Minder: Faulty Machine Detection for Large-scale Distributed Model Training

Yangtao Deng[1], Xiang Shi[2], Zhuo Jiang[2], Xingjian Zhang[1], Lei Zhang[2]
Zhang Zhang[2], Bo Li[2], Zuquan Song[2], Hang Zhu[2], Gaohong Liu[2]
Fuliang Li[3], Shuguang Wang[2], Haibin Lin[2], Jianxi Ye[2], Minlan Yu[4]
[1]*Tsinghua University* [2]*ByteDance* [3]*Northeastern University* [4]*Harvard University*

Abstract

Large-scale distributed model training requires simultaneous training on up to thousands of machines. Faulty machine detection is critical when an unexpected fault occurs in a machine. From our experience, a training task can encounter two faults per day on average, possibly leading to a halt for hours. To address the drawbacks of the time-consuming and labor-intensive manual scrutiny, we propose Minder, an automatic faulty machine detector for distributed training tasks. The key idea of Minder is to automatically and efficiently detect faulty distinctive monitoring metric patterns, which could last for a period before the entire training task comes to a halt. Minder has been deployed in our production environment for over one year, monitoring daily distributed training tasks where each involves up to thousands of machines. In our real-world fault detection scenarios, Minder can accurately and efficiently react to faults within 3.6 seconds on average, with a precision of 0.904 and F1-score of 0.893.

1 Introduction

Recent years have witnessed a rapid increase in dataset sizes and the number of parameters in models, especially in Large Language Models (LLMs). The GPT-4 model [18], an instance of the Mixture-of-Experts (MoE) paradigm, demonstrates this growth with its 1.8T parameters. Other latest models also exhibit this trend, with parameter counts exceeding 500 billion [24, 69]. The feasibility of training such extensive models efficiently has been realized through large-scale machines and GPUs [39, 40]. It has also been accompanied by advancements in distributed model training [49, 68], high-performance collective communication [48, 65], and fault-tolerant techniques [36]. A system of such vast size and complexity involves a huge amount of computation, communication, and storage resources as well as software support for a task. Consequently, the potential for faults is high, leading to the possibility of task failure.

Faulty machine detection thus has become a significant bottleneck in the maintenance of distributed tasks. In our production environment, an accidental hardware or software fault occurs twice a day on average. The entire task may be forced to stop for hours or days until fixed for retraining. The economic loss for a customer can reach more than $1700 in a 128-machine task for 40 minutes (case in 2.1). Training a GPT-2 model with 1.5 billion parameters and 40GB dataset [67], for instance, takes 200 days utilizing an NVIDIA V100 GPU [10] (or 12 days for a DGX-2H server). If the training process is frequently interrupted by such faults, operating expenses and time costs will increase significantly.

However, the current manual diagnosis method is unsatisfactory. Once a halted task notification is received, the engineer needs to check the training parameters. Meanwhile, engineers from the training, physical networking, storage, and hardware teams, are also involved in diagnosis, since a fault can occur in any machine component. Examining machine logs and conducting offline performance tests on relevant hardware devices are required until the fault is detected (usually for hours). Delayed notifications, incomplete log content, and the complicated process of manual diagnosis amplify the unpredictability of time and labor costs.

It's necessary to design effective and accurate faulty machine detection methods that can quickly react to faults at runtime, not only providing better reliability but also eliminating manual efforts. Achieving such goals is challenging because a machine can fail due to various types of faults. These faults can occur in any possible component, including hardware and software, and can be intra-host or inter-host. Besides, the abnormal pattern of monitoring metrics varies from task to task, making the traditional supervised anomaly detection methods impractical, because even the same behavior might be abnormal in a task with a different workload and machine scale. Additionally, there isn't an individual monitoring metric that necessarily signals a fault. For instance, CPU or GPU usage is the most sensitive metric for fault in-

The first two authors contributed equally to this paper. Zhuo Jiang and Minlan Yu are the corresponding authors. This work was done while Yangtao Deng and Xingjian Zhang were doing a joint research project at ByteDance.

dication, based on our observation from real production data. However, neither one is guaranteed to identify the faulty machine for Error Correction Code (ECC) errors. If noises exist in monitoring data, the detection may be even misguided. Therefore, faulty machine detection for distributed training is challenging.

Instead of creating a monolithic predictor with available monitoring data, we developed Minder by leveraging the ideas of similarity (3.1), continuity (3.2), training individual models for each monitoring metric (3.3), and metric prioritization (3.4). Minder resolves the challenges by recognizing that a machine with a fault displays an abnormal pattern in certain metric data that differs from other machines and lasts for a duration. We also train individual models for data denoising. We then track the dissimilarity between the denoised data for each machine and monitor its duration. By repeating this to individual metrics, the faulty machine is detected. To further expedite detection, we prioritize metrics to identify the most sensitive ones when a fault occurs.

We designed, implemented, and deployed Minder for all the distributed training tasks. Minder operates without interrupting the running of the training machines, only requiring the pulling of monitoring data from the Data APIs for backend run-time analysis. Host metrics used by Minder cover the aspects of computation, communication, and storage. Manual labors are released from the debugging process since Minder can react within 3.6 seconds (6.1), reducing over 99% of the time of manual debugging (shorter time by 500 ×). Minder has an overall precision and F1-score of 0.904 and 0.893.

We make the following contributions.

- An investigation of the fault types and their correlations with various monitoring metrics (2.3). We empirically explain why some metrics are more sensitive to faults and outline the challenges for the detection (2.4).

- The ideas of similarity, continuity, denoising models, and metric prioritization for the design of Minder (3, 4). Our thorough evaluation of Minder's implementation (5) and and ablation experiments highlight its fast reaction, high accuracy, and proper design choices (6).

- Lessons we encountered when deploying Minder in practical (7). We also point out future directions.

2 Motivation

2.1 Negative Impacts of Faults in Real-World Production Environments

Once an unexpected hardware or software fault occurs in a machine, it does harm to the entire distributed training task at the machine level.

Faults are frequent due to the long training duration and large scale. The relationship between a task's machine scale and the daily number of faults is illustrated in Figure 1. Based on our seven-month statistics, the machine scale could significantly exceed 1024, training on more than 10000 GPUs. The occurrence of unexpected faults is highly correlated with the task scale, with an average of two faults a day. If we cannot react quickly to these incidents, such a frequent fault is a burden for training large models.

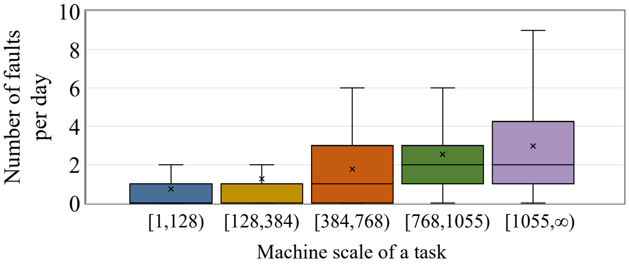

Figure 1: Fault frequency of tasks with different machine scale sizes.

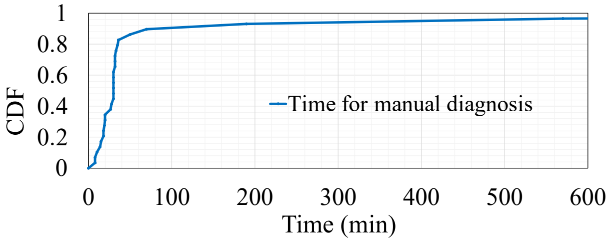

Figure 2: Time for task diagnosis in seven months.

A single fault can cause a large-scale task halt. Based on empirical observations, a fault during the training process often originates as a host problem (*e.g.,* CUDA error, NVLink error [5]). However, this issue can eventually cause a cascade effect, leading to the entire task being interrupted or a noticeable slowdown. For example, a hardware ECC error during communication will make the distributed systems interrupt (*e.g.,* when running in PS [48] or All-reduce [65]) due to NCCL timeout [12] or network disconnections, thereby could resulting in more than half thousand of machines to stop and be in idle state in the production environment.

Faults take a long time to diagnose, increasing labor and resource costs. Based on seven-month data in 2023, the time until the faulty machine can be manually diagnosed is shown in Figure 2. The time lasts over half an hour on average and can be days. It ultimately results in GPU and NIC resources being left idle, leading to additional time costs upon resolution. More delays can result in significantly increased costs and time requirements for users.

An example: PCIe downgrading. We highlight a real-world instance where a 128-machine training was forced to slow down severely for 40 minutes due to a PCIe degradation (PCIe is an intra-host, high-speed serial bus that connects a variety of devices, such as graphics cards, NICs, and solid-state drives). The faulty machine encountered PCIe degradation from 800Mbps to 500Mbps. The congested communication ultimately results in the underutilization of computational resources, where thousands of V100 GPUs were compelled to remain slowed down for 40 minutes. Such cost wastes for customers can be $650 for 40 minutes (given the public renting price of $2.48 per GPU for an NVIDIA V100 [6]).

2.2 Today's Solution and Drawbacks

A straightforward approach after an unexpected fault occurs is inspecting the existing hardware and software logs on the machines, as well as `dmesg` [2], `netstat`, `top` commands. However, this process involves the following three limitations.

