SuperBench: Improving Cloud AI Infrastructure Reliability with Proactive Validation

Yifan Xiong, Yuting Jiang, Ziyue Yang, and Lei Qu, *Microsoft Research;*
Guoshuai Zhao, Shuguang Liu, Dong Zhong, Boris Pinzur, Jie Zhang, Yang Wang,
Jithin Jose, Hossein Pourreza, Jeff Baxter, Kushal Datta, Prabhat Ram,
Luke Melton, and Joe Chau, *Microsoft;* Peng Cheng, Yongqiang Xiong,
and Lidong Zhou, *Microsoft Research*

https://www.usenix.org/conference/atc24/presentation/xiong

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

SuperBench: Improving Cloud AI Infrastructure Reliability with Proactive Validation

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[†]

[†]Microsoft Research   [‡]Microsoft

Abstract

Reliability in cloud AI infrastructure is crucial for cloud service providers, prompting the widespread use of hardware redundancies. However, these redundancies can inadvertently lead to hidden degradation, so called "gray failure", for AI workloads, significantly affecting end-to-end performance and concealing performance issues, which complicates root cause analysis for failures and regressions.

We introduce SuperBench, a proactive validation system for AI infrastructure that mitigates hidden degradation caused by hardware redundancies and enhances overall reliability. SuperBench features a comprehensive benchmark suite, capable of evaluating individual hardware components and representing most real AI workloads. It comprises a Validator which learns benchmark criteria to clearly pinpoint defective components. Additionally, SuperBench incorporates a Selector to balance validation time and issue-related penalties, enabling optimal timing for validation execution with a tailored subset of benchmarks. Through testbed evaluation and simulation, we demonstrate that SuperBench can increase the mean time between incidents by up to 22.61×. SuperBench has been successfully deployed in Azure production, validating hundreds of thousands of GPUs over the last two years.

1 Introduction

In the past decade, the surging demand for deep learning [44, 50–52] has spurred the development of exa-scale cloud AI infrastructure, which entails significantly high costs [33, 39, 65]. When incidents such as customer workload failures or performance regressions arise within the infrastructure, they can propagate throughout all nodes (i.e., servers) due to AI workloads' gang-scheduled and synchronization characteristics, leading to magnified penalties. For example, during a two-month distributed training for the OPT model, Meta reported over 105 training restarts resulting from failures on more than 100 VMs in the cloud [67], indicating 1.25 incidents per day

and 61k GPU hours affected [66]. To minimize such costly and disruptive incidents, maintaining system reliability is of paramount importance for cloud service providers.

Hardware redundancies are designed into cloud AI infrastructure for reliability purposes, including redundant GPU compute units [8, 37], GPU memory row remapping [48], over-provisioned (under-subscribed) networking links [32], etc. However, component-wise redundancies can surprisingly introduce hidden degradation, so called "gray failure" [30] or "fail-slow" [25] in traditional cloud, and make incident patterns even more complex in AI era due to following unexpected reasons. (i) AI workloads usually run long for weeks or months. The continuous and repetitive use of redundant components will cause them become problematic gradually with lower performance. For example, in Azure A100 cluster, each InfiniBand top-of-rack (ToR) switch has multiple redundant up links. When part of the redundant links are broken, certain traffic patterns such as all-to-all collective communication can experience throughput regression due to congestion. Such redundancy introduces *gradual performance degradation* rather than a binary either good or bad state for hardware components. (ii) This gradual degradation may not affect certain AI workloads in the early stage. For instance, ResNet model training can barely utilize all A100 GPU or networking resources, therefore less sensitive to trigger issues. Since it is difficult for cloud providers to unveil such partial redundancy loss timely with monitoring on existing workloads only, *standalone tests are required* to stress the hardware to pinpoint issues. (iii) Redundancies can also hide performance issues to some extent. When broken redundancies cause AI workload regression, repairing only partial redundancies instead of all can usually resolve the current incident and recover the workload. For example, when multiple redundant links are broken, operators may only need to replace one link to unblock the workload in a limited time. However, this property *decreases reliability* and increases the likelihood of future incidents when any single link goes bad.

As a result, although cloud providers apply hardware pre-qualification tests and timely troubleshooting for incidents,

---

[*]Equal contribution.

these methods are not aware of redundancies and cannot improve reliability. First, qualification tests are performed by hardware vendors before delivery but it cannot address reliability issues that arise in cloud environment and services. Second, troubleshooting is highly workload-dependent and can be time-consuming. It typically only repairs the partial redundancies for recovery, rather than restoring the full redundancies of the affected and non-dominant hardware components. As a result, the overall reliability remains low. For example, in Azure's A100 clusters, the *mean time between incidents* (MTBI) is 17.5 hours, and 38.1% of those incidents take more than 1-day to recover. What's worse, troubleshooting is only performed passively when incidents are raised by customers, it cannot restore redundancies timely before affecting workloads. However, such beforehand restoring is possible because gradual performance degradation can be detected by specific tests before incidents.

Considering the aforementioned problems, our key insight is that component-wise redundancies can surprisingly compromise the reliability of cloud AI infrastructure in unexpected ways. These redundancies can lead to incremental performance regressions, which should be identified and rectified before affecting end-to-end workloads. Rather than relying on reactive troubleshooting, which focuses on visibly defective components, we adopt a proactive approach to validate cloud AI infrastructure components before incidents transpire. This strategy aims to improve reliability at reasonable costs by balancing the trade-off between validation expenses and anticipated incident penalties.

To effectively minimize the MTBI, proactive validation should satisfy the following requirements: (i) *Comprehensive*: To detect incidents that are undetected by vendors in new clusters and only surface in customer workloads, validation must be comprehensive and encompass a wide range of AI workloads. (ii) *Clear-cut*: Given that hardware components can exhibit gradual performance degradation and measurements are prone to natural variance, it is essential to establish a clear-cut boundary between defective and normal performance. Repetitions of the same test should also yield consistent results, rather than fluctuating between outcomes. (iii) *Cost-efficient*: Proactive validation necessitates additional measurements, which consume time. Therefore, it must be cost-efficient, ensuring that validation expenses remain significantly lower than the incident-associated penalties.

Nevertheless, addressing these requirements presents significant challenges. Firstly, the sheer number of workloads and exponential node combinations result in an immense search space for all scenarios, making it impossible to encompass every aspect in the validation process. Secondly, after performance being measured, there is no ground truth available for defective components. Identifying which components are defective is problematic, as hardware specifications cannot reliably predict workload performance. Moreover, AI hardware often exhibits substantial variations [61], further

complicating the differentiation process. Lastly, the validation time and MTBI can be interdependent, since fewer validated components lead to shorter times between incidents. Determining when to validate which components for optimal cost-efficiency, while achieving the longest MTBI with least measurement time, proves to be a challenging endeavor.

To address these challenges, we introduce a proactive validation system, SuperBench, featuring three key modules: (i) First, we analyze workload distributions in large-scale, multi-tenant AI clusters to identify a small number of representative workloads and parameters applicable to most customers. Based on this, we develop a comprehensive benchmark set, including end-to-end benchmarks for representative workload patterns and micro-benchmarks for individual hardware components. (ii) A *Validator* is designed to conduct a series of benchmarks on specified nodes. To clearly distinguish between the benchmark results of defective and functional components, the space of cumulative distribution, rather than average metrics, is used for similarity clustering to learn benchmark criteria offline. (iii) To balance the trade-off between validation time and incident coverage, benchmarks are selected based on real-time estimations of incident probability. A *Selector* is designed to predicts node incident through a probability model and determine when to validate on which subset of benchmarks. The selected benchmarks are executed by the Validator, generating new benchmark and defect data to periodically update criteria and probability model, allowing the system to evolve in tandem with the latest node statuses.

SuperBench has been successfully deployed in one of the largest real-world AI infrastructure, Azure, for more than 2 years, where it identifies 10.36% nodes as defects, contributing to enhanced reliability. All benchmarks in SuperBench have been open-sourced on GitHub[1] and widely used by AI hardware vendors [1, 2] as a standard. Simulation results demonstrate that proactive validation in SuperBench can increase MTBI by $22.61\times$ and $1.11\times$ compared to the absence of validation and full set validation without benchmark selection, while increasing user GPU hours by $4.81\times$ and reducing validation time cost by 92.07%, respectively.

