strategies of model training is constantly evolving, which brings new traffic patterns that are unknown to SkeletonHunter. In these cases, SkeletonHunter may fail to cluster the containers' RNICs into the same group, or generate a larger probing matrix than necessary, leading to a higher probing overhead.

To address these issues, we can further make efforts from two aspects. First, SkeletonHunter can evaluate the fidelity of the inferred traffic skeleton before the actual probing. For example, it can validate whether the traffic skeleton persistently aligns with the actual traffic bursts. Second, on the user interface of our container services, we can provide an option for users to manually disable the probing if the users are aware that their workloads use the parallelism strategies beyond standard collective communication primitives. We can also provide some degrees of flexibility for users to add hooks to their intra-host software components (e.g., RDMA and collective communication libraries), so as to further validate the traffic skeleton. Besides, our engineering team is also working on a more generic traffic skeleton inference algorithm that can adapt to the evolving parallelism strategies.

False Detections. We did encounter some false detections with SkeletonHunter in production. Specifically, one of the main reasons is that SkeletonHunter mostly focuses on detecting end-to-end connectivity issues, while the connectivity of the intra-host components for model training is only partially covered. As we have mentioned in Table 1, SkeletonHunter can detect PCIe-to-NIC errors, which manifest as the unconnectivity of a container to any other containers in the training task. However, it cannot detect the GPU-to-GPU and GPU-to-PCIe connectivity issues. This is because such connectivity is not network-related but hardware-related. These issues can only be covered by other hardware monitoring tools, which are orthogonal to SkeletonHunter.

Some of the false detections are caused by the defects of SkeletonHunter itself. In order to accurately measure end-to-end latency, SkeletonHunter leverages the precision time protocol [3] to eliminate clock drifts. This requires the agent to respond to the probing requests in a timely manner, while we have encountered some cases where the agent crashes and cannot respond to the probing. As a result, SkeletonHunter regards the corresponding links as problematic by mistake and triggers the alarms. Such issues reveal the importance of improving the reliability of the monitoring system before using it to monitor the reliability of the production system.

It should be noted that SkeletonHunter is only one of the crucial components for failure detections and recovery in our containerized modeling training service. The above false detection caused by SkeletonHunter can still be resolved by our other monitoring systems like error log analyzers.

8 Experience

Using Pings for Connectivity Monitoring. Choosing to use pings for probing is actually a trade-off between the monitoring overhead and the monitoring accuracy. Of course, one can choose to examine all packets and all real-time connections to get the most accurate traffic skeleton and the corresponding connectivity information. However, this would incur a very high overhead and may also induce privacy issues in the multi-tenant environment.

Deployment Experience of Existing Methods. Before we developed and deployed SkeletonHunter, we had tried several existing solutions, in particular Pingmesh and its variants [47, 56]. We found

that these solutions are not fully suitable for our production environment because they are not designed for multi-tenant containerized model training whose network topology is constantly changing. Also, some of the existing methods require special hardware features like IP-in-IP techniques of the switches [78], which induce additional operational costs and stability risks.

Handling Detected Failures. Although SkeletonHunter mainly focuses on failure detection, it also provides some basic failure handling capabilities by integrating with our existing network management systems. When SkeletonHunter detects an anomaly, it will first trigger an alert to notify our network operation team. Also, it will automatically add the corresponding hosts or RNICs to the blacklist, so that no new training tasks will be scheduled on them until the issue is fully resolved. Besides, we are now developing a more comprehensive live migration mechanism for the quick recovery of training containers, which aims to minimize the impact of network failures on training tasks.

Accelerating Agent Evolution. Due to the rapid development of large-scale model training scenarios, the infrastructure (e.g., GPUs, RNICs, and data center topologies) and the trained models are constantly evolving, which requires the continuous upgrade of SkeletonHunter. In the ten months of operation, we have conducted over 20 online updates, and each is fully deployed across all our physical clusters. To achieve such rapid updates, we employ a series of optimizations in our implementation.