The notification of when to trigger a diagnosis is not timely. Firstly, engineers are only alerted once the task has stopped entirely. Certain faults will slow down the training speed in Model FLOPs Utilization (MFU) but are not severe enough to stop the task. The current approach fails to detect degradation in performance as long as the task continues to run. The task will continue running with deteriorated performance for a period in the PCIe downgrading example.

Scrutinized content is incomplete or redundant. Upon a fault notification, the logs or counters recorded during training will be reviewed. The logs are typically maintained in plain text, including built-in software-layer logs (*e.g.,* NCCL and CUDA logs), hardware-layer logs, and network logs. Previous knowledge and experience guide the decision on which logs to include and check, resulting in some log content being overlooked. Meanwhile, logs do not include monitoring metrics like GPU power, temperature, and NVLink bandwidth. In the PCIe degradation case (2.1), identifying the faulty machine was hard as critical monitoring data, like Priority-based Flow Control (PFC) packet rates, was not promptly inspected from the logs. Besides, the log content frequently includes redundant data, such as environmental parameters and warnings. The detection time will be lengthened.

The diagnosis analysis is a complicated and time-consuming process. Engineers from multiple teams are involved in the diagnosing process. Figure 2 presents the time-consuming diagnosis process. It could take as long as several days to detect the faulty machine.

Manual diagnosis procedure and fault propagation for the PCIe downgrading case (2.1). The detection took 40 minutes in total and involved multiple teams. The engineer in charge scrutinized model-related information, parallelism settings [40], dependencies, environmental, and framework (*e.g.,* Megatron-LM [68]) parameters. Meanwhile, the network team scrutinized intra-host throughput, Remote Direct Memory Access (RDMA) traffic, packet loss/randomness, congestion indicators, drivers, and routing. The storage and hardware teams inspected HDFS&SSD usage, GPU&CPU usage, NIC health, and machine scheduling.

The fault propagation initiated from PCIe downgrading to PFC surge. The NIC buffer of the faulty machine was filled after the PCIe degraded. The consequent bottleneck inter-host communication caused a PFC Tx packet surge. The congestion also raised both Explicit Congestion Notifications (ECN) received and Congestion Notification Packets (CNP) sent [17]. As a result, the NIC throughput across all machines dropped from 6.5Gbps to 4.9Gbps. Reduced computation data led to declined GPU tensor core usage [7]. Thus, the training performance was downgraded.

2.3 Real-world Faulty Case Studies

To address the drawbacks of the manual log analysis, we first conducted an in-depth review of the fault types that occurred over seven months. Table 1 shows the common types of faults, their frequencies, and the proportion of instances for each fault type that a metric could indicate. The proportions are determined empirically by examining the available instances and quantifying the number that exhibited abnormal patterns in the monitoring data following a fault. The monitoring data contains multiple metrics and is sampled per second.

We come up with the following observations. Firstly, hardware faults make up the majority of faults (55.8%), in which ECC errors constitute a large proportion (38.9%). Errors that happened in CUDA or GPU also make up a large proportion. Unfortunately, these errors are hard to predict or avoid.

Moreover, each metric displays varying probabilities of indicating a type of fault. There isn't a single metric that effectively signals all of them. ECC error, NIC dropout, GPU card drop, NVLink error, CUDA, and GPU execution error strongly correlate with CPU or GPU metrics. Likewise, PCIe downgrading, NIC dropout, and machine unreachable are most relevant to network metrics, such as PFC and throughput. We explain these observations from our experience. For the CPU metric, a fault on a machine will cease the CPU process, reducing its CPU usage. However, the other machines retain their processes for a while, waiting for data synchronization, due to the Kubernetes management, NCCL timeout setting, and heartbeat mechanism setting with a predefined time window. Thus, these machines maintain normal CPU usage before the task halts. For the GPU metric, a GPU drop or process kill during computation leads to low GPU usage. Conversely, other machines carry on running their CUDA kernels before the timeout and heartbeat check, maintaining normal GPU usage. For the PFC metric, PFC signals on the faulty machine surge abruptly when the NIC buffer is filled, due to congestion-related network errors. Other machines exhibit a low number of PFC packets. The same trend applies to memory usage. Nonetheless, disk usage does not exhibit significant fluctuations based on our experience. Therefore, most fault types strongly correlate with CPU, GPU, Memory, and PCIe metrics. As an exception, AOC errors happen on the switch or the machine-side cables. If a switch AOC error occurs, machines connected to this switch port will be affected instantly. Such a large scale of affected machines can quickly propagate adverse effects to others. It is hard to capture abnormal patterns with second-level monitoring data.

Table 1: Fault types and the proportion of instances for each fault type being indicated by a metric.

Fault type		**Frequency of each fault type**	**Metrics**					
			CPU	**GPU**	**PFC**	**Throughput**	**Disk**	**Memory**
Intra-host hardware faults (55.8%)	**ECC error**	**38.9%**	80.0%	65.7%	8.6%	45.7%	11.4%	57.1%
	PCIe downgrading	**6.6%**	0.0%	8.3%	100%	33.3%	8.3%	0.0%
	NIC dropout	**5.7%**	100%	100%	0.0%	100%	0.0%	100%
	GPU card drop	**2.0%**	75.0%	70.0%	5.0%	50.0%	20.0%	55.0%
	NVLink error	**1.7%**	83.3%	50.0%	16.7%	50.0%	0.0%	66.7%
	AOC error	**0.9%**	25.0%	25.0%	0.0%	25.0%	25.0%	25.0%
Intra-host software faults (28.0%)	**CUDA execution error**	**14.6%**	61.9%	57.1%	19.0%	33.3%	14.3%	61.9%
	GPU execution error	**7.7%**	50.0%	71.4%	14.3%	42.9%	21.4%	42.8%
	HDFS error	**5.7%**	57.1%	57.1%	0.0%	14.3%	0%	14.3%
Inter-host network faults (6.0%)	**Machine unreachable**	**6.0%**	47.4%	63.2%	0.0%	53.6%	26.3%	15.8%
Others (10.3%)	-	**10.3%**	-	-	-	-	-	-

Note: the introduction of each fault type is in Appendix A.

2.4 Challenges

Based on the observations, we discovered that the following challenges must be addressed:

Challenge 1: Any machine could fail in various ways. Advanced machines, such as Nvidia DGX-A100 [11], incorporate as many as 8 Nvidia A100 GPUs and 4 Mellanox 200 Gb/s RDMA NICs (RNIC), all of which are potential fault points. As presented in Table 1, a fault could happen from intra-host computation, and communication, to inter-host networks, or from hardware (GPU, CPU, PCIe, NVLink, RNIC, memory, and disk) to software communication libraries (*e.g.,* NCCL), and training frameworks (*e.g.,* CUDA). It is hard to detect a faulty machine under all the unknown circumstances.

Challenge 2: The normal state of a monitoring metric is task-dependent. Training tasks have various machine scales, data sizes, models, and training frameworks. Thus, a monitoring metric may have different normal states for various tasks. For instance, a GPU temperature of 70 degrees Celsius is abnormal where GPU clock frequency is 1350MHz, but is regarded as normal in a task where GPUs work with a high clock frequency of 1800MHz. Traditional supervised anomaly detection is inappropriate for differentiating between normal and faulty machine states because the same input monitoring data can be labeled as either normal or abnormal.

Challenge 3: The correlation between fault types and monitoring metrics is not necessarily one-to-one. On one hand, a metric anomaly may be caused by various fault types. For example, a decrease in CPU usage could be caused by ECC error, NVLink error, and other faults. On the other hand, for a fault type, there isn't a single metric that necessarily signals it. As an example in Table 1, ECC errors could potentially be noticed by either CPU or GPU usage of the faulty machine, yet neither metric guarantees nor necessarily both. Thus, a single fault type can be manifested via multiple abnormal metrics, with no solitary metric providing a guaranteed indication. Instead, the metrics exhibit an "or" correlation for a fault type. Consequently, we cannot simply use a model that incorporates the data from all metrics. Rather, we use individual per-metric models for detection (analysis in 6.3).

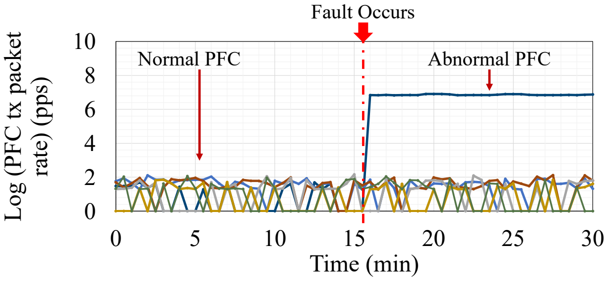

Figure 3: PFC tx packet rate pattern for each machine before and after a fault occurs.

Challenge 4: Noises exist in time series monitoring data. The monitoring data inevitably consists of noises due to jitters, inaccurate sensors, temperature, timestamp misalignment, network interruptions, or other issues. Short-term noises will mislead us to regard a machine as the faulty one, resulting in extra time and labor burden. Thus, the raw data cannot be used directly for detection (analysis in 6.3).

3 Design Overview

In this section, we introduce four design choices to address the challenges, revolving around Minder: an automatic, responsive, and accurate watcher to detect the machine with an unpredicted fault during distributed training that leads to a task halt. We examine the dissimilarity (3.1) among machines. Then we evaluate the continuity (3.2) of the detected faulty machine candidate for filtering out bursty jitters. Since metric data contains noises, we use models to denoise and reconstruct the raw data. Specifically, individual models corresponding to various monitoring metrics are trained, instead of integrating them into a single model (3.3). We also prioritize the metrics (3.4). The faulty machine can be detected swiftly by the top-prioritized metrics and corresponding models.