Our contributions can be summarized as follows:

- We identify the existence of gray failure in the new era of AI, which arises from AI hardware redundancies, and propose a proactive validation approach to mitigate its significant impact in cloud AI infrastructure.

- We observe and analyze a vast number of workload incidents, including failures and regressions, within a production cloud AI infrastructure, summarizing their sources and root causes.

- We establish the design goals for a proactive validation system and implement SuperBench to address these aims.

- We conduct extensive evaluations of SuperBench and de-
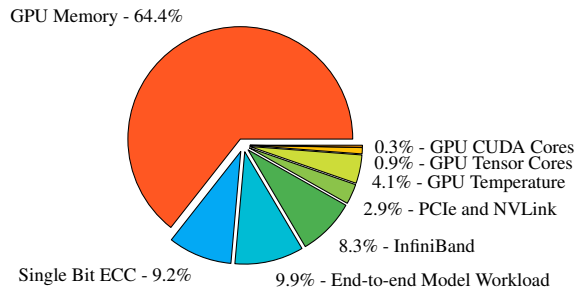
---

[1]https://github.com/microsoft/superbenchmark

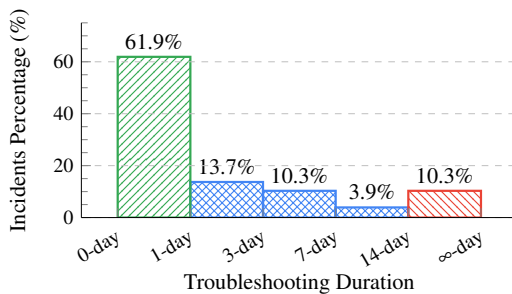Figure 1: Percentage of infrastructure incidents' sources.

Figure 2: Incidents troubleshooting duration distribution.

ploy the system in a production cloud environment, demonstrating its benefits and practicality in real-world.

2 Background and Motivation

2.1 Massive Incidents in AI Infrastructure

Incidents Summary In cloud AI infrastructure, we have observed that a wide range of components experience failure or performance regression. Figure 1 illustrates that more than 8 components in GPU node can occur performance issues according to 1-month tickets in Azure. However, even when only one component suffer issue, the real workloads will be reflected and incident ticket will be raised by customers.

When an incident occurs in cloud AI infrastructure, it typically takes a significant amount of time for troubleshooting in order to localize the defective nodes or links for mitigation. Meanwhile, all nodes involved in the incidents are waiting for troubleshooting, resulting in a considerable waste of GPU hours. Figure 2 shows that 38.1% of incidents require more than 1 day to resolve, and 10.3% incidents take more than 2 weeks. Thus it is crucial to identify defective nodes or links timely ahead of incidents for resources saving.

Upon analyzing the related incidents, we discovered that the main reasons for high incident rate stem from three aspects: rapid-evolving AI hardware, cloud environment, and immature software. We will explain more in the following paragraphs with real cases observed in Azure.

Rapid Hardware Evolution AI hardware defects may not be fully identified by vendors' qualification tests, as certain issues only surface in real workloads. Data center GPUs, in particular, are evolving rapidly, with new releases every one or two years [8, 37]. Vendors typically only examine individual hardware components at full utilization during qualification tests, but some regressions occur exclusively under specific workload patterns.

For example, we identified a regression that only occurs in certain A100 VMs when computation and communication are executed concurrently. Standalone computation benchmarks (e.g., GPU GEMM tests [45]) or communication benchmarks (e.g., NCCL AllReduce tests [47]) cannot expose this issue, as defective GPUs exhibit the same performance as non-defective ones. The problem arises only during simultaneous computation and communication due to the overlapping traffic pattern triggering interference in L2 cache within GPU memory.

Cloud Environment Cloud infrastructure environments can introduce additional incidents, as they differ from vendors' qualification environments in terms of power, temperature, and other factors. For example, data centers in tropical areas experience more incidents due to higher temperatures. We observed a $35\times$ increase in defective InfiniBand links with $> 10^{-12}$ bit error rate in data centers in tropical areas compared to data centers in higher latitudes, leading to significantly degraded performance for training and inference. Another example is GPU throttling. Even within the same data center, different racks or locations within the same rack can exhibit varying temperatures. However, all GPUs are designed with identical cooling and heatsinks by vendors, resulting in GPUs located in warmer locations potentially experiencing thermal throttling if they cannot receive more cool air.

Software Immaturity At the application level, AI software stacks frequently iterate to co-evolve with hardware and adapt to new architectures and features. For instance, CUDA [46] and ROCm [7] release new versions every one or two months, while new GPUs may only support newer software versions. Ensuring a mature and reliable software stack under such rapid evolution is difficult. Furthermore, most AI workloads are not resilient to software or hardware failures. Because AI training needs to synchronize tensors periodically with high frequency, any single failure or regression can quickly propagate to all involved nodes and cause workload incidents. For example, a single node issue can cause an entire distributed training job with 100+ nodes to hang, resulting in a costly health check across all nodes by the on-call person [66].

2.2 Observations on Hardware Redundancies

Over-provisioned Networking Links In Azure A100 clusters, there are over-provisioned InfiniBand networks with multiple redundant links between ToR switches and aggregation
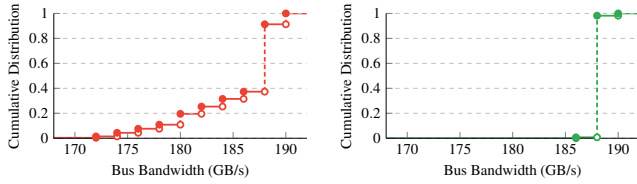
(a) Several ToR swithes have < (b) All ToR switches have $\geq$ 50%
50% of the redundant links up.   of the redundant links up.

Figure 3: Cumulative distribution of 2-node all-reduce bandwidth from a 24-node testbed with different redundancy ratios.

Table 1: Row remapping impact on end-to-end workloads.

correctable errors in row remapping	$1 \sim 10$	>10
row remapping node ratio of all nodes	3.19%	0.18%
regression node ratio of remapping nodes	5.6%	83.3%

switches. However, we find that to avoid performance regression due to congestion, more than half of the redundant links must be functional. In other words, at most half of the redundancies can be broken. Figure 3 depicts a real-world case of workload regression when running on 24 nodes with 192 InfiniBand NICs in a fat-tree InfiniBand testbed. Each ToR switch has 25% redundant uplinks. When some of the edge switches have over half of the redundant links down, multiple 2-node pairs experience a significant downgrade in all-reduce bus bandwidth performance when running traffic simultaneously. Only when operators repair the redundant links of all involved ToR switches to at least 50% does the all-reduce bandwidth for all 2-node pairs return to normal performance.

GPU Memory Row Remapping   A100 GPUs are equipped with redundant rows for every bank in HBM, introducing a hardware mechanism called row-remapping to replace known degraded memory cells with sparse ones in hardware and prevent the use of degraded memory [48]. This remapping is transparent to software, with no address space changes, and the degraded memory is replaced in hardware. Table 1 shows that when >10 correctable errors are remapped in redundant rows, there is a 77.8% higher chance of experiencing regression in end-to-end workloads compared to $1 \sim 10$ errors.

Infrastructure-wise Reliability   Component-wise redundancy can somewhat conceal end-to-end performance issues, but it would lead to more frequent incidents over time, diminishing the reliability of the entire infrastructure. As shown in Figure 4, we count the duration between customer-reported incidents for each node in one cluster, which has 20.7k jobs per month, then calculate the mean duration between i^{th} and $i+1^{th}$ incidents across all nodes that have $i+1$ incidents occurred. We find that the mean duration decreases from 719.4 hours to 151.7 hours compared between the 1st incident and the 20th incident. Furthermore, we infer the time to failure for jobs at different scales, supposing all nodes in the job have
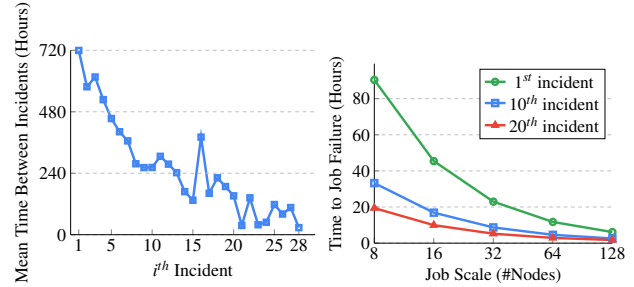

Figure 4: Left: mean time between i^{th} incidents for nodes. Right: time to failure for jobs if all nodes in the job have i^{th} incidents occurred.

the ith incident occurred and share the constant failure rate.