For the deployment of the SkeletonHunter agent, the utilization of sidecar containers (cf. §6) decouples the deployment and updates of the agent from the updates of the training tasks. With sidecar containers, we designed monthly routine releases (to support significant upgrades) and weekly emergency releases (to support hot fixes). After a new agent release, new training tasks automatically run with the latest version of the sidecar container. As old training tasks are gradually finished, the agent will then complete the full updates across all physical clusters and thier hosted containers.

Although using sidecar containers may introduce some overhead, it is still cost-efficient because we have deployed other services over sidecar containers besides SkeletonHunter's agents, such as VF statistic monitoring and container health checks. Therefore, using sidecar containers is in fact a unified solution for monitoring and managing our services with minimal operational costs. In addition, we have also designed a suite of strategies to enable consistently high-performance execution of serverless applications for SkeletonHunter agents and sidecar containers, such as data-driven scheduling, resource optimizations based on workload distributions, as well as adaptive resource allocation for load balancing.

Coordinating All Systems for Containerized Model Training. The containerized model training scenario differs significantly from traditional bare-metal deployments, especially when providing multi-tenant services that require strict security isolation. For example, the use of SR-IOV techniques for network virtualization to achieve secure containers with kernel isolation [12, 25] introduces an essential drawback—after enabling VFs on RNICs, RDMA capability cannot be enabled on the physical function (PF). This makes it impossible to monitor and diagnose the RDMA network directly on the host, and presents a significant challenge during the development of our multi-tenant containerized model training services. A number of RDMA-related components (such as monitoring, probing, configuration, and management) require modifications in both deployments and implementations. While SkeletonHunter only shares the same network environment with the training containers through the sidecar mechanism, the refactoring of other functionalities is much more complicated.

Although this paper focuses on particular issues related to network monitoring and diagnosis, containerized training services need a collaborative effort of nearly all network components to address emerging challenges and ensure optimal user experience. This creates considerable potential for further research and innovations.

9 Related Work

Data Center Network Diagnosis. Over the past decade, there have been plenty of systems and solutions for diagnosing the data center network through comprehensive monitoring [30, 48, 52, 53, 70, 71, 75–77, 81, 83, 85, 89] or opportunistic probing [29, 32, 34, 38, 41, 47, 56, 78, 88]. For example, EverFlow [89] performs fine-grained packet matching to identify the root cause of network issues, while PINT [32] probabilistically samples packets to achieve in-band telemetry. However, so far there is no method for efficient and effective monitoring and diagnosing of the reliability of the containerized model training. SkeletonHunter is the first-of-its-kind work to achieve the goal by leveraging the inherent characteristics of containerized model training to enable fine-grained, accurate monitoring and diagnosis while minimizing overhead.

Troubleshooting Large Model Training. Developing efficient and effective diagnosis systems for large-scale model training is now an important research area in the community. Some of the existing approaches gather customized monitoring data during training process from mainstream training frameworks (e.g., MegaScale [50], TorchProfiler [27], and Dynolog [7]) or collective communication processes (e.g., C4 [39] and Holmes [63]). However, these methods often require considerable user intervention or training framework modifications, and sometimes can even lead to performance degradation. On the other hand, SuperBench [84] proposes a comprehensive benchmark but is limited to offline diagnosis. In contrast, SkeletonHunter is designed to diagnose network anomalies at runtime without the modifications to the RDMA and collective communication libraries, which operates transparently to users.

10 Conclusion

Despite the great benefits of containerized large model training, it is challenging for CSPs to ensure the reliability of the network infrastructure for it. Existing monitoring and diagnosing tools for data centers are inefficient or inaccurate in such a new scenario. In this paper, we present SkeletonHunter, a monitoring and diagnosis system for containerized large model training. It infers *traffic skeletons* to reduce monitoring overhead while achieving high accuracy in network failure detection and localization. After its deployment, SkeletonHunter has uncovered thousands of network failures, significantly improving the network reliability and reducing operational costs. In a broader sense, our work provides a promising direction to infer the traffic patterns of the users' workloads for efficient management of large-scale data center networks.

Acknowledgments

We acknowledge Container Network, High-Performance Network, and PAI teams in Alibaba Cloud that contributed to the success of SkeletonHunter. We thank our shepherd, Sujata Banerjee, and the anonymous reviewers for their valuable comments and suggestions. This work is supported in part by the National Key R&D Program of China under grant 2022YFB4500703, the NSFC under grants 62332012 and 62472245, and the Alibaba Research Intern Program.