3.1 Machine-level Similarity

To address challenges 1 and 2, we notice that machine-level metric data exhibit similar behavior in parallel distributed training [68] at the second level.

Data parallelism (DP) operates by splitting data equally among GPUs that store duplicate model parameters and optimizer states. Pipeline parallelism (PP) assigns model layers to multiple GPUs in a pipelined manner. In tensor parallelism (TP), each GPU executes a portion of a computation process to improve hardware utilization in parallel.

By integrating these techniques into a 3D parallelism framework, large-scale model training, such as LLM training [40] or multi-modal training, can be effectively facilitated. TP is typically constrained within a single machine, whereas DP or PP groups involve inter-host communications. The computation, storage, and communication loads are evenly balanced across machines at the second level. Therefore, similar monitoring data fluctuations will be observed across machines. In the PCIe downgrading example, the initial `PFC Tx Packet Rate` patterns are notably uniform for all the machines in Figure 3. However, if a machine undergoes a fault, its monitoring data will display distinctive differences, offering an opportunity for detection. This principle can be extended to other metrics, as demonstrated by the possibilities offered in Table 1. Hence, the denoised metric data from each machine is used for calculating the similarity with others. The machine identified with the longest "distance" from others could be assumed to be faulty, regardless of the fault type.

Why not use a supervised learning model for faulty machine detection? Unlike many supervised learning-based anomaly detection approaches [21, 27, 45, 53], Minder uses unsupervised learning and similarity-based distance check, due to the different problem contexts. Firstly, labeling the data as normal or abnormal is impractical. As mentioned in challenge 2, the abnormal pattern is task-dependent. Different tasks present different normal ranges for the same monitoring metric under varying working conditions. Secondly, our objective is to identify which machine is to be blamed for the unexpected fault. This is not merely a classification problem of distinguishing normal or abnormal cases. Thus, developing a universal model via supervised training is challenging.

3.2 Machine-level Continuity

To further tackle challenge 2, we utilize the notion of abnormality continuity. Typically, abnormal performance persists for a few minutes upon a fault, but jitters typically last for a short duration. As illustrated in Figure 3, the machine with PCIe degradation experienced significantly higher `PFC Tx Packet Rate` for a period compared to other machines. By inspecting fault instances from seven-month data in 2023, the duration of abnormal performance after a fault occurs is depicted in Figure 4. Most abnormal patterns last for over five minutes. Thus, if we recognize a machine displaying such dissimilarity continuously for a period, the machine may be faulty. In the case of raw data containing bursty noises, they will be filtered out (as analyzed in 6.4).

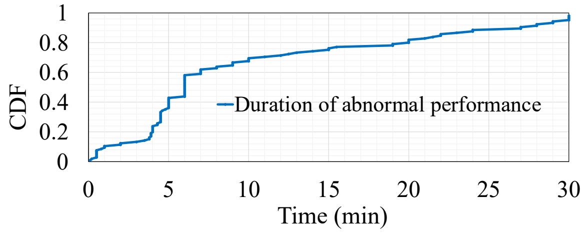

Figure 4: Duration of abnormal performance following a fault.

3.3 Individual Learning-Based Denoising Models for Each Monitoring Metric

For challenge 4, we utilize simple learning models for data denoising and reconstruction, in addition to basic data alignment and normalization. Variational autoencoder (VAE) and other generative probabilistic models are recognized for learning time-series data patterns and features [52, 70]. They are also known for learning embedding schemes that can infer the generation factors for most of the training data. This makes unsupervised learning particularly suited for modeling normal behavior in an anomaly detection task. As such, raw data is denoised and reconstructed by our learning-based models before being fed into further detection (as analyzed in 6.3).

Meanwhile, based on challenge 3, no single metric provides a guaranteed indication for a specific fault type. The indication probability by a metric varies across different types of faults. For this reason, we opt for training individual models for each monitoring metric. These models and their corresponding metric data are used independently for denoising, similarity, and continuity detection.

We do not merge all potential metrics into a single model for two main reasons. First, the time series of multiple metrics do not fluctuate in the same manner when a fault arises. Moreover, metrics' indication capacity differs even for a particular fault type, and one metric's capability varies across different types. As a result, integrating them into one model could lead to the model being misdirected or confused by the array of metrics (as analyzed in 6.3).

3.4 Prioritized Metric Sequence

To expedite detection, we prioritize metrics and only use the models of the top ones, since plenty of metrics (in Appendix B) could be collected and each could be trained with a model. This ensures we use the metrics with higher indication probabilities earlier. The faulty machine could therefore be detected sooner. If a model and its associated metric data cannot detect a faulty machine, we then move to the next metric

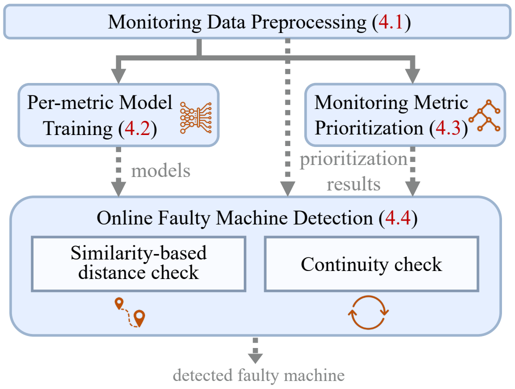
Monitoring Data Preprocessing (4.1)
Per-metric Model Training (4.2)
Monitoring Metric Prioritization (4.3)
models
prioritization results
Online Faulty Machine Detection (4.4)
Similarity-based distance check
Continuity check
detected faulty machine

Figure 5: System architecture of Minder.

and its model, following the prioritization result. We repeat this process until a faulty machine is identified.

4 Minder Framework

The architecture of Minder is shown in Figure 5. Preprocessing (4.1) is required for raw data from each machine. Per-metric Model Training (4.2) and Monitoring Metric Prioritization (4.3) will train models and prioritize the metrics in their sensitivity to faults respectively, as two independent processes. The models and prioritization results are then used in Online Faulty Machine Detection (4.4) for run-time detection.

4.1 Preprocessing

Given a stream of monitoring data from each machine, Minder needs to aggregate them into a series of time windows. Within each window, Minder can do data denoising and machine-level similarity check on a set of metrics. By checking the anomaly continuity from consecutive time windows, Minder detects the faulty machine.

Thus, Minder preprocesses the collected monitoring data if it lacks alignment among certain metrics. Minder first aligns the sampling points across all machines based on the corresponding sampling timestamps. If sample points are missed, Minder uses data from the nearest sampling time for padding.

Normalization is adopted to ensure that the multi-dimensional monitoring data is integrated into an even distribution. Minder normalizes the monitoring data based on the upper and lower limits of each metric, using the Min-Max normalization technique.

4.2 Per-metric Model Training

As specified in 3.3, we train models for the learning-based denoising and reconstruction for subsequent detection. Since no single metric provides a guaranteed indication and a model incorporating various metrics might be misdirected, individual models should be trained for each metric.

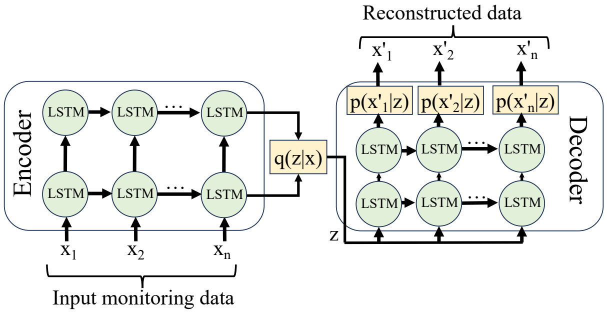

Figure 6: The LSTM-VAE structure for Minder.

The preprocessed per-machine data within a time window is used as input instances to train an unsupervised model. For example, to train a model for `CPU Usage`, we use `CPU Usage` sample data within a time window with a length of w (*e.g.,* 8) and a stride of 1 from each machine of the task. Multiple $1 \times w$ vectors are fed into the model respectively for training. Models for `CPU Usage`, `PFC Packet Rates`, and so on are individually trained. The parameters in the model include *hidden_size* (*e.g.,* 4), *latent_size* (*e.g.,* 8), and *lstm_layer* (*e.g.,* 1). With a model, the time series input of a monitoring metric from a machine could be reconstructed as a denoised vector for this machine.

Training choice: LSTM-VAE. Specifically, VAE models are trained for Minder. VAE is widely used for denoising [70] and compression of high-dimensional features [71], where normal vectors will be reconstructed into similar embeddings while abnormal ones will be reshaped into a more distinctive outlier. This highly effective unsupervised DL technique can reconstruct time series inputs into an arbitrary latent representation without sacrificing original characteristics. Consequently, VAE can enhance the accuracy and robustness of anomaly detection where labeling is absent [78]. When the input consists mainly of normal training state vectors and only a small proportion corresponds to a faulty period, the VAE learns the vector distribution and denoises the jitters.