Why Troubleshooting Doesn't Work   It is hard for troubleshooting to identify the root cause of each incident and restore reliability. In a distributed AI system, there are many components and tiers, some of which may not provide appropriate error information when an incident occurs. For example, some incidents suffered "uncorrectable NVLink error detected during the execution" error reported by ML framework, but the root cause turned out to be timeout in InfiniBand network rather than broken NVLink. A process of elimination is required to investigate each component and tier before discovering the actual cause of failure or regression. Moreover, such workload incidents can also be closely tied to customers' applications and may not be directly related to infrastructure issues. Additionally, due to asynchronous execution in GPUs and adaptive routing in InfiniBand networks, incidents can be non-deterministic and hard to reproduce.

Even when troubleshooting successfully locates and resolves a current failure or regression, it cannot identify all potential issues. As AI infrastructure employs redundant resources to maintain reliability, when a failure occurs, it may indicate there are already multiple accumulated failures across various components behind the visible failure, leading to a decline in overall reliability capability. Unfortunately, troubleshooting alone cannot fully address these types of issues.

2.3   Challenges of Validation

Proactive validation is a practical method to tackle troubleshooting issue in AI infrastructure. Whether validation is passed or not clearly determines whether it is the responsibility of customer workloads or hardware from the perspective of cloud providers, thereby preventing any need for customer-specific troubleshooting, such as misconfiguration. However, there are 3 key challenges as follows:

Diverse Workloads   Numerous AI models run on cloud AI infrastructure, and these AI workloads are highly diverse. Figure 5 shows the percentage of different workloads after analyzed over 56k+ GPU jobs in several internal clusters. These workloads can be divided into three major categories,
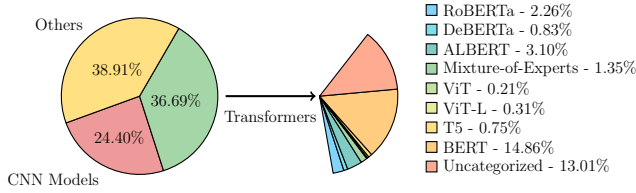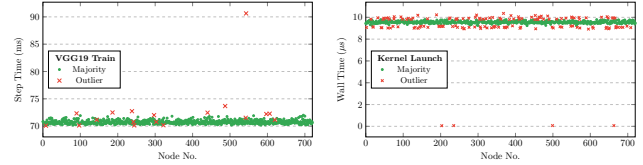
Figure 5: GPU job percentage for diverse workloads.

(a) LOF (*neighbors* = 10) on VGG19 training step time

(b) One-Class SVM (ν = 0.2) on GPU kernel launch wall time

Figure 6: Outlier detection methods for benchmark metrics.

including Transformers, CNN, and others. Within each category there exist tens of different models and many of them, e.g., 35.5% of all Transformers, are hard to be identified, indicating the diversity in AI workloads. We also analyzed the nodes with workload performance regression in this cluster in a 3-month period. Although there were 4.7% affected nodes in total, we found that 21.5% of them experienced a downgrade for only one workload during troubleshooting after running 6 common workloads on the same node. Therefore, performance regression can occur only for specific workloads, making it challenging for a validation system to ensure all customer workloads can be run with expected performance.

Distributed workloads run on multiple nodes in different scales. The node scales and orders exponentially expands the validation scope. For instance, all-reduce communication is commonly used during AI workloads and implemented using ring or tree algorithms. There are $n!$ ring permutations given the same set of n nodes with different orders, and different permutations utilize distinct link sets. This causes defective links only impact certain node scale and order. It's nearly impossible to cover all workloads with all node scales and orders in validation.

Lack of Ground-truth for Criteria   Determining performance criteria is challenging, as there is no ground truth to differentiate the performance of defective components from others. Workloads typically use performance metrics like end-to-end latency or throughput. Because different workload patterns have entirely different efficiencies, the end-to-end performance cannot be directly mapped to hardware specifications (e.g., GPU FLOPS, memory bandwidth, NVLink speed). Additionally, the software stack can significantly affect these efficiencies. Therefore, it is difficult to determine whether the measured performance is acceptable or not on a given node.

One possible approach is adopting outlier detection to filter defects as outliers among a large number of performance data points. However, such an approach is challenging to identify defects effectively since the distribution is diverse for different benchmarks. The boundary between outliers and normal ones is unclear, and careful hyper-parameter tuning is required for each benchmark on a case-by-case basis. As shown in Figure 6, the Local Outlier Factor (LOF) algorithm [14], which relies on estimating data density, may mark points in expected performance but with less density as outliers. The One-Class SVM [58], which relies on distances to set boundaries, may

also mark false positives when data is dense within an interval. Thus, defining the performance criteria of defective components clearly and reasonably remains challenging.

Furthermore, measurements always involve variation, including multiple runs on the same hardware or a single run on multiple hardware, which cannot be eliminated [42, 61]. The variation also depends on workload types. For example, MLPerf roughly divides its benchmarks into stable vision benchmarks and higher variance benchmarks, which have ±2.5% and ±5.0% variances [43], respectively. Due to this, it becomes more challenging to define clear-cut criteria so that the determination of defective components remains consistent across different runs with variations.

Duration and Coverage Trade-off   GPU hours in cloud AI infrastructure are expensive. For instance, renting 1,024 A100 GPUs on Azure costs $3.06M per month [12] while the same amount of TPU chips on GCP costs $2.41M per month [17]. Fewer and shorter tests can significantly save time and cost. However, the duration and coverage of validation can be interdependent, so less and shorter tests may reduce validation coverage and increase the risk of regression on customer workloads. Moreover, it can be challenging to maintain high reliability over time through cost-effective validation, as hardware quality tends to decrease gradually.

3   System Design

3.1   Overview

SuperBench is proposed to tackle above challenges with the following observations: (i) Despite the diversity of AI workloads, a majority can be represented by a few typical workloads, while component-wise tests can cover the rest. (ii) Instead of analyzing average metrics, all measured intermediate data (or its distribution) from a single benchmark run can be employed to establish a clear-cut boundary with maximum margins between defects or normal ones. (iii) The incident probability of a node increases gradually with use. Thus, a few benchmarks may cover most potential issues if that node passed a validation involving all benchmarks in the recent past.

Principles   The above observations guide the principles applied in the SuperBench system design:
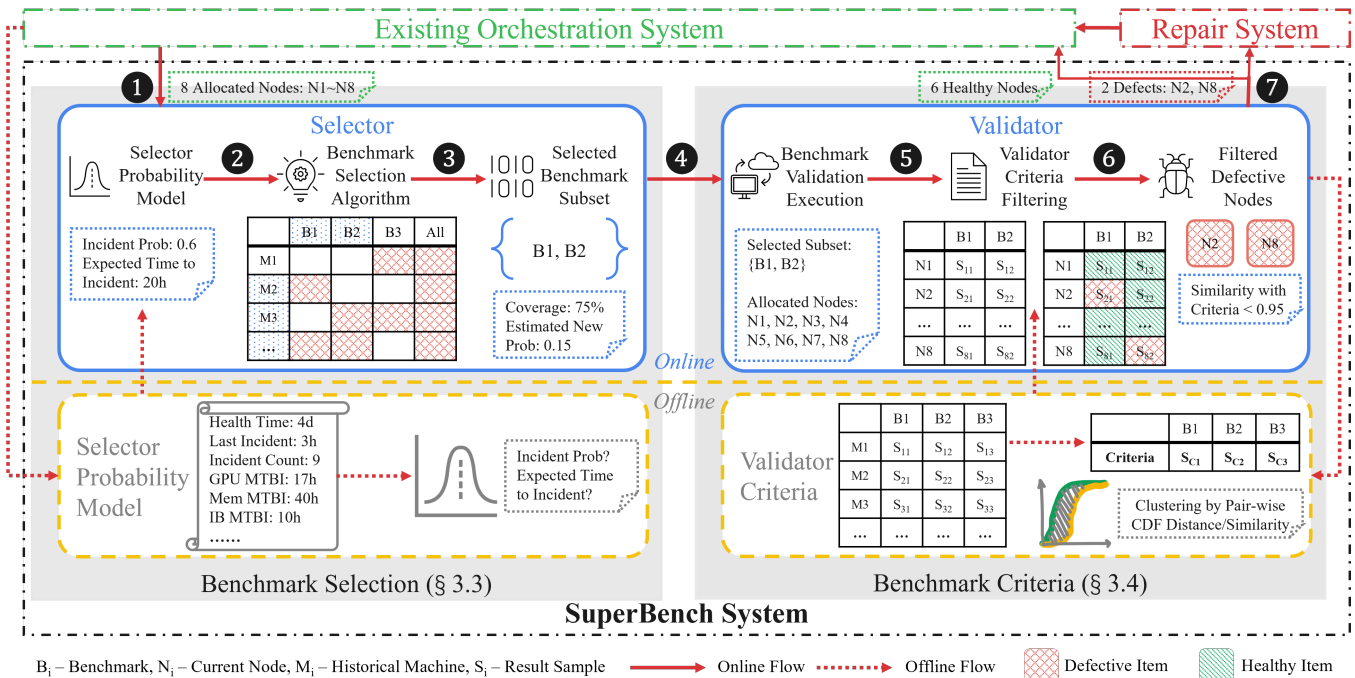
Figure 7: SuperBench system architecture and an example workflow.