References

[1] 2014. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. https://datatracker.ietf.org/doc/html/rfc7348. (2014).
[2] 2018. Lognormal Distribution. https://www.sciencedirect.com/topics/engineering/lognormal-distribution. (2018).
[3] 2020. IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2019* (2020), 1–499.
[4] 2022. Doubling All2all Performance with NVIDIA Collective Communication Library 2.12. https://developer.nvidia.com/blog/doubling-all2all-performance-with-nvidia-collective-communication-library-2-12/. (2022).
[5] 2023. 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).
[6] 2024. Container Network Interface (CNI). https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/. (2024).
[7] 2024. Dynolog: A Performance Monitoring Daemon for Heterogeneous CPU-GPU Systems. https://github.com/facebookincubator/dynolog. (2024).
[8] 2024. GPT-3 Powers the Next Generation of Apps. https://openai.com/index/gpt-3-apps/. (2024).
[9] 2024. Networking Overview. https://docs.docker.com/engine/network/. (2024).
[10] 2025. Azure Machine Learning Documentation. https://learn.microsoft.com/en-us/azure/machine-learning/?view=azureml-api-1. (2025).
[11] 2025. Configuring Distributed Training for PyTorch. https://cloud.google.com/ai-platform/training/docs/distributed-pytorch. (2025).
[12] 2025. The Container Security Platform. https://gvisor.dev/. (2025).
[13] 2025. DataParallel — PyTorch 2.7 Documentation. https://docs.pytorch.org/docs/stable/generated/torch.nn.DataParallel.html. (2025).
[14] 2025. DeepSeek-V3. https://github.com/deepseek-ai/DeepSeek-V3. (2025).
[15] 2025. Docker Containers for Training and Deploying Models. https://docs.aws.amazon.com/sagemaker/latest/dg/docker-containers.html. (2025).
[16] 2025. Environment Variables — NCCL 2.25.1 Documentation. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-ib-timeout. (2025).
[17] 2025. Industry Leading, Open-Source AI | Llama by Meta. https://www.llama.com/. (2025).
[18] 2025. Microsoft/Msccl: Microsoft Collective Communication Library. https://github.com/microsoft/msccl. (2025).
[19] 2025. Mixtral of Experts a High Quality Sparse Mixture-of-Experts. https://mistral.ai/news/mixtral-of-experts/. (2025).
[20] 2025. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl. (2025).
[21] 2025. Openucx/Ucc. UCX. (2025).
[22] 2025. Qwen3. https://github.com/QwenLM/Qwen3. (2025).
[23] 2025. Realtime Compute for Apache Flink - Alibaba Cloud. https://www.alibabacloud.com/product/realtime-compute. (2025).
[24] 2025. Simple Log Service. https://www.alibabacloud.com/help/en/sls/. (2025).
[25] 2025. The Speed of Containers, the Security of VMs. https://katacontainers.io/. (2025).
[26] 2025. TensorFlow. https://www.tensorflow.org/. (2025).
[27] 2025. Torch.Profiler — PyTorch 2.7 Documentation. https://docs.pytorch.org/docs/stable/profiler.html. (2025).
[28] Jont B Allen and Lawrence R Rabiner. 1977. A Unified Approach to Short-Time Fourier Analysis and Synthesis. *Proc. IEEE* 65, 11 (1977), 1558–1564.
[29] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 2018. 007: Democratically Finding the Cause of Packet Drops. In *Proc. of USENIX NSDI*. 419–435.
[30] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. 2016. Taking the Blame Game Out of Data Centers Operations with NetPoirot. In *Proc. of ACM SIGCOMM*. 440–453.
[31] Richard C Aster, Brian Borchers, and Clifford H Thurber. 2018. *Parameter Estimation and Inverse Problems*. Elsevier.
[32] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-Band Network Telemetry. In *Proc. of ACM SIGCOMM*. 662–680.
[33] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. 2000. LOF: Identifying Density-Based Local Outliers. In *Proc. of ACM SIGMOD*. 93–104.
[34] Tobias Bühler, Romain Jacob, Ingmar Poese, and Laurent Vanbever. 2023. Enhancing Global Network Monitoring with Magnifier. In *Proc. of USENIX NSDI*.