As depicted in Figure 6, the VAE comprises an encoder and a decoder. The encoder extracts temporal features into a latent space embedding z. Subsequently, the decoder utilizes z to restore the data to a new dimension output as a reconstruction of the distribution. Various statistical or machine learning techniques can be employed in the encoder and decoder to determine the optimal distributions. Given that our data is temporal time series, we utilize LSTM as both the encoder and decoder to extract temporal characteristics [52]. LSTM considers both forward and backward information of a time series to obtain complete correspondence information. As such, LSTM is an ideal choice for VAE.

4.3 Monitoring Metric Prioritization

As introduced in 3.4, we aim to use only the most representative metrics and their models for quick faulty machine detection. Given a large number of metrics, pinpointing the metrics more sensitive to faults is critical. By using the top prioritized metrics and their associated models first, the faulty machine will be detected more quickly.

Consequently, following the steps below, Minder will generate a prioritized list of metrics by their sensitivity to faults. The prioritization results can then be used in run-time detection, specifying which metrics and their models should be used first. Note that this process runs in parallel with the model training process in 4.2.

Step 1: Z-score calculation for evaluating metric sensitivity to faults. To identify the most sensitive metrics, Minder utilizes the Z-score [23], because it depicts the dispersion of data distribution. A metric with a higher Z-score relates to an imbalanced distribution, where a faulty machine shows a dissimilar pattern from others. For a monitoring metric, the Z-score is computed for each machine at a sampling data point from the preprocessed data. For the j-th monitoring metric:

$$Z_{ij} = \frac{x_{ij} - \bar{x_j}}{s_j}$$

where Z_{ij} is the Z-score of the i-th machine, x_{ij} is the sample value of the i-th machine, while $\bar{x_j}$ and s_j are the average value and the standard deviation of all machines on the j-th metric. When a fault occurs, the affected machine exhibits abnormal behavior, leading to outlier samples with high Z-scores.

For a time window of a training task, we use $max(Z_{ij})$ across all the machines for the j-th monitoring metric, indicating the extent of the dispersion among machines.

Step 2: Prioritization of the monitoring metrics. Based on the maximum Z-score for each monitoring metric, Minder uses a decision tree [32, 60] to prioritize the sensitivity of each metric in identifying the faulty machine. We resort to a decision tree for two primary reasons. Firstly, the logical structure of decision trees bears a resemblance to rule-based policies that are common in networking monitoring systems. For example, certain monitors utilize simple threshold rules, such as when `CPU Usage` drops to nearly zero [83]. Secondly, decision trees offer high expressiveness and faithfulness, attributed to their lack of parameters and the ability to represent complex decision-making [22].

To construct a decision tree, Minder gathers the maximum Z-score for each metric from step 1 as an individual instance for the time window of the training task. The instance is labeled manually as normal or abnormal depending on whether a faulty machine exists within this window. Instances across multiple time windows and multiple training tasks are used together to train a decision tree.

As shown in Figure 7, monitoring metrics are prioritized based on their sensitivity to faults. The decision tree employs a step-by-step approach to classify the instances by analyzing the Z-scores of each metric. Nodes located closer to the root of the tree indicate that the corresponding monitoring metrics are more sensitive to the occurrence of a faulty machine. PFC, CPU, GPU, and NVLink-related metrics are identified as the most informative ones. Specifically, `CPU Usage` is relevant to running process states, while the four GPU metrics relate to the computation states. Additionally, `NVLink Bandwidth` and `PFC Packet Rates` are indicative of intra-host and inter-host network quality. The outcome aligns with Table 1, where CPU and GPU enjoy the highest priority.

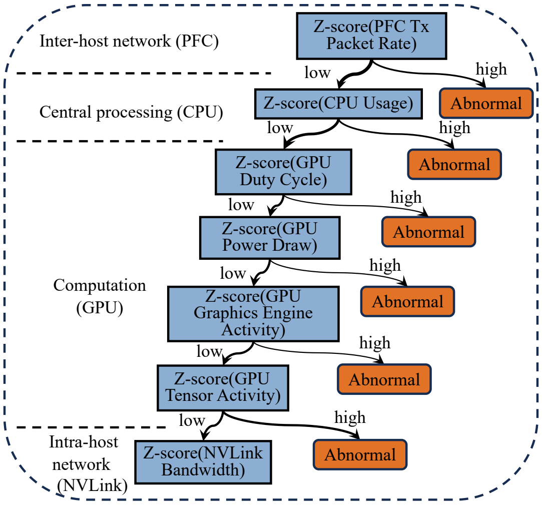

Figure 7: Top 7 layers of the decision tree for prioritization.

4.4 Online Faulty Machine Detection

During run-time detection, Minder leverages the concepts of machine-level similarity (3.1) and continuity (3.2), since the faulty machine tends to exhibit an abnormal pattern over a period. Given the data from each machine, Minder follows the order of the decision tree prioritization results and selects a metric. The metric data is then denoised by the correlated model. Minder then runs step 1&2 introduced below for detection. If none is detected, Minder picks the next metric and repeats step 1&2 until one is detected. Fortunately, if no one is detected after passing all the models, Minder assumes no anomaly occurs up to this time.

Step 1: Similarity-based distance check per time window. To detect a faulty machine during a time window, Minder compares the similarity (3.1) among all the machines for the same metric. In the presence of a faulty machine, its denoised data from LSTM-VAE is distinguishable from others.

To initiate the detection for a time window, the monitoring data from each machine (*e.g.,* a $1 \times w$ vector for a machine) is fed into the corresponding model successively. The reconstructed embedding for the i-th machine is captured by Minder for the following distance calculation. Specifically, Minder calculates the pairwise Euclidean distances of embeddings between every two machines, as it expresses distinct differences between normal and abnormal samples and provides characterizations of various record types [76]. For each machine, Minder calculates the sum of the distances to other machines, representing its dissimilarity. Since the distance magnitude shifts with machine scales, we calculate the normal score for each sum value of the machines to normalize.

The machine with the maximum normal score is probably the faulty one. If the maximum normal score is higher than a *similarity threshold*, the machine is assumed as a candidate of the time window.

Step 2: Continuity check for consecutive time windows. The detected candidate of a time window might be a false alarm due to instant bursts or temporary counter noises, so the idea of continuity (3.2) is essential. This is because faults often lead to deteriorated performance for a period. Minder shifts the time window with a stride of one to detect the potentially faulty machine for new windows. If the same machine is detected with consecutive times that exceed a *continuity threshold*, it is considered a truly faulty machine. A proper *continuity threshold* can be set as 4 minutes as it is adequate to filter out short-term noises and will not exceed the typical lasting time of a deteriorated performance in Figure 3.

5 Implementation

Minder has been deployed in our ML system for a year. It runs on a dedicated machine with two dual-port ConnextX-6 25G-RNICs [9], 128 Intel Xeon Platinum 8336C CPUs, 512G memory, and 1.6T disk. The high-speed RNICs ensure the fast transmission of monitoring data, while computation and storage resources are adequate for real-time detection.

Minder monitors all the ongoing training tasks throughout their life cycles in our production environment. For a task, Minder is called at pre-determined intervals (*e.g.,* every 8 minutes). Upon a call, Minder pulls 15-minute data for the metrics listed in Appendix B from a database for all machines associated with the task. The metrics cover aspects of computation, communication, and storage. The database updates monitoring data per second from all the machines. If Minder identifies a faulty machine, an alert is triggered to a driver and relevant engineers. After the driver submits the machine IP to be blocked and the Pod information to Kubernetes, the faulty machine will be evicted and replaced by a new one, before a fast recovery from recent checkpoints [40]. Importantly, the running of Minder will not interfere with online distributed training tasks, as Minder works as a backend service.

Task workload. The monitored training tasks are distributed across four to over 1000 machines, with GPU numbers up to more than 10000 for a task. Concurrent tasks could be monitored by Minder. Each task is running on machines of homogeneous GPU and RNIC architectures on rail-optimized topology with up to three layers of switches. TP for computation, PP for gradient calculation and propagation, and DP that performs gradient synchronization are efficiently used for our LLM pre-training. These 3D parallelism strategies [40, 68] (3.1) facilitate balanced computation (GPU usage, power, temperature *etc.*), storage (memory usage *etc.*), and communication (intra-host and inter-host throughput) across machines. Models trained in our ML system include the Transformer [74] and so on. The model sizes range from under 32B to over 500B.

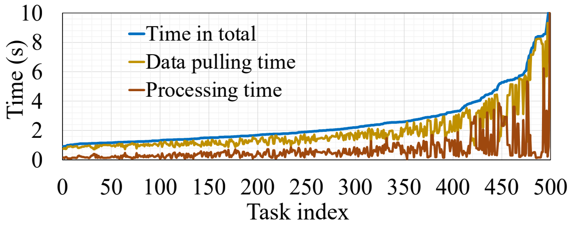

Figure 8: The total data processing time for a call of Minder.

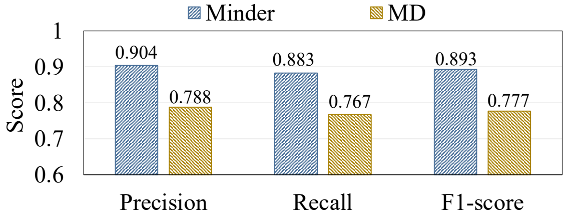

Figure 9: Comparison with a baseline algorithm MD [30, 46].