- *Data-driven approaches*. Given the extensive data on defective nodes and benchmark results in cloud AI infrastructure, we utilize data-driven approaches to fit an incident probability model and formulate criteria statistically.

- *Quick but frequent validations*. Proactive validation should not be an one-time effort. Each node should be validated quickly but frequently in line with incident probability estimations to guarantee enduring reliability.

Architecture As depicted in Figure 7, SuperBench system comprises two primary components: a Selector and a Validator. The Selector determines when and which benchmark to execute based on incoming events. It offline builds a probability model to predict nodes incident probability based on their real-time statuses, and selects a benchmark subset based on a greedy algorithm for each validation event accordingly. The Validator conducts the benchmark execution and filters defective nodes, based on execution results and offline calculated benchmark criteria. The new node statuses and benchmark results will be continuously collected by SuperBench to periodically update the offline model and criteria.

Workflow The validation workflow is as follows. ❶ SuperBench carries out two types of validation: event-triggered and regular validation. The former is initiated by events from orchestration system, such as the node additions, node allocations for jobs. The latter is periodically checked by the Selector to validate existing nodes with high risks. ❷ For each validation, Selector will first obtain statuses of related nodes, like total up time, historical incident count, MTBI of

different incident types, etc., and predict incident probability with an offline trained model. ❸ If the incident probability is too high for the usage, Selector will select a subset of benchmarks considering the historical benchmark coverage and validation time trade-off. ❹ The subset decision will be passed to Validator. ❺ Validator will then execute the selected subset of benchmarks on the corresponding nodes and get the results for each benchmark. ❻ The results will be compared with Validator criteria, which is offline unsupervised learned from historical benchmark results. The node will be filtered as defective if any of its benchmark result has lower similarity with criteria than a threshold. ❼ SuperBench outputs the defective nodes for further handling. A repair system can be employed to maintain a defective buffer with nodes to be out for repair (OFR) and a hot buffer with repaired healthy nodes. It can swap new defective nodes with healthy ones from the hot buffer and pass to orchestration system in the runtime.

Integration The proposed SuperBench validation system can be integrated into existing clusters that employ an orchestration system [34, 63, 64], requiring the following interfaces supported in the control plane:

- Validation Events: It requires specific events from the orchestration system to initiate validation, including (1) when new nodes join the cluster or cluster-wide firmware/software is upgraded, (2) when customer jobs are allocated to specific nodes, and (3) when incidents are reported by customers and particular nodes are cordoned/drained.

- Validation Results: SuperBench identifies the defective nodes from validation node set. The orchestration system can choose to use the rest of healthy nodes or find replaced nodes from a dedicated hot buffer.

- Monitored Data: The Selector probability model requires status change data from all nodes when incidents occur.

3.2 Benchmark Set Design Choices

The whole validation process in SuperBench relies on a comprehensive benchmark set. The benchmark set should be able to discover all known incidents and evolve when unseen incident occurs for the first time. In this section, we present how it constructs a qualified set of validation benchmarks, encompassing both end-to-end benchmarks and micro-benchmarks.

End-to-end Benchmarks   As the validation process aims to ensure infrastructure reliability under certain customer workloads, we first choose benchmarking the most typical end-to-end AI models to represent customer workloads and prevent potential regressions preemptively. By examining the distribution of workloads within an internal first-party AI training platform and categorizing models based on keywords in commands and user logs, we can coorelate these workloads with several foundational models, including different CNN models [27, 29, 60] and Transformers like BERT [62] and GPT [15] models. To optimize GPU resources and ensure training convergence, customers must also carefully configure model parameters (e.g., batch size, sequence length, etc.). We analyze parameter distributions for different foundational models and choose the most prevalent settings.

Given that workloads are continuously evolving, and current typical end-to-end benchmarks may not accurately represent future workloads, we designed the benchmark set to be extensible. If existing benchmarks fail to detect incidents for a new workload, or private workloads run in specific clusters, the corresponding model can be added as an end-to-end benchmark and utilized for validation. This rarely happened in our experience, except for few new dominate models.

Micro-benchmarks   Rare workloads may exhibit distinct patterns when accessing specific hardware components, and the limited number of representative end-to-end benchmarks may not be able to encompass these rare workload behaviors. Therefore, in addition to end-to-end benchmarks for typical workloads, we employ component-wise micro-benchmarks to assess individual hardware components, including CPU, memory, GPU compute units, GPU memory, NIC/HCA, disk, and their interconnections such as PCIe, xGMI, NVLink, and network links. We leverage third-party tools accredited by hardware vendors for this purpose.

Furthermore, we derive pattern-wise micro-benchmarks in between to emulate the most common workload patterns or primitives involving several components. For example,

cuBLAS or cuDNN kernels with commonly used shapes, all-reduce or all-to-all collective communication primitives with commonly used message sizes, etc. These patterns and primitives are profiled offline from typical workloads and enriched with regression cases during the diagnostic experience.

3.3 Benchmark Selection

Given a node set N, the Selector will query real-time statuses for these nodes and predict incident probability for each node through an *offline probability model*. If their joint probability p of occurring an incident is higher than a threshold p_0, that is, the mathematical expectation of time to incident is shorter than the mathematical expectation of job duration, the Selector will select a benchmark subset for validation through an *online selection algorithm*. Otherwise the validation will be skipped to save node hours.

Offline Probability Model   Assuming that all nodes in a large-scale cluster share the same incident probability distribution, predicting the likelihood of each node experiencing an incident during a customer workload run requires modeling the node incident distribution as a continuous function of time:

$$P(T_{incident} \le t) = \int_0^t f(s)\,ds = F(t) \tag{1}$$

where $f(t)$ and $F(t)$ denote the density function and cumulative distribution function of the incident time. Traditional survival analysis [36] suggests different statistical methods to model such probability, e.g., exponential distribution which assumes failure rate is constant over time, Weibull distribution which constrains failure rate correlated to two parameters.

Since the failure rate of GPU nodes changes over time due to the gradual degradation pattern, we need to model it without any assumptions on failure rate distribution. The Cox-Time model [38], which is an extension of the mostly used Cox proportional hazards model [19] for time-to-event prediction in survival analysis, allows for time-varying hazard and removes the proportional assumption. We apply the state-of-the-art Cox-Time model with neural network [38] for this purpose, assume node failure rate is only related to time and some covariates, and treat node statuses as covariates to failure rate, including time since the last incident, historical incident counts, mean time between incidents, etc., in different incident categories. The Cox-Time model is trained with historical node incident data and will estimate incident probability distribution, given node's statuses and past incident history.