[35] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. 2021. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *Proc. of ACM SOSP*. 228–242.
[36] Milind Chabbi and Murali Krishna Ramanathan. 2022. A Study of Real-World Data Races in Golang. In *Proc. of ACM PLDI*. 474–489.
[37] Weiwei Chu, Xinfeng Xie, Jiecao Yu, Jie Wang, Amar Phanishayee, Chunqiang Tang, Yuchen Hao, Jianyu Huang, Mustafa Ozdal, Jun Wang, Vedanuj Goswami, Naman Goyal, Abhishek Kadian, Andrew Gu, Chris Cai, Feng Tian, Xiaodong Wang, Min Si, Pavan Balaji, Ching-Hsiang Chu, and Jongsoo Park. 2025. Scaling Llama 3 Training with Efficient Parallelism Strategies. In *Proc. of ACM ISCA*. 1703–1716.
[38] Rui Ding, Xunpeng Liu, Shibo Yang, Qun Huang, Baoshu Xie, Ronghua Sun, Zhi Zhang, and Bolong Cui. 2024. RD-Probe: Scalable Monitoring with Sufficient Coverage in Complex Datacenter Networks. In *Proc. of ACM SIGCOMM*. 258–273.
[39] Jianbo Dong, Bin Luo, Jun Zhang, Pengcheng Zhang, Fei Feng, Yikai Zhu, Ang Liu, Zian Chen, Yi Shi, Hairong Jiao, Gang Lu, Yu Guan, Ennan Zhai, Wencong Xiao, Hanyu Zhao, Man Yuan, Siran Yang, Xiang Li, Jiamang Wang, Rui Men, Jianwei Zhang, Chang Zhou, Dennis Cai, Yuan Xie, and Binzhang Fu. 2025. Enhancing Large-Scale AI Training Efficiency: The C4 Solution for Real-Time Anomaly Detection and Communication Optimization. (2025). arXiv:cs/2406.04594 https://arxiv.org/abs/2406.04594
[40] Yaozu Dong, Yu Chen, Zhenhao Pan, Jinquan Dai, and Yunhong Jiang. 2012. ReNIC: Architectural Extension to SR-IOV I/O Virtualization for Efficient Replication. *ACM Transactions on Architecture and Code Optimization* 8, 4, Article 40 (2012).
[41] Chongrong Fang, Haoyu Liu, Mao Miao, Jie Ye, Lei Wang, Wansheng Zhang, Daxiang Kang, Biao Lyv, Peng Cheng, and Jiming Chen. 2020. VTrace: Automatic Diagnostic System for Persistent Packet Loss in Cloud-Scale Overlay Network. In *Proc. of ACM SIGCOMM*. 31–43.
[42] Marie Farge. 1992. Wavelet Transforms and Their Applications to Turbulence. *Annual Review of Fluid Mechanics* 24, 1 (1992), 395–458.
[43] Jiawei Fei, Chen-Yu Ho, Atal N. Sahu, Marco Canini, and Amedeo Sapio. 2021. Efficient Sparse Collective Communication and Its Application to Accelerate Distributed Deep Learning. In *Proc. of ACM SIGCOMM*. 676–691.
[44] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proc. of Springer PVM/MPI*. 97–104.
[45] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. 2024. RDMA Over Ethernet for Distributed Training at Meta Scale. In *Proc. of ACM SIGCOMM*. 57–70.
[46] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA Over Commodity Ethernet at Scale. In *Proc. of ACM SIGCOMM*. 202–215.
[47] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, Varugis Kurien, and Midfin Systems. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proc. of ACM SIGCOMM*. 139–152.
[48] Vipul Harsh, Wenxuan Zhou, Sachin Ashok, Radhika Niranjan Mysore, Brighten Godfrey, and Sujata Banerjee. 2023. Murphy: Performance Diagnosis of Distributed Cloud Applications. In *Proc. of ACM SIGCOMM*. 438–451.
[49] Pao-Lu Hsu and Herbert Robbins. 1947. Complete Convergence and the Law of Large Numbers. *Proceedings of the National Academy of Sciences* 33, 2 (1947), 25–31.
[50] 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. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *Proc. of USENIX NSDI*. 745–760.
[51] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Proc. of USENIX ATC*. 437–450.
[52] Pravein Govindan Kannan, Nishant Budhdev, Raj Joshi, and Mun Choon Chan. 2021. Debugging Transient Faults in Data Centers Using Synchronized Network-Wide Packet Histories. In *Proc. of USENIX NSDI*. 253–268.
[53] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. 2019. Confluo: Distributed Monitoring and Diagnosis Stack for High-Speed Networks. In *Proc. of USENIX NSDI*. 421–436.