6 Evaluation

Dataset. Our dataset includes 150 run-time fault instances from online distributed training tasks with 3D parallelism [40]. The dataset includes instances over nine months. Each task involves 4 to over 1500 machines (up to 10,000 NVIDIA Ampere GPUs), covering all the scale groups in Figure 1. 30% of the tasks involve a minimum of 600 machines. All fault types listed in Table 1 are covered. The dominant ones are ECC error (25.7%), CUDA execution error (15%), GPU execution error (10%), and PCIe downgrading (8.6%). Our dataset focuses on faults in an individual machine, as they account for 99% of all faulty cases in the production environment (6.6 for concurrent faulty machine evaluation). The monitoring metrics in Appendix B were collected at the second-level granularity. Due to the fast eviction and recovery process, verifying whether all evicted machines are faulty is challenging. Consequently, our dataset encompasses the run-time instances where the actual faulty machine could be manually confirmed via offline log-checking, nccl-tests, or hardware tests (*e.g.,* [55]). For LSTM-VAE training, we use data from the first three months and the rest for evaluation.

Metrics. For a task, we denote true positives (TP) as the correct machine detection following a fault, and false negatives (FN) as errors in machine detection or missed detections during a fault. True negatives (TN) refer to the correct approvals when machines are running normally, while false positives (FP) refer to false detections when there is no fault. Then we calculate Precision, Recall, and F1-score as our metrics.

6.1 Overall Performance

Total data processing time. In Figure 8, a call of Minder takes 3.6 seconds on average to make an alert. It includes data pulling time (fetching the pertinent monitoring data from

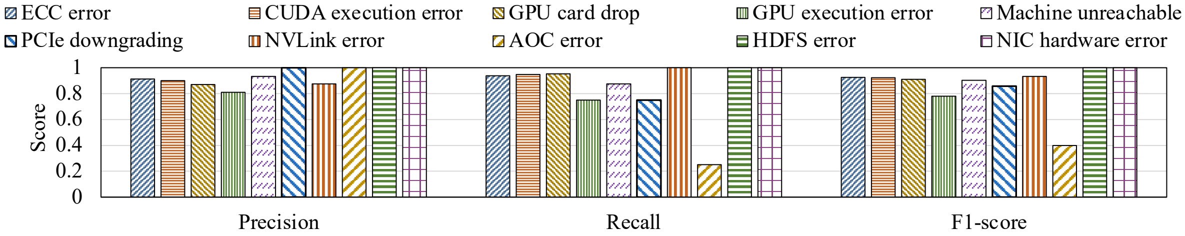

Figure 10: Accuracy for various fault types.

Data APIs) and processing time (preprocessing, and running inference for faulty machine detection). Due to Minder's deployment on an exclusive machine, it reduces the time by 99% (shorter time by 500 × compared with Figure 2) if engineers otherwise manually inspect machine information one by one.

Comparison with the baseline. The baseline for comparison is Mahalanobis Distance (MD) [30, 46, 57]. MD is widely used in identifying outliers. It considers the variable correlations in multi-dimensional data and calculates features like mean, variance, skewness, and kurtosis before applying principle component analysis (PCA) and computing the pairwise distances. We keep other processes the same for comparison.

In Figure 9, the precisions are 0.904&0.788, with recalls of 0.883&0.767, leading to F-1 scores of 0.893&0.777 for Minder and MD. These findings demonstrate that Minder effectively detects actual faults and lowers the rate of false alarms. MD detects the outlier machine from the statistical perspective but exhibits lower accuracy, indicating that jitters and noises interfere with statistical features. Minder outperforms MD by using LSTM-VAE for denoising and extracting the data patterns for a better distance calculation.

Performance breakdown with fault types. By considering individual fault types in Table 1, the results slightly differ. In Figure 10, Minder effectively handles faults like ECC error, CUDA execution error, GPU card drop, machine unreachable, NVLink error, HDFS error, and NIC hardware error. These faults are relative to CPU, GPU states, and networking performances that Minder monitors.

GPU execution error and PCIe downgrading present a lower recall, since some faulty instances have concurrent faulty GPUs or PCIe links within a machine. Owing to the 3D parallel topology, faults swiftly impact multiple machines in both the DP and PP groups, leading to an instant group effect. Thus, the time-series granularity at the second level is insufficient for Minder. Other fault types, such as AOC errors, are partially missed due to the current absence of optical cable-related counters for capturing useful monitoring metrics. We will continue to have more hardware counters in the future. Overall, AOC errors occupy only a small portion, so the overall accuracy is still high.

Upon examining the errors made by Minder, most of them were not entirely incorrect. In many error cases, the machine incorrectly detected showed short-term metric fluctuations or performance jitters before being normalized afterward. These detected fluctuations, however, were not the root cause of the task halt. Nonetheless, these errors should not be ignored as they led to short-term jitters in the training performance.

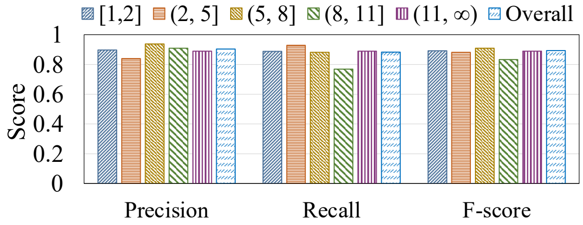

Figure 11: Accuracy for tasks with varied lifecycle fault occurrences.

Performance breakdown by fault occurrences of a task's lifetime. Training tasks with various workloads experience distinct occurrences of faults. Figure 11 categorizes the performance relative to the fault numbers throughout a task's lifetime, which may span up to months for extensive workloads. In our dataset, 70% of the tasks display no more than five faults, whereas over 15% face more than eight faults throughout their lifetime. Given the lack of interdependence among individual faults, the accuracy is not tied to the fault occurrences. Since faults are random and a faulty machine will be promptly auto-replaced in the production environment, the fault occurrences do not affect performance. Despite the lower recall for the (8, 11] group, it primarily originates from the limited task number of this group.

6.2 Analysis of Monitoring Metric Selection

To validate the proper selection of monitoring metrics in (4.3), we conduct experiments to show that fewer or more metrics do not improve accuracy. Notably, most of the remaining metrics not used by Minder are GPU-related, including `GPU Temperature`, `GPU Clocks`, `GPU Memory Bandwidth Usage`, and `GPU FP Engine Activity`. Thus, we use only `GPU Duty Cycle` to train a GPU model with fewer metrics, while adding these unused GPU-related monitoring metrics to train a GPU model with more metrics. We keep the other settings unchanged for the comparison.

Figure 12 reveals that including more monitoring metrics achieves a higher recall, but its precision is lower. More metrics may introduce mutual interference, since different metrics may indicate different patterns that will obfuscate the detection. On the other hand, using fewer metrics undermines outlier detection capacity due to the exclusion of key metrics.

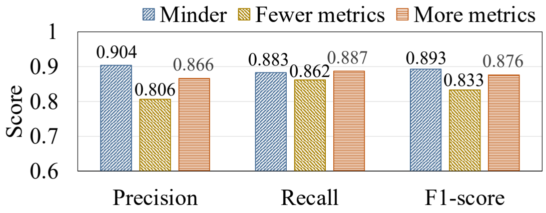

Figure 12: Comparison with different metric selections.

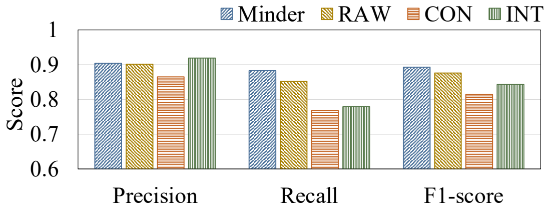

Figure 13: Comparison with different model selections.

Our optimal selection of metrics achieves the best precision, meaning that Minder alerts far fewer false alarms than others. Based on the metric priority result (4.3), the top metrics are sufficient to cover all components that might malfunction.

6.3 Analysis of Model Selection

We contrast Minder against other statistical methods or model variations to show the proper choice of LSTM-VAE. A simple approach is calculating the Euclidean Distances of the preprocessed raw data (RAW) without using VAE. Variants of LSTM-VAE include concatenating the embeddings of all the models as a whole for distance calculation (CON) or training an integrated LSTM-VAE model with all the monitoring metrics (INT).

In Figure 13, Minder outperforms others in recall and F1-score. Their recalls vary. Worse recall generated by RAW illustrates the significance of eliminating noises. Minder, in contrast, reconstructs for denoising. CON and INT show worse recall because they consider multiple metrics with equal significance by calculating distances from evenly concatenated embeddings or regarding all the metrics as a whole for input. However, not all monitoring metrics have an equal sensitivity to faults. Mutual interference exacerbates the performance of CON and INT. Minder, on the other hand, leverages the VAE for denoising and separates metrics to enhance the performance. Besides, comparing the input and reconstructed data of LSTM-VAE yields a Mean Squared Error (MSE) lower than 0.0001, demonstrating effective reconstruction.

6.4 Analysis of Continuity and Threshold

To verify the feasibility of continuous detection, we compare Minder without the application of continuity (3.2). Minder ensures the same machine is detected multiple times. Without continuity, an alert will be made immediately upon a fault detection during the time window. The results in Figure 14 imply that the overall performance is worse without continuity.