Online Selection Algorithm   We formulate the benchmark selection problem as follows. Given a set of benchmarks $B = \{B_1, \ldots, B_n\}$, the running time of benchmark B_i as t_i, and a set of defective nodes $\{M_i\}$ identified from the historical validation result data. We assume that the full set of benchmarks shall discover all incidents, a subset of benchmarks with coverage C can identify the incident with probability $p \times C$ hence

Algorithm 1 Benchmark Selection Algorithm

Input: node set N, benchmark set B, expected prob. p_0
Output: benchmark subset $SB \subseteq B$
1: **function** INCIDENTPROB(N, SB)
2:     $\mathcal{C}$ = num of defects subset SB found / num of defects full set B found (according to historical validation data)
3:     $p = 1 - \prod_{n \in N}(1 - \text{CoxTime}_{t_0}(n))$
4:     **return** $p \times (1 - \mathcal{C})$
5: **function** BENCHMARKSELECTION(N, B)
6:     $SB \leftarrow \varnothing$, $p \leftarrow$ INCIDENTPROB(N, SB)
7:     **while** $p > p_0$ **and** $SB \neq B$ **do**
8:         $curr \leftarrow (B_1, 0.0)$
9:         **for** B_i **in** B **do**
10:            $\Delta p \leftarrow p -$ INCIDENTPROB(N, $SB \cup \{B_i\}$)
11:            **if** $\Delta p/t_i > curr[1]$ **then** $curr \leftarrow (B_i, \Delta p/t_i)$
12:         $SB \cup = \{curr[0]\}$, $p \leftarrow$ INCIDENTPROB(N, SB)
13:     **return** SB

Algorithm 2 Criteria Calculation Algorithm

Input: sample set $S = \{S_1, \dots, S_N\}$, similarity threshold α
Output: criteria S_C for the given benchmark
1: **function** GETCENTROID(S)
2:     ▷ centroid can also be calculated by samples' mean in distribution space
3:     $median = \text{argmax}_{i=1}^n \sum_{j=1}^n \text{similarity}(S_i, S_j)$
4:     **return** S_{median}
5: **function** CRITERIACALC(S, α)
6:     $Defects \leftarrow \varnothing$, $S_C \leftarrow$ GETCENTROID(S)
7:     **while** $\min_{S_i \in S \setminus Defects} \text{similarity}(S_C, S_i) \leq \alpha$ **do**
8:         $Defects \leftarrow \{S_i \mid \text{similarity}(S_C, S_i) \leq \alpha\}$
9:         $S_C \leftarrow$ GETCENTROID($S \setminus Defects$)
10:     **return** S_C

---

decrease incident probability to $p \times (1 - \mathcal{C})$, where the coverage is defined as percentage of defective nodes identified by the benchmark subset in history. For example, suppose B identified 10 defective nodes in the past, B_1 identified 2 defects $\{M_1, M_2\}$ ($\mathcal{C} = 0.2$) while B_2 identified $\{M_2, M_3, M_4\}$ ($\mathcal{C} = 0.3$). The defect M_2 can be identified by either B_1 or B_2, so running the subset $\{B_1, B_2\}$ can only identify 4 defects with $\mathcal{C} = 0.4$. The goal of benchmark selection is to find a subset of benchmarks such that the new incident probability $p \times (1 - \mathcal{C}) \leq p_0$ while minimizing the total benchmark time.

Since each defective node may be covered by one or more benchmarks, this problem is NP-hard and can be considered as a variation of the 0-1 knapsack problem where different items overlap. We propose Algorithm 1 to greedily select benchmarks with $O(n^2)$ time complexity, where n is the number of benchmarks in benchmark set B, instead of traversing all possible combinations with $O(2^n)$ complexity, based on the sub-optimal probability decrement per time unit guideline.

3.4 Benchmark Criteria

Given the node set and selected benchmark subset, the Validator will execute the benchmarks on corresponding nodes to produce the benchmark results. For the first validation during cluster build-out, the full set of benchmarks are run and *offline criteria* is unsupervised learned for each benchmark upon its results on all nodes. For the following selective validations, the *online defects filtering* will compare the benchmark result on each node to its criteria and determine the defective node if any of its result violates the criteria. The Validator also adaptively tunes benchmark parameters for *repeatability*.

Offline Criteria To clearly compare benchmark results across different runs or nodes, we first define the similarity of two measured benchmark samples. This includes sam-

ples from micro-benchmarks, which may only have a single value as the result, or end-to-end benchmarks that track AI workload runs and record a series of performance numbers from repeated steps over a certain period. Given two time-series benchmark samples $S_1 = \{s_{1,1}, \dots, s_{1,n}\}$ and $S_2 = \{s_{2,1}, \dots, s_{2,m}\}$, to ensure there is no asymptotic or significant regression at any time, we define the distance and similarity between two benchmark samples in their distribution space based on the empirical cumulative distribution function (CDF):

$$d(S_1, S_2) = \int_0^\infty \frac{|CDF_{S_1}(x) - CDF_{S_2}(x)|}{\max(CDF_{S_1}(x), CDF_{S_2}(x))} \, dx \quad (2)$$

$$\text{similarity}(S_1, S_2) = 1 - d(S_1, S_2) \quad (3)$$

The distance $d(S_1, S_2)$ represents the absolute integral area between their CDF curves, normalized to the $[0, 1]$ range for similarity comparison among different benchmarks.

Having a benchmark and its result samples $S_1, \dots, S_N$ from different nodes, the criteria S_C is then calculated to differentiate defective result samples from normal ones. In order to establish a clear-cut boundary, for any S_i except defective samples, S_C should satisfy $\text{similarity}(S_C, S_i) > \alpha$, where α is an empirical value set by requirements. We therefore apply a similarity-based clustering algorithm to iteratively exclude defective samples and calculate the median sample for the rest as S_C, as shown in Algorithm 2.

Online Defects Filtering The Validator leverages benchmark similarity between the measured benchmark result and its criteria to determine whether the node should be considered defective or not. Since validation only requires testing whether the result on given node exhibits worse performance (i.e., lower throughput or larger latency) than the normal performance to identify defects, we compare the one-direction distance derived from Equation (2) between the runtime observed result sample S_{Obs} and offline criteria S_C for throughput-like metrics (elsewise replace max with min):

$$d_{1-side}(S_{Obs}, S_C) = \int_0^\infty \frac{\max(0, CDF_{S_{Obs}}(x) - CDF_{S_C}(x))}{\max(CDF_{S_{Obs}}(x), CDF_{S_C}(x))} \, dx \quad (4)$$

Similarity is then calculated by $1 - d_{1-side}$ and the same empirical threshold α is used to filter out under-performant nodes, which will be processed by the repair system next.

Repeatability Due to the natural variations when running the same workload across different runs or nodes, particularly in AI software/hardware, it is essential to ensure that validation benchmarks are minimally affected by such variations so that the measured samples can accurately represent the underlying distribution of both hardware performance and variation. We define the *repeatability* metric as *"the arithmetic mean of pairwise similarities from N different nodes or runs"* to ascertain whether benchmark settings are acceptable within the empirical similarity threshold α.

To effectively validate nodes, the *repeatability* metric must be maximized. However, out-of-the-box benchmarks from public tools or hardware vendors may exhibit large variations, making them unsuitable for validation requirements. We utilize existing benchmarks as a starting point and adhere to the guidelines below to enhance their repeatability:

1. Decouple different hardware components to distinguish various sources of variation. For example, binding processes to local cores and memory to avoid random remote memory access when validating GPU, loading training data from memory to eliminate disk access variation, etc.

2. Adaptively search for benchmark parameters to reduce benchmark duration for the given hardware/software combination, validating nodes in the shortest possible time while maintaining repeatability within the threshold.

3. After firmware/driver updates, re-tune and re-evaluate the repeatability in case it deteriorates on newer versions.

4 Implementation

Benchmark Set Table 2 shows the full benchmark set chosen for SuperBench. Each benchmark has pre-defined and configurable parameters, such as GEMM shapes, message sizes, model batch sizes, etc., as well as randomly generated input data. During each validation, the Selector selects a subset of benchmarks, and the Validator subsequently executes those benchmarks in two phases in sequence: the single-node phase and the multiple-node phase. In each phase, the selected benchmarks are performed in a bottom-up manner. Micro-benchmarks are conducted first to validate individual components, followed by end-to-end benchmarks to simulate customer workloads. Defective nodes are removed after each phase to ensure they do not affect subsequent benchmarks.

Networking Validation For the distributed microbenchmark which performs pairwise scans for RDMA verbs to ensure the networking quality of InfiniBand or RoCEv2 during large-scale AI training, we propose the following two algorithms to reduce validation time:

Full Scan in $O(n)$ The pairwise scans for all nodes is to ensure that the networking bandwidth between any two

Table 2: Full benchmark set in SuperBench.