[54] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. 2020. PANIC: A High-Performance Programmable NIC for Multi-Tenant Networks. In *Proc. of USENIX OSDI*. 243–259.

[55] Juncai Liu, Jessie Hui Wang, and Yimin Jiang. 2023. Janus: A Unified Distributed Training Framework for Sparse Mixture-of-Experts Models. In *Proc. of ACM SIGCOMM*. 486–498.

[56] 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. 2024. R-Pingmesh: A Service-Aware RoCE Network Monitoring and Diagnostic System. In *Proc. of ACM SIGCOMM*. 554–567.

[57] Wei Liu, Kun Qian, Zhenhua Li, Feng Qian, Tianyin Xu, Yunhao Liu, Yu Guan, Shuhong Zhu, Hongfei Xu, Lanlan Xi, Chao Qin, and Ennan Zhai. 2025. Mitigating Scalability Walls of RDMA-based Container Networks. In *Proc. of USENIX NSDI*.

[58] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proc. of ACM SIGCOMM*. 101–114.

[59] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. 2022. Software-Hardware Co-Design for Fast and Scalable Training of Deep Learning Recommendation Models. In *Proc. of ACM ISCA*. 993–1011.

[60] Fionn Murtagh and Pedro Contreras. 2012. Algorithms for Hierarchical Clustering: An Overview. *WIREs Data Mining and Knowledge Discovery* 2, 1 (2012), 86–97.

[61] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proc. of ACM SC*.

[62] Whitney K Newey and Daniel McFadden. 1994. Large Sample Estimation and Hypothesis Testing. *Handbook of Econometrics* 4 (1994), 2111–2245.

[63] Pranoy Panda, Ankush Agarwal, Chaitanya Devaguptapu, Manohar Kaul, and Prathosh A P. 2024. HOLMES: Hyper-Relational Knowledge Graphs for Multi-hop Question Answering using LLMs. (2024). arXiv:cs.CL/2406.06027 https://arxiv.org/abs/2406.06027

[64] Yanghua Peng, Ji Yang, and Chuan Wu. 2017. deTector: A Topology-aware Monitoring System for Data Center Networks. In *Proc. of USENIX ATC*.

[65] Teresa Pepe and Marzio Puleri. 2015. Network Tomography: A Novel Algorithm for Probing Path Selection. In *Proc. of IEEE ICC*. 5337–5341.

[66] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martın Casado. 2015. The Design and Implementation of Open vSwitch. In *Proc. of USENIX NSDI*.

[67] 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. 2024. Alibaba HPN: A Data Center Network for Large Language Model Training. In *Proc. of ACM SIGCOMM*. 691–706.

[68] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proc. of ACM SIGKDD*. 3505–3506.

[69] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. 2014. Planck: Millisecond-Scale Monitoring and Control for Commodity Networks. In *Proc. of ACM SIGCOMM*. 407–418.

[70] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren. 2017. Passive Realtime Datacenter Fault Detection and Localization. In *Proc. of USENIX NSDI*. 595–612.

[71] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. 2022. Continuous In-Network Round-Trip Time Monitoring. In *Proc. of ACM SIGCOMM*. 473–485.

[72] Enge Song, Yang Song, Chengyun Lu, Tian Pan, Shaokai Zhang, Jianyuan Lu, Jiangu Zhao, Xining Wang, Xiaomin Wu, Minglan Gao, Zongquan Li, Ziyang Fang, Biao Lyu, Pengyu Zhang, Rong Wen, Li Yi, Zhigang Zong, and Shunmin Zhu. 2024. Canal Mesh: A Cloud-Scale Sidecar-Free Multi-Tenant Service Mesh Architecture. In *Proc. of ACM SIGCOMM*. 860–875.