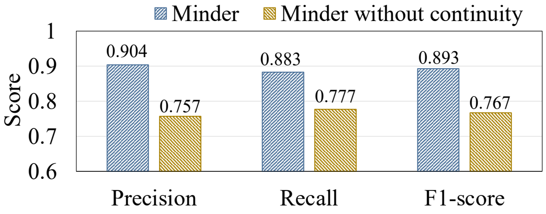

Figure 14: Accuracy with/without continuity.

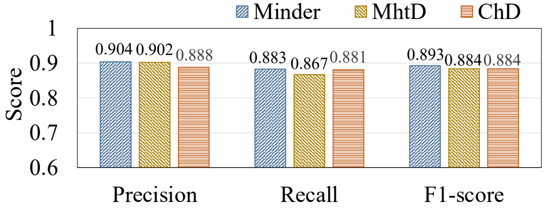

Figure 15: Comparison with different distance measures.

More false alarms are triggered due to occasional short-term jitters. The continuity emphasizes the degraded performance for a period caused by a fault's gradual impact on other machines. Thus, Minder filters discrete bugs, noises, or jitters.

Note that we set the *continuity threshold* (4.4) as four minutes to reduce false alarms caused by jitters or noises. That means Minder only alerts when a detected machine endures dissimilarity for four minutes. The threshold is chosen empirically based on the fault duration in Figure 4. Most of the faults last longer than four minutes before the halt. A shorter continuity threshold introduces more false alarms, while a longer one excludes more actual faulty detection results.

6.5 Choice of Distance Measures

We compare Minder 's pair-wise Euclidean Distance algorithm with Manhattan Distance (MhtD) and Chebyshev Distance (ChD). MhtD adds up the absolute distances from each dimension of the embeddings while ChD uses the largest difference in their coordinates. We replace the distance algorithm to rerun the experiments.

In Figure 15, Minder achieves similar performance to others, suggesting that the embeddings from LSTM-VAE are already representative and the outlier is distinctive for any distance calculation method. The comparison with MhtD implies that spatial distribution is a solid representation because both methods use the distances of multiple dimensions. ChD's worse precision suggests that a single spatial difference is insufficient for comparing dissimilarity. Minder considers the overall distance from the outlier to other normal ones and the dissimilarity will be intensified by summing the distances.

6.6 Performance with Multiple Concurrent Faulty Machines

Analyzing Minder's ability to handle multiple faults is intuitive. Such detection capability largely depends on the faulty machine scale ratio and the granularity of monitoring data.

Given the 3D parallelism [40], a machine participates in multiple DP and PP groups. More faulty machines impact more groups, inducing faster negative propagation across the entire cluster. The data granularity determines the visibility of the propagation process. With a brief duration of less than 100 milliseconds per interaction, the dissimilar pattern may be overlooked due to coarse-grained monitoring. Yet, millisecond-level monitoring is not widely deployed due to the high overhead.

In our production environment, concurrent faulty machine instances only occur due to automatic switch reboots or switch-related AOC errors. A switch reboots in response to high temperatures, extreme port congestion, and so on. Thirty-two connected machines will be forced to go offline out of a total of 600 machines in our environment. However, Minder hardly distinguishes the faulty outliers. Firstly, the fault ratio is relatively high. Given our rail-optimized topology and 3D parallelism mechanism, communication among 32 machines contains at most 256 DP groups, quickly propagating across other machines. Moreover, the current second-level time granularity monitoring limits Minder's ability. The rapid spread is only observable at the millisecond level following several training interactions. However, the rate of such multiple faulty machine instances is less than 1%, sometimes zero a month. Should a switch reboot, the switch monitoring system will automatically alert the engineers.

To demonstrate Minder's ability to detect multiple faulty machines, we injected PCIe downgrading into two of four machines simultaneously, with customized millisecond-level monitoring. Thus, the ratio of faulty machines is higher and the data granularity is finer than the switch-reboot instance. In the experiment, each machine was equipped with eight NVIDIA Ampere GPUs, running Reduce-Scatter collectively. Two PCIe links on two machines were purposely degraded. With the millisecond-level data from the NICs, Minder could detect the two NICs connected to the faulty PCIe links. These two NICs presented the largest outlier distances during Reduce-Scatter. Figure 16 shows the millisecond-level monitoring, where normal NICs demonstrate a high throughput at the beginning of each Reduce-Scatter step to transmit their data to the next node. In contrast, the NICs with downgraded PCIe links exhibit steady and low throughput. Thereafter, normal ones drop to zero, after transmitting their data, waiting for the slow NICs for synchronization. Minder can capture this dissimilarity, thanks to the granularity of milliseconds. Our injection experiments also show Minder's ability to detect other concurrent faults, such as GPU degradation and NIC throughput downgrading.

7 Discussion

In this section, we discuss the lessons we learned during implementation and potential future works.

Minder and other monitoring tools. Large-scale model training involves the cooperative efforts of multiple teams.

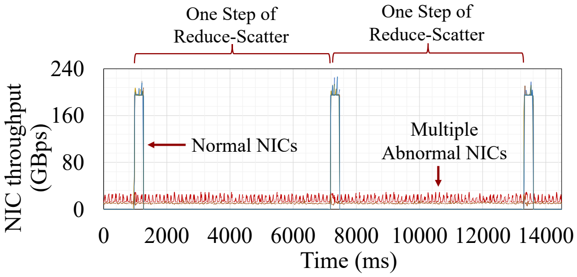

Figure 16: Millisecond-level NIC throughput for all machines after injection of PCIe downgrading on two NICs.

Other monitoring tools used along with Minder include switch state monitoring, periodic heartbeat messages (IP, hardware states, Pod names *etc.*), RDMA traffic down-limit alerts, R-Pingmesh [54] (a pingmesh-like [33] connection testing), and automatic text analysis for GPU error detection. As described in 5, a detected machine will be replaced before restarting the task. These approaches primarily target network connection and jitters or GPU states, whereas Minder provides metrics covering computation, storage, and communication resources. The combined use of them enhances the detection efficacy. Meanwhile, offline testing tools, such as DCGM [1], EUD [3], are used for intra-host bottlenecks diagnosis, though not feasible for run-time fault identification.

Similar systems and tools are introduced. For example, SuperBench [77] is a proactive system to ensure cloud AI infrastructure reliability. It runs model training and component benchmarks to identify incremental performance degradation on defective machines. The first difference is that it works proactively because incremental performance degradation exists due to hardware redundancy. However, such gradual degradation process before a fault occurs is hardly observed in our distributed training. Second, SuperBench mainly improves the hardware-side reliability. Real-time software errors also occur frequently during model training. Besides, SuperBench performs offline validation by running benchmarks. Minder otherwise monitors the tasks throughout their life cycles. Thus, the integration of such a proactive system with Minder is a more promising debugging solution. By executing benchmarks on the machines in a job and monitoring their running status, faults might be largely prevented and rapidly detected.

Minder's Generality. Minder can be extended in data granularity and the spectrum of available metrics. Second-level monitoring is currently deployed and used by Minder. Minder has demonstrated improved detection capability for concurrent faulty machines with millisecond-level monitoring data in 6.6. With finer-grained data, the annihilated rapid propagation from the straggler will be revealed, as a training iteration only lasts tens of milliseconds. Besides, the currently available metrics for Minder are out-of-band hardware counters. Other metrics (*e.g.,* AOC counters) and in-band traces (Torch Profiler [16], the Megatron-LM timer, or CUDA event timer [40]) could also be utilized by Minder. These new traces offer fine-grained training and collective communication operation information for comprehensive performance monitoring.

Minder's robustness of other faults. Notably, Minder is allowed to detect new or rare fault types not covered in training, as long as the monitoring data presents discernible dissimilarities. For concurrent faulty machines, provided multiple machines present a distinctive dissimilarity, Minder can detect them simultaneously.

Machine-level similarity. Large models such as those in LLM training and multi-modal tasks often employ 3D parallelism. This results in a trend of consistency across machines for computation, communication, and storage, allowing Minder to detect faulty machines. As a result, we focus on machine-level detection instead of finer granularity. We also consider Minder in other workloads, such as large-scale inference and fine-tuning. As long as these workloads satisfy the requirements of inter-machine metric similarity and fault continuity, Minder could be applied. Future work will explore Minder's effectiveness in other workloads.

Not all failed tasks have the right label. Although the labeled machine is the root cause, the Minder-detected machine may also have temporary performance fluctuations. It is necessary to inform engineers of such performance jitters.

Root cause analysis. Minder detects faults at the machine level. The root cause for a fault indicated by a metric is uncertain. Extra labor is employed for further network jitter and short-term straggler analysis. In the future, we plan to design fine-grained run-time monitoring for root cause identification.

8 Related Work

In this section, we briefly introduce related works on anomaly diagnosis [20, 28, 29, 31, 37, 38, 41, 59, 61, 72, 84].

Intra-host diagnosis. Run-time diagnosis detects anomalies without disturbing the running tasks. Leveraging the existing counters [8, 14] from the machine is a direct method [70, 71, 78]. For example, BRCA [63], MonitorRank [43], and FChain [62] construct a dependency graph based on historical monitoring data or traces for anomaly alerts. These DL algorithms have been proven to be automatic, robust, and flexible. Another naive but effective way is to check logs or `dmesg` [2] using natural language processing (NLP)-based methods [19]. However, log-based approaches are limited to their log content and information processing abilities.