	Micro benchmarks	End-to-end benchmarks

Single Node Phase	Computation
	• GPU kernel launch
	• GPU GEMM [5,45]
	• cuBLAS kernels
	• cuDNN kernels
	• GPU burn
	Communication
	• CPU latency [31]
	• GPU H2D/D2H bandwidth
	• GPU copy bandwidth
	• NVLink all-reduce
	• IB HCA loopback [56]
	• IB single-node all-reduce
	Comput./Comm. Overlap
	• MatMul/all-reduce overlap
	• Sharding MatMul
	Disk IO
	• FIO rand/seq read/write [10]
		Multi-GPU training
		• CNN models
		  – ResNet [27] 50/101/152
		  – DenseNet [29] 169/201
		  – VGG [60] 11/13/16/19
		• RNN models
		  – LSTM
		• Transformers
		  – BERT [62] base/large
		  – GPT-2 [15] small/large
		• Long-running stress
		  – GPT-2 large

Multiple Node Phase	Networking
	• All pair RDMA verbs
	• GPU collective communication [6,47]
	  – all-reduce
	  – all-gather
	  – all-to-all
		Multi-node training
		• CNN models
		• RNN models
		• Transformers
		• Long-running stress
		(same as above)

nodes are expected so that AI workloads with all-to-all traffic (e.g., Mixture-of-Experts models) won't have regression. The key idea is to simultaneously run network tests for all nodes, which can avoid traffic collision by leveraging the Clos network [16] property. The problem can be formulated as follows. Given a network with N NICs, suppose $N \in 2\,\mathbb{N}$ and pairs are symmetric, schedule all possible $\frac{N \cdot (N-1)}{2}$ NIC pairs into $N-1$ rounds. In each round, exactly $\frac{N}{2}$ pairs will run pairwise benchmarks (e.g., GPU-direct RDMA write [56], NCCL/RCCL [6,47] 2-node all-reduce) without NIC intersection. We address this problem by leveraging the circle method in round-robin tournaments [35] as the scheduling algorithm.

Quick Scan in $O(1)$ To eliminate the increasing time required for large-scale networking validation, we further propose topology-aware networking validation, which scans network links within a fixed duration, irrespective of the network scale or node number. We select node pairs such that the distance (number of hops) between each pair is 2-hop (nodes under the same ToR switch), 4-hop (nodes under the same aggregation switch), 6-hop (nodes across the core switch), and so on. For each hop number, we include all nodes without node overlap, ensuring that every node is included once and only once for the given hop. For a k-tier fat-tree network topology, only k rounds are needed for any scale. In each round, we run the selected networking benchmarks for each hop number.

Benchmark Parameter Searching For end-to-end benchmarks, validation only needs to measure a number of iterations after performance becomes stable, rather than running into model convergence or completion. We offline search for such optimal warmup and measurement steps to reduce the validation time. Since model structures and hyper-parameters like batch size and sequence length have already been determined based on representative workloads, the search focuses on the number of warmup steps w and measurement steps n. It aims to minimize the total step number, i.e., benchmark running time, while maximizing the repeatability across nodes. This process is performed offline.

We formulate the problem as follows: Assuming the benchmark runs K steps periodically and has a sequence of throughput numbers $t_1, \ldots, t_K$, the objective is to search for parameters $w, n \in [1, K]$ such that n is minimized and sub-sequence $t_w, \ldots, t_n$ is self-similar within the similarity threshold α (same as the threshold used in Section 3.4). First, we calculate the period of the cycle p across all K steps using classical seasonal decomposition by moving averages [59] and divide them into $\frac{K}{p}$ cycles. Next, we traverse from the beginning to calculate the pairwise similarity of all cycles, stop when a certain number of continuous similar periods are found. w and n are set to the beginning and end of those continuous periods. Finally, (w_i, n_i) from different nodes are traversed to select the one that maximizes the average of pairwise similarity across all nodes.

5 Evaluation

5.1 Experiment Settings

SuperBench has been deployed in Azure production environments since Day-0 when new AI SKUs were introduced, so it is hard to perform an apple-to-apple comparison between the complete SuperBench system and the original system without validation. Consequently, we divide our experiment settings into two parts: First, we gather node incident statistics and traces from on-premise clusters and use this data to simulate node incidents with and without validation, assessing the efficacy of the offline-trained incident probability model and the selection algorithm in the Selector. Second, we execute the full set of benchmarks on production clusters and two testbeds to collect benchmark results for criteria evaluation in the Validator.

Node Incident Trace We collect a 4-month period node incident events, including all node failures and customer tickets, from different internal on-premise GPU clusters with 1k nodes in total. These clusters share the same hardware as cloud VMs but never run validation. Each incident event has timestamps of when the event started and ended, incident reasons, and involved hardware components. We use this trace to fit the probability model with each node's status during simulation.

Node Allocation Request Trace The internal on-premise GPU cluster deploys a container orchestration system where users can submit GPU training jobs to run. We also collect allocation requests within the same clusters, including the requested node number, submission timestamp, and duration, to simulate node allocation requests for benchmark selection.

Cluster Benchmark Dataset We construct a benchmark dataset using the results obtained from the full set of benchmarks during the production cluster build-out. This dataset comprises 3k+ A100 VMs and 24 benchmarks with 2,441 metrics on each VM. We use this dataset to build criteria, label defective nodes, and calculate the defect coverage of each benchmark for the benchmark selection algorithm.

Testbed We utilize the following internal GPU testbeds:

- 144× MI250X VMs. Each VM has 8× AMD Instinct MI250X 120 GB GPUs [4] and 96× 2nd-Gen AMD Epyc cores. GPUs are inter-connected by xGMI links, while VMs are connected by 8× 200 Gbps HDR InfiniBand.

- 64× H100 VMs. Each VM has 8× NVIDIA H100 80 GB SXM GPUs [8] and 96× 4th-Gen Intel Xeon cores. GPUs are inter-connected by NVLink and NVSwitch, while VMs are connected by 8× 400 Gbps NDR InfiniBand.

Setup We deploy SuperBench to the testbeds and run all benchmarks with the Validator. The environment for validation uses the exact same settings as production clusters, including firmware versions, hypervisor settings, BIOS configurations, etc. We fixed all software versions by using a dedicated VHD image for VM and a Docker image for validation execution. The VHD image is based on Ubuntu 22.04 [11], while the Docker image is based on NVIDIA NGC container image 20.12 [49]. Both images include the necessary GPU/OFED drivers and related libraries. For the benchmark selection, we leverage PyCox [38] to build and train the probability model.

5.2 Benchmark Selection Evaluation

Offline Probability Model We evaluate whether the Cox-Time model in Section 3.3 can effectively use its fitted probability distribution to predict the *time before next incident* (TBNI) given node statuses, including total up time, historic incident counts and MTBI in different categories, etc. These status numbers are concatenated into a feature vector as the model's input, while the expected TBNI is used as prediction output. Failure to predict TBNI for the given nodes will result in inappropriate benchmark selections hence higher MTBI. We extract 46,808 node status samples and corresponding TBNI from the node incident trace. 80% of the samples are used for model training and 20% are used for evaluation.

We establish three baselines for comparison:

- Exponential Distribution $S(t) = e^{-\lambda t}$ [13]. It assumes the incident rate λ is constant across different node statuses.

Table 3: Accuracy of different probability models.

Model	Accuracy
Exponential Distribution	75.12%
Exponential Distribution per Incident Count	63.03%
Exponential Distribution per Hour	75.12%
Cox-Time Model	**93.13%**

- Exponential Distribution per Incident Count. It assumes the incident rate is simply related to historical incident count, as informed by Figure 4, and constructs exponential distributions for node statues with the same incident count.

- Exponential Distribution per Hour. It assumes the incident rate is related to current up time. The incident rate for each hour H is calculated by dividing the number of samples with at least an H-hour life by the total number of samples.

We compare the accuracy of different probability models. For each sample, which contains one single incident, the prediction accuracy is calculated by $\frac{\|[prediction]-[TBNI]\|}{2400}$, where the prediction is capped at the trace length of 2,400 hours (100 days) to make accuracy $\leq 100\%$. The model accuracy is defined as the average of its accuracy on all test samples.

The results are shown in Table 3. The first and third models yield the same accuracy since they both predict $> 2,400$ hours TBNI for all samples. The Cox-Time model, which considers multiple node statuse as covariates, achieves a high accuracy of 93.1%, significantly outperforming the other baseline models. This result also highlights the important relationship between node statuses and incident probability distribution.