[73] Brent Stephens, Aditya Akella, and Michael M. Swift. 2018. Your Programmable NIC Should Be a Programmable Switch. In *Proc. of ACM HotNets*. 36–42.

[74] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency. In *Proc. of ACM ISCA*. 1–15.

[75] Haifeng Sun, Jiaheng Li, Jintao He, Jie Gui, and Qun Huang. 2023. OmniWindow: A General and Efficient Window Mechanism Framework for Network Telemetry. In *Proc. of ACM SIGCOMM*. 867–880.

[76] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2016. Simplifying Datacenter Network Debugging with Pathdump. In *Proc. of USENIX OSDI*. 233–248.

[77] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2018. Distributed Network Monitoring and Debugging with SwitchPointer. In *Proc. of USENIX NSDI*. 453–456.

[78] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. 2019. NetBouncer: Active Device and Link Failure Localization in Data Center Networks. In *Proc. of USENIX NSDI*. 599–614.

[79] Abraham Wald. 2004. *Sequential Analysis*. Courier Corporation.

[80] Siqi Wang, Zhengyu Chen, Bei Li, Keqing He, Min Zhang, and Jingang Wang. 2024. Scaling Laws Across Model Architectures: A Comparative Analysis of Dense and MoE Models in Large Language Models. In *Proc. of ACL EMNLP*. 5583–5595.

[81] Weitao Wang, Xinyu Crystal Wu, Praveen Tammana, Ang Chen, and T S Eugene Ng. 2022. Closed-Loop Network Performance Monitoring and Diagnosis with SpiderMon. In *Proc. of USENIX NSDI*. 267–285.

[82] Shmuel Winograd. 1978. On Computing the Discrete Fourier Transform. *Math. Comp.* 32, 141 (1978), 175–199.

[83] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. 2019. Zeno: Diagnosing Performance Problems with Temporal Provenance. In *Proc. of USENIX NSDI*. 395–420.

[84] 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. 2024. SuperBench: Improving Cloud AI Infrastructure Reliability with Proactive Validation. In *Proc. of USENIX ATC*. 835–850.

[85] Kaicheng Yang, Yuhan Wu, Ruijie Miao, Tong Yang, Zirui Liu, Zicang Xu, Rui Qiu, Yikai Zhao, Hanglong Lv, Zhigang Ji, and Gaogang Xie. 2023. ChameleMon: Shifting Measurement Attention as Network State Changes. In *Proc. of ACM SIGCOMM*. 881–903.

[86] Dong Young Yoon, Yang Wang, Miao Yu, Elvis Huang, Juan Ignacio Jones, Abhinay Kukkadapu, Osman Kocas, Jonathan Wiepert, Kapil Goenka, Sherry Chen, Yanjun Lin, Zhihui Huang, Jocelyn Kong, Michael Chow, and Chunqiang Tang. 2024. FBDetect: Catching Tiny Performance Regressions at Hyperscale through In-Production Monitoring. In *Proc. of ACM SOSP*. 522–540.

[87] Zhuolong Yu, Bowen Su, Wei Bai, Shachar Raindel, Vladimir Braverman, and Xin Jin. 2023. Understanding the Micro-Behaviors of Hardware Offloaded Network Stacks with Lumina. In *Proc. of ACM SIGCOMM*. 1074–1087.

[88] 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. 2020. Flow Event Telemetry on Programmable Data Plane. In *Proc. of ACM SIGCOMM*. 76–89.

[89] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, Haitao Zheng, and U C Santa Barbara. 2015. Packet-Level Telemetry in Large Datacenter Networks. In *Proc. of ACM SIGCOMM*. 479–491.

[90] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, Steve Lacy, Hang Wang, Aaron Wisner, Chris Lewis, and Henri Bahini. 2024. Resiliency at Scale: Managing Google's TPUv4 Machine Learning Supercomputer. In *Proc. of USENIX NSDI*. 761–774.