Offline diagnosis requires specific tools to detect possible intra-host bottlenecks when the machine is not running tasks. Liu *et al.* [55] and Martinasso *et al.* [58] implement tools to test if there is NVLink [13] or PCIe link [15] degradation or congestion. Collie [44] is a "fuzzing"-like implementation to help uncover potential performance bottlenecks in RNICs. Deepview [82] is designed for virtual hard disk failure localization. These approaches are useful ahead of running tasks, so they cannot be directly used during the large-scale training process.

Inter-host diagnosis. NetBouncer [73] leverages the IP-in-IP technique to actively localize failure devices or links among millions of servers in a data center network. Similarly, SNAP [79] monitors TCP statistics and socket logs for network diagnosis. Pingmesh [33], Haecki [34], and Fathom [66] monitor end-to-end latencies between arbitrary servers or detailed RPC performance in data centers. Nonetheless, they only detect failures along the routes instead of intra-host hardware failures that degrade the training speed. Cloud system diagnosis [26, 35, 47, 81] are based on contextual data patterns and associations of microservices on multiple machines. Such patterns and dependencies are eliminated in distributed training, where machines exhibit similar workloads.

Algorithms for anomaly detection and diagnosis. The first type is statistics-based methods. Apart from Euclidean distance, Pearson Correlation [25], Kendall's tau [42], and Spearman Correlation [80] also quantify the similarity between two vectors and discover the deviating anomalies. Setting parameters as thresholds from experienced operators is usually required [56].

Supervised algorithms are widely used for anomaly detection [45, 53]. EGADS in Yahoo [45] leverages diverse machine learning methods for large-scale univariate time series anomaly detection. Machine learning algorithms like random forest [64] are used for incident routing [27], VM compromise detection [21]. However, supervised learning does not fit in our context, where the goal is to detect the faulty machine instead of classification.

Unsupervised learning [50, 51, 56, 70, 71, 75, 78] identifies outliers from monitoring data for root cause detection and machine state detection, or enables dialogue-based diagnosis chatting. Apart from VAE (4.2), clustering [50, 51, 56] is commonly used to cluster the machines with similar monitoring data change patterns or detect anomalies for time series.

9 Conclusion

This paper presents Minder, addressing the problem of faulty machine detection in distributed training tasks. Minder leverages the concept of similarity among machines and the continuity of a fault during training. Minder has been deployed in our production environment for a year to assist engineers in training diagnosis. Evaluation results demonstrate the reduced time required by Minder and the effectiveness of its design choices for training tasks.

This work does not raise any ethical issues.

Acknowledgments

We thank all the anonymous NSDI reviewers for their feedback that greatly improved the paper. We thank the broader ByteDance High-speed Network team and Applied Machine Learning team for their support throughout this project.

References

[1] DCGM Diagnostics. https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/dcgm-diagnostics.html.

[2] dmesg. https://linuxhint.com/dmesg_tutorial/.

[3] Extended Utility Diagnostics (EUD). https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/dcgm-eud.html.

[4] GPU Memory and Duty Cycle. https://cloud.google.com/kubernetes-engine/docs/concepts/gpus.

[5] GPU Metrics. https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/feature-overview.html#profiling-metrics.

[6] GPU Pricing. https://cloud.google.com/compute/gpus-pricing.

[7] GPU tensor core utility. https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/feature-overview.html#profiling-metrics.

[8] Intel Performance Counter Monitor. https://github.com/opcm/pcm.

[9] NVIDIA ConnectX-6 Dx Network Adapters. https://www.nvidia.com/en-us/networking/ethernet/connectx-6-dx/.

[10] Nvidia DGX-2H. https://docs.nvidia.com/dgx/pdf/dgx2-user-guide.pdf.

[11] Nvidia DGX-A100. https://www.nvidia.com/en-us/data-center/dgx-a100/.

[12] Nvidia NCCL timeout. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html.

[13] Nvidia NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/.

[14] Nvidia System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface.

[15] PCIe. https://docs.nvidia.com/certification-programs/pdf/nvidia-certified-configuration-guide.pdf.

[16] torch profiler. https://pytorch.org/docs/stable/profiler.html.

[17] *Infiniband Trade Association. Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A17: RoCEv2 (IP routable RoCE).* 2014.

[18] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[19] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370, 2006.

[20] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. Liu, J. Padhye, G. Outhred, and B. T. Loo. Closing the network diagnostics gap with vigil. In *Proceedings of the SIGCOMM Posters and Demos*, pages 40–42. 2017.

[21] B. Arzani, S. Ciraci, S. Saroiu, A. Wolman, J. Stokes, G. Outhred, and L. Diwu. {PrivateEye}: Scalable and {Privacy-Preserving} compromise detection in the cloud. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 797–815, 2020.

[22] H. Blockeel and L. De Raedt. Top-down induction of first-order logical decision trees. *Artificial intelligence*, 101(1-2):285–297, 1998.

[23] C. Cheadle, M. P. Vawter, W. J. Freed, and K. G. Becker. Analysis of microarray data using z score transformation. *The Journal of molecular diagnostics*, 5(2):73–81, 2003.

[24] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[25] I. Cohen, Y. Huang, J. Chen, J. Benesty, J. Benesty, J. Chen, Y. Huang, and I. Cohen. Pearson correlation coefficient. *Noise reduction in speech processing*, pages 1–4, 2009.

[26] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou. Sage: practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–151, 2021.

[27] J. Gao, N. Yaseen, R. MacDavid, F. V. Frujeri, V. Liu, R. Bianchini, R. Aditya, X. Wang, H. Lee, D. Maltz, et al. Scouts: Improving the diagnosis process through domain-customized incident routing. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 253–269, 2020.

[28] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. {SIMON}: A simple and scalable method for sensing, inference and measurement in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 549–564, 2019.

[29] D. Ghita, K. Argyraki, and P. Thiran. Toward accurate and practical network tomography. *ACM SIGOPS Operating Systems Review*, 47(1):22–26, 2013.

[30] H. Ghorbani. Mahalanobis distance and its application for detecting multivariate outliers. *Facta Universitatis, Series: Mathematics and Informatics*, pages 583–595, 2019.

[31] J. Gong, Y. Li, B. Anwer, A. Shaikh, and M. Yu. Microscope: Queue-based performance diagnosis for network functions. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 390–403, 2020.

[32] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi. A survey of methods for explaining black box models. *ACM computing surveys (CSUR)*, 51(5):1–42, 2018.

[33] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[34] R. Haecki, R. N. Mysore, L. Suresh, G. Zellweger, B. Gan, T. Merrifield, S. Banerjee, and T. Roscoe. How to diagnose nanosecond network latencies in rich end-host stacks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 861–877, 2022.

[35] V. Harsh, W. Zhou, S. Ashok, R. N. Mysore, B. Godfrey, and S. Banerjee. Murphy: Performance diagnosis of distributed cloud applications. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 438–451, 2023.

[36] T. He, X. Li, Z. Wang, K. Qian, J. Xu, W. Yu, and J. Zhou. Unicron: Economizing self-healing llm training at scale. *arXiv preprint arXiv:2401.00134*, 2023.

[37] B. Heller, C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley, et al. Leveraging sdn layering to systematically troubleshoot networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 37–42, 2013.

[38] H. Herodotou, B. Ding, S. Balakrishnan, G. Outhred, and P. Fitter. Scalable near real-time failure localization of data center networks. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1689–1698, 2014.

[39] Q. Hu, Z. Ye, Z. Wang, G. Wang, M. Zhang, Q. Chen, P. Sun, D. Lin, X. Wang, Y. Luo, Y. Wen, and T. Zhang. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 709–729, Santa Clara, CA, Apr. 2024. USENIX Association.

[40] Z. Jiang, H. Lin, Y. Zhong, Q. Huang, Y. Chen, Z. Zhang, Y. Peng, X. Li, C. Xie, S. Nong, et al. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[41] E. Katz-Bassett, H. V. Madhyastha, V. K. Adhikari, C. Scott, J. Sherry, P. Van Wesep, T. E. Anderson, and A. Krishnamurthy. Reverse traceroute. In *NSDI*, volume 10, pages 219–234, 2010.

[42] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[43] M. Kim, R. Sumbaly, and S. Shah. Root cause detection in a service-oriented architecture. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):93–104, 2013.

[44] X. Kong, Y. Zhu, H. Zhou, Z. Jiang, J. Ye, C. Guo, and D. Zhuo. Collie: Finding performance anomalies in {RDMA} subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 287–305, 2022.

[45] N. Laptev, S. Amizadeh, and I. Flint. Generic and scalable framework for automated time-series anomaly detection. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1939–1947, 2015.

[46] C. Leys, O. Klein, Y. Dominicy, and C. Ley. Detecting multivariate outliers: Use a robust variant of the mahalanobis distance. *Journal of experimental social psychology*, 74:150–156, 2018.

[47] L. Li, X. Zhang, S. He, Y. Kang, H. Zhang, M. Ma, Y. Dang, Z. Xu, S. Rajmohan, Q. Lin, et al. Conan: Diagnosing batch failures for cloud systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 138–149. IEEE, 2023.

[48] M. Li, D. G. Andersen, A. J. Smola, and K. Yu. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems*, 27, 2014.

[49] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[50] X. Li, J. Lin, and L. Zhao. Linear time complexity time series clustering with symbolic pattern forest. In *IJCAI*, pages 2930–2936, 2019.