Online Selection Simulation   To understand whether the benchmark selection can proactively identify defective nodes and save node hours, we simulate customer jobs in a cluster and evaluate whether the benchmark selection reduces incidents and improves utilization. The simulation will mark nodes with incidents under certain categories. Note that we assume the full set of benchmarks can always discover all incidents and whether the selected subset can proactively prevent the incident is decided by coverage from historical validation data. We simulate 720 hours (30 days) of cluster activities. The detailed simulation process is described as follows:

1. Simulator sets up FIFO queues for both job and node, and employ *stressed replay* based on the allocation request trace to schedule jobs to the nodes in a best-effort manner.

2. For current allocated nodes, simulator will sample the next incident time and an incident category for each node based on the probability distribution trained on incident trace.

3. Selector selects a subset of benchmarks by Algorithm 1.

4. Simulator calculates the probability on whether the selected subset can identify simulated incidents based on benchmark coverage instead of running actual benchmarks.
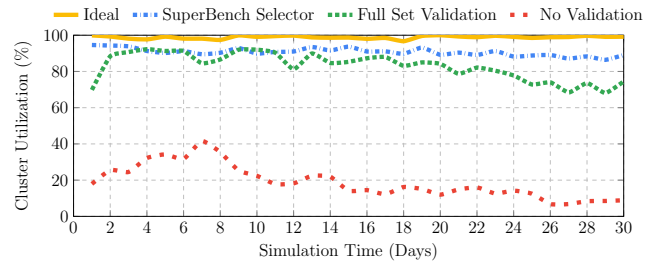

Figure 8: Simulated average node utilization with different benchmark selection policies within 30 days.

Table 4: Simulated average node validation time and MTBI with different benchmark selection policies in 30-day.

Policy	Absence	Full Set	SuperBench Selector
Validation Time (h)	0	100.40	**7.96**
MTBI (h)	11.59	236.26	**262.05**

5. If the Selector filters defects before job running, related defective nodes will be sent to repair, and the rest nodes and the job will be pushed to the end of respective queues.

6. If the Selector fails to predict defects that cause incidents during job running, a new benchmark selection will be performed based on updated node statuses. The job will be pushed to the queue rear and continue where it lefts off.

7. For the node repairing, the duration is 1.5 days (expectancy of ticket time in Figure 2) for no validation case to troubleshoot potential problematic nodes, and 1 hour for validation case to replace defects with healthy ones in hot buffer according to the empirical experience.

Apart from the Algorithm 1, we establish three baselines for comparison: (1) absence of validation, no validation and nodes are sent for repair upon each incident, (2) full set validation, where nodes run validation with the full set of benchmarks upon each job allocation and each incident, and (3) ideal baseline, where all nodes are healthy forever and no incidents occur.

We evaluate the effectiveness of different policies, focusing on three metrics: average node utilization, average node validation time cost, and node MTBI of a cluster. For each node, its utilization is calculated by dividing its up time by total time, its validation time is the sum of the durations of all validations, while its MTBI is calculated by dividing its up time by the number of incidents that occurred within it.

The average node utilization results are presented in Figure 8. SuperBench Selector with benchmark selection algorithm achieves a high cluster utilization of 90.70%, improving the no validation baseline by $4.81\times$ and the full set baseline by $1.09\times$. As depicted in Table 4, the Selector reduces 92.07% validation time cost per node compared to the full set baseline, which is irrelevant to simulation duration since validation is performed regularly in production. In terms of MTBI, the
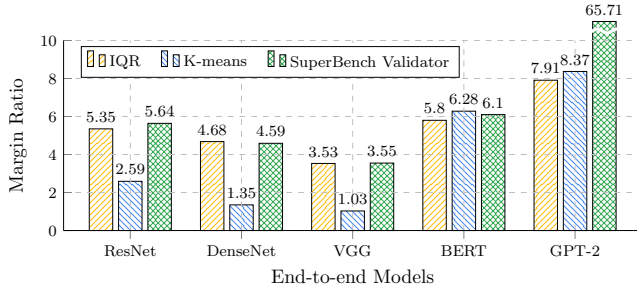
Figure 9: Margin ratios of different criteria methods.

Selector achieves a high 262.05 hours, enhancing the no validation baseline by $22.61\times$ and the full set baseline by $1.11\times$. The Selector experiences an average of 4.79 incidents per node, 0.16 higher than full set baseline. Despite more incidents, it benefits from reduced down time due to selective validation, resulting in longer up time and better MTBI.

5.3 Benchmark Criteria Evaluation

Criteria and Defects Filtering To evaluate benchmark criteria, we collect a series of step throughput for all end-to-end benchmarks on 144 MI250X VMs and generate the criteria using different methods to compare their effectiveness. Note that there is no ground-truth on which nodes are defective. Any nodes with failures or performance regressions are defined as defects by definition. To compare whether the criteria can maximize the margin between healthy and defective nodes, we define the *Margin Ratio* metric as $\frac{\min_{i \in defective} d(S_i, S_C)}{\max_{j \in healthy} d(S_j, S_C)}$.

Apart from the proposed method, we establish two baselines with typical outlier detection methods, including the interquartile range (IQR) [20] and k-means [26]. For the proposed method, we use criteria S_C calculated by Algorithm 2 and choose $\alpha = 0.95$ as the similarity threshold. For IQR, we use average throughput of each sample to calculate the lower quartile Q_1 and upper quartile Q_3 [20]. The S_C for IQR is then calculated as the median of samples higher than $Q_1 - 1.5 \cdot (Q_3 - Q_1)$. For k-means, we use the default Euclidean distance as the benchmark samples distance in the k-means clustering algorithm and set the cluster number $k = 2$. S_C is set to the average of all samples in the majority cluster.

As shown in Figure 9, out of a total of 5 models, IQR and k-means both have 4 models worse than the proposed Algorithm 2, which can achieve relatively better margin ratios across different benchmarks compared with baseline methods. For GPT-2, both baselines classify nodes with marginal performance as defective, which subsequently decreases the numerator in the margin ratio calculation, leading to an unclear boundary. On the contrary, the proposed method identifies them as healthy to maximize the margin between healthy and defective nodes, resulting in significantly larger margin ratio.

Table 5: Repeatability after benchmark parameters tuned.

End-to-end Models	Repeatability (FP32 / FP16)		Time Saving FP32 / FP16
	Fixed Parameters	Tuned Parameters	
ResNet	98.70% / 97.52%	98.70% / 97.55%	73.96% / 78.30%
DenseNet	99.13% / 98.82%	99.12% / 98.76%	73.96% / 73.96%
VGG	98.63% / 97.38%	98.62% / 97.51%	75.59% / 70.70%
LSTM	99.51% / 98.11%	99.52% / 98.11%	73.96% / 73.96%
BERT	99.62% / 99.44%	99.62% / 99.44%	73.96% / 77.21%
GPT-2	99.51% / 99.37%	99.48% / 99.36%	73.96% / 67.45%

Table 6: Effectiveness and repeatability in real deployment.

Benchmark	Repeatability	# Defects / # Total
IB HCA loopback	99.96%	6.04%
H2D/D2H bandwidth	99.68%	2.03%
BERT models	99.39%	1.59%
CPU latency	99.60%	1.33%
IB single-node all-reduce	99.90%	1.10%
ResNet models	99.21%	0.73%
GPT-2 models	99.19%	0.53%
LSTM models	98.66%	0.46%
DenseNet models	97.70%	0.40%
MatMul/all-reduce overlap	97.88%	0.33%
NVLink all-reduce	99.89%	0.30%
GPU GEMM	99.44%	0.23%

Repeatability We evaluate the effectiveness of the proposed adaptive benchmark parameter searching by comparing its repeatability to a baseline with sufficiently long and fixed step parameters, i.e., 72 warmup and 3,072 measurement steps. We execute end-to-end benchmarks on 64 H100 VMs and collect the warmup and measurement step numbers for each run. The repeatability metric is measured as the arithmetic mean of similarity scores between each sample and criteria. As shown in Table 5, repeatability regression of the proposed method (tuned parameters) is less than 1% compared to the baseline (fixed parameters), while saving $67.5\% \sim 78.3\%$ validation time cost for all models.

5.4 Deployment in Cloud

We have deployed SuperBench in Azure production environment for two years, where it acts as a quality gate to identify most defective nodes before delivering GPU clusters to public cloud services. We collect the benchmark dataset during the cluster build-out phase from 24k+ A100 GPUs (3k+ VMs) in 90 days and demonstrate its practicality.