[51] Z. Li, Y. Zhao, R. Liu, and D. Pei. Robust and rapid clustering of kpis for large-scale anomaly detection. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2018.

[52] S. Lin, R. Clark, R. Birke, S. Schönborn, N. Trigoni, and S. Roberts. Anomaly detection for time series using vae-lstm hybrid model. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4322–4326. Ieee, 2020.

[53] D. Liu, Y. Zhao, H. Xu, Y. Sun, D. Pei, J. Luo, X. Jing, and M. Feng. Opprentice: Towards practical and automatic anomaly detection through machine learning. In *Proceedings of the 2015 internet measurement conference*, pages 211–224, 2015.

[54] K. Liu, Z. Jiang, J. Zhang, S. Guo, X. Zhang, Y. Bai, Y. Dong, F. Luo, Z. Zhang, L. Wang, X. Shi, H. Xu, Y. Bai, D. Song, H. Wei, B. Li, Y. Pan, T. Pan, and T. Huang. R-pingmesh: A service-aware roce network monitoring and diagnostic system. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 554–567. Association for Computing Machinery, 2024.

[55] K. Liu, Z. Jiang, J. Zhang, H. Wei, X. Zhong, L. Tan, T. Pan, and T. Huang. Hostping: Diagnosing intra-host network bottlenecks in {RDMA} servers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 15–29, 2023.

[56] P. Liu, Y. Chen, X. Nie, J. Zhu, S. Zhang, K. Sui, M. Zhang, and D. Pei. Fluxrank: A widely-deployable framework to automatically localizing root cause machines for software service failure mitigation. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 35–46. IEEE, 2019.

[57] P. C. Mahalanobis. On the generalized distance in statistics. *Sankhyā: The Indian Journal of Statistics, Series A (2008-)*, 80:S1–S7, 2018.

[58] M. Martinasso, G. Kwasniewski, S. R. Alam, T. C. Schulthess, and T. Hoefler. A pcie congestion-aware performance model for densely populated accelerator servers. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 739–749. IEEE, 2016.

[59] D. S. Matteson and N. A. James. A nonparametric approach for multiple change point analysis of multivariate data. *Journal of the American Statistical Association*, 109(505):334–345, 2014.

[60] Z. Meng, M. Wang, J. Bai, M. Xu, H. Mao, and H. Hu. Interpreting deep learning-based networking systems. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 154–171, 2020.

[61] R. N. Mysore, R. Mahajan, A. Vahdat, and G. Varghese. Gestalt: Fast,{Unified} fault localization for networked systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 255–267, 2014.

[62] H. Nguyen, Z. Shen, Y. Tan, and X. Gu. Fchain: Toward black-box online fault localization for cloud systems. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 21–30. IEEE, 2013.

[63] X. Nie, Y. Zhao, K. Sui, D. Pei, Y. Chen, and X. Qu. Mining causality graph for automatic web-based service diagnosis. In *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2016.

[64] A. Palczewska, J. Palczewski, R. M. Robinson, and D. Neagu. Interpreting random forest models using a feature contribution method. In *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*, pages 112–119. IEEE, 2013.

[65] P. Patarasuk and X. Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[66] M. A. Qureshi, J. Yan, Y. Cheng, S. H. Yeganeh, Y. Seung, N. Cardwell, W. De Bruijn, V. Jacobson, J. Kaur, D. Wetherall, et al. Fathom: Understanding datacenter application network performance. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 394–405, 2023.

[67] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[68] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[69] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

[70] Y. Su, Y. Zhao, C. Niu, R. Liu, W. Sun, and D. Pei. Robust anomaly detection for multivariate time series through stochastic recurrent neural network. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2828–2837, 2019.

[71] M. Sun, Y. Su, S. Zhang, Y. Cao, Y. Liu, D. Pei, W. Wu, Y. Zhang, X. Liu, and J. Tang. Ctf: Anomaly detection in high-dimensional time series with coarse-to-fine model transfer. In *IEEE INFOCOM 2021-IEEE conference on computer communications*, pages 1–10. IEEE, 2021.

[72] P. Tammana, R. Agarwal, and M. Lee. Simplifying datacenter network debugging with {PathDump}. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 233–248, 2016.

[73] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang. {NetBouncer}: Active device and link failure localization in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 599–614, 2019.

[74] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[75] H. Wang, A. Abhashkumar, C. Lin, T. Zhang, X. Gu, N. Ma, C. Wu, S. Liu, W. Zhou, Y. Dong, W. Jiang, and Y. Wang. NetAssistant: Dialogue based network diagnosis in data center networks. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 2011–2024, Santa Clara, CA, Apr. 2024. USENIX Association.

[76] D. J. Weller-Fahy, B. J. Borghetti, and A. A. Sodemann. A survey of distance and similarity measures used within network intrusion anomaly detection. *IEEE Communications Surveys & Tutorials*, 17(1):70–91, 2014.

[77] Y. Xiong, Y. Jiang, Z. Yang, L. Qu, G. Zhao, S. Liu, D. Zhong, B. Pinzur, J. Zhang, Y. Wang, J. Jose, H. Pourreza, J. Baxter, K. Datta, P. Ram, L. Melton, J. Chau, P. Cheng, Y. Xiong, and L. Zhou. SuperBench: Improving cloud AI infrastructure reliability with proactive validation. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 835–850, Santa Clara, CA, 2024. USENIX Association.

[78] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng, et al. Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications. In *Proceedings of the 2018 world wide web conference*, pages 187–196, 2018.

[79] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[80] J. H. Zar. Spearman rank correlation. *Encyclopedia of biostatistics*, 7, 2005.

[81] Z. Zeng, Y. Zhang, Y. Xu, M. Ma, B. Qiao, W. Zou, Q. Chen, M. Zhang, X. Zhang, H. Zhang, et al. Traceark: Towards actionable performance anomaly alerting for online service systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 258–269. IEEE, 2023.

[82] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, M. Chintalapati, A. Krishnamurthy, and T. Anderson. Deepview: Virtual disk failure diagnosis and pattern detection for azure. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 519–532, 2018.

[83] P. Zhao, M. Kurihara, J. Tanaka, T. Noda, S. Chikuma, and T. Suzuki. Advanced correlation-based anomaly detection method for predictive maintenance. In *2017 IEEE International Conference on Prognostics and Health Management (ICPHM)*, pages 78–83. IEEE, 2017.

[84] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015.

Table 2: Monitoring metrics collected by Minder.

Monitoring Metrics	Description
CPU Usage	Percentage of CPU time being used.
PFC Tx Packet Rate	Periodic counts of PFC packets sent by RDMA-enabled devices.
Memory Usage	Percentage of memory being used.
Disk Usage	Percentage of storage space being used on a disk.
TCP Throughput	Periodic counts of the amount of TCP data being transmitted by a NIC.
TCP+RDMA Throughput	Periodic counts of the amount of TCP and RDMA data being transmitted by an NIC.
GPU Memory Used [4]	The amount of GPU memory being used by processes.
GPU Duty Cycle [4]	Percentage of time over the past sample period when the accelerator is active.
GPU Power Draw	Periodic counts of the GPU power consumption.
GPU Temperature	The temperature of a GPU while it is operating, measured in degrees Celsius.
GPU SM Activity [5]	Averaged percentage of time when at least one warp is active on a multiprocessor.
GPU Clocks	The clock speed of a GPU, reflecting the frequency of the GPU's processor.
GPU Tensor Core Activity [5]	Percentage of cycles when the tensor (HMMA / IMMA) pipe is active.
GPU Graphics Engine Activity [5]	Percentage of time when any portion of the graphics or compute engines are active.
GPU FP Engine Activity [5]	Percentage of cycles when the FP pipe is active.
GPU Memory Bandwidth Utilization [5]	Percentage of cycles when data is sent to or received from the device memory.
PCIe Bandwidth [5]	The rate of data transmitted/received over the PCIe bus.
PCIe Usage [5]	Percentage of the bandwidth being used on the PCIe bus.
GPU NVLink Bandwidth [5]	The rate of data transmitted/received over an NVLink.
ECN Packet Rate	Periodic counts of ECN packets transmitted/received by a NIC.
CNP Packet Rate	Periodic counts of CNP packets transmitted/received by a NIC.

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

A Fault Types

The fault types are listed in Table 1.ECC error: caused by corrupted or lost data in (GPU) memory. PCIe downgrading: a link fault leading to a slow PCIe sending/receiving rate. NIC dropout: a NIC is missing from the OS. GPU Card drop: a disconnected GPU card. NVLink error: a link fault between two Nvidia GPUs. AOC error: an error in high-speed active optical cables (AOC) on either the host network card or the switch side. CUDA execution error: an unexpected overflow or configuration leading to a failed CUDA program. GPU execution error: unexpected page-fault, out-of-memory, and other incorrect processing leading to GPU hang or other results. HDFS error: HDFS connection timeout, io error, and so on when loading or saving checkpoints. Machine unreachable: mostly due to malfunctioning SSH services or virtual machine services. Others: illegal memory access, failed scheduling, no disk storage, low resource usage, switch reboot, and so on.

B Collected Monitoring Metrics

Table 2 contains the monitoring metrics that we choose to collect in our production environment, though only a portion of them are used for training and detection. Other available host metrics could also be used by Minder.