Benchmark Effectiveness We generate criteria for the dataset and use it to filter out defective nodes with failure or regression, and calculate the percentage of defects identified by each benchmark, including micro-benchmarks and end-to-end benchmarks, as shown in Table 6. The defects percentages are sorted in reverse order, and each defective node may be filtered by one or several benchmarks. In total, 10.36% nodes are filtered as defects without duplication.

Repeatability at Scale   We also analyze the repeatability of those effective micro-benchmarks and end-to-end benchmarks among healthy nodes after filtering out defective ones. Table 6 also shows the repeatability on around 3k nodes, where most benchmarks have it greater than 99.0%, while all effective benchmarks have greater than 97.5% repeatability.

6   Related Works

Gray Failure in Cloud   There exist related works that explore the concept of gradual performance degradation under different terms, as opposite to fail-stop failures. For example, "fail-stutter" [9] model is proposed to address the issue of the same components behaving differently. Do et al. [22] and Gunawi et al. [24] evaluate and study the "limping" hardware in cloud which exhibits significant degradation. Huang et al. [30] discuss the "gray failure" problem in cloud systems and its differential observability traits. Lou et al. [41] evaluate the "partial failures" in software. Gunawi et al. [25] analyze a variety of "fail-slow" hardware incidents within production systems, providing valuable observations and suggestions.

In the new era of AI, computing resources become even more powerful [3] and such gradual degradation patterns cause more significant consequences due to gang-scheduled and synchronized AI workloads. We study this degradation pattern in the GPU-based cloud at scale and correlate it with the redundancies in hardware. Instead of diagnosing issues as they occur, we advocate for a proactive validation approach that regularly validates all components to restore redundancies, consequently mitigating the impacts of such degradation.

Elastic Training and Fault-tolerant AI   In synchronized AI workloads, one slow GPU can make all other GPUs wait and stall the entire training process. Popular deep learning frameworks support elastic distributed training to address this issue, such as Torch Elastic [53] and Horovod Elastic [28], which will scale down nodes when there are hardware failures and scale up after recovery. In addition to ML-agnostic solutions, there are also algorithmic aware fault-tolerant techniques [54, 55] for AI training to bypass faulty nodes.

However, these elastic and fault-tolerant techniques may also introduce non-determinism for workloads and affect training convergence or accuracy [40], which is not transparent to customers. In contrast, the validation system we proposed aims to reduce the faulty hardware in infrastructure from the cloud service provider's perspective with no assumptions on customer workloads.

Performance Benchmarking   Performance benchmarking is used to measure which computer system can complete tasks more quickly across different types. For traditional CPU-based systems, SPEC [21] and LINPACK [23] are widely-used to rank supercomputers. For AI systems, particularly GPU-based, benchmarking also shares the same goal of measuring which AI system is faster. DeepBench [57] compares

typical operations in deep learning, while DAWNBench [18] and MLPerf [43] consider end-to-end training or inference performance and measure time to convergence as metrics.

All the above benchmarks for benchmarking target ranking or competition for peak performance of different types of systems. In contrast, the performance validation we proposed in this work aims to identify defects across reproductions of the same system, which has entirely different requirements.

7   Conclusion

Inherent redundancies in cloud AI infrastructure can give rise to a unique degradation pattern for AI workloads, leading to diminished reliability. In this paper, we delved into incidents associated with this concealed degradation pattern and addressed it through the design of a proactive validation system, SuperBench. The system comprises a comprehensive benchmark suite, a Selector designed to efficiently trade-off validation time and coverage, and a Validator which executes benchmarks and filters defects with clear-cut criteria. The system also iterates and evolves with the rich data gathered by itself. Our evaluation in testbed and simulation demonstrates its ability to increase the MTBI by $22.61\times$. Moreover, Super-Bench has showcased its practicality in a real-world cloud deployment.

References

[1] AMD instinct accelerator claims. https://www.amd.com/en/claims/instinct. Accessed Jan 15, 2024.

[2] Democratizing AI with PyTorch foundation and ROCm support for PyTorch. https://pytorch.org/blog/democratizing-ai-with-pytorch/. Accessed Jan 15, 2024.

[3] TOP10 System - November 2023. https://www.top500.org/lists/top500/2023/11/. Accessed Jan 14, 2024.

[4] AMD. Introducing AMD CDNA 2 architecture. https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf. Accessed April 29, 2022.

[5] AMD. Next generation BLAS implementation for ROCm platform. https://github.com/ROCmSoftwarePlatform/rocBLAS. Accessed April 29, 2022.

[6] AMD. RCCL performance benchmark tests. https://github.com/ROCmSoftwarePlatform/rccl-tests. Accessed April 29, 2022.

[7] AMD. ROCm releases. https://github.com/RadeonOpenCompute/ROCm/releases. Accessed April 7, 2023.

[8] Michael Andersch, Greg Palmer, Ronny Krashinsky, Nick Stam, Vishal Mehta, Gonzalo Brito, and Sridhar Ramaswamy. NVIDIA Hopper architecture in-depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/. Accessed April 29, 2022.

[9] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, pages 33–38. IEEE, 2001.

[10] Jens Axboe. Flexible I/O tester. https://github.com/axboe/fio. Accessed April 29, 2022.

[11] Microsoft Azure. HPC images in Azure marketplace. https://github.com/Azure/azhpc-images. Accessed April 29, 2022.

[12] Microsoft Azure. Linux virtual machines pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/#pricing. Accessed April 7, 2023.

[13] Alessandro Birolini. *Reliability Engineering*, volume 5. Springer, 2007.

[14] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. LOF: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 93–104, 2000.

[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[16] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[17] Google Cloud. Cloud TPU pricing. https://cloud.google.com/tpu/pricing/#pricing-components. Accessed April 7, 2023.

[18] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Dawnbench: An end-to-end deep learning benchmark and competition. *Training*, 100(101):102, 2017.

[19] David R Cox. Regression models and life-tables. *Journal of the Royal Statistical Society: Series B (Methodological)*, 34(2):187–202, 1972.

[20] Frederik Michel Dekking, Cornelis Kraaikamp, Hendrik Paul Lopuhaä, and Ludolf Erwin Meester. *A Modern Introduction to Probability and Statistics: Understanding why and how*, volume 488. Springer, 2005.

[21] Kaivalya M Dixit. Overview of the SPEC benchmarks, 1993.

[22] Thanh Do, Mingzhe Hao, Tanakorn Leesataornwongsa, Tiratat Patana-anake, and Haryadi S Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–14, 2013.

[23] Jack J Dongarra. The linpack benchmark: An explanation. In *International Conference on Supercomputing*, pages 456–474. Springer, 1987.

[24] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesataornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM symposium on cloud computing*, pages 1–14, 2014.

[25] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)*, 14(3):1–26, 2018.

[26] John A Hartigan and Manchek A Wong. Algorithm AS 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[28] Horovod. Elastic horovod. `https://horovod.readthedocs.io/en/v0.27.0/elastic.html`. Accessed April 7, 2023.

[29] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4700–4708, 2017.

[30] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 150–155, 2017.

[31] Intel. Intel memory latency checker. `https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html`. Accessed April 29, 2022.

[32] Jithin Jose. MVAPICH2 at Azure: Enabling high performance on cloud. `https://hibd.cse.ohio-state.edu/static/media/talks/slide/Jithin-sc22-osu-bof.pdf`. Accessed April 7, 2023.

[33] Norm Jouppi and David Patterson. Google's cloud TPU v4 provides exaflops-scale ml with industry-leading efficiency. `https://cloud.google.com/blog/topics/systems/tpu-v4-enables-performance-energy-and-co2e-efficiency-gains`. Accessed April 7, 2023.

[34] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, et al. Service fabric: a distributed platform for building microservices in the cloud. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[35] Thomas P Kirkman. On a problem in combinations. *Cambridge and Dublin Mathematical Journal*, 2:191–204, 1847.

[36] John P Klein and Melvin L Moeschberger. *Survival Analysis: Techniques for censored and truncated data*, volume 1230. Springer, 2003.

[37] Ronny Krashinsky, Olivier Giroux, Stephen Jones, Nick Stam, and Sridhar Ramaswamy. NVIDIA Ampere architecture in-depth. `https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/`. Accessed April 29, 2022.

[38] Håvard Kvamme, Ørnulf Borgan, and Ida Scheel. Time-to-event prediction with neural networks and Cox regression. *arXiv preprint arXiv:1907.00825*, 2019.

[39] Jennifer Langston. Microsoft announces new supercomputer, lays out vision for future AI work. `https://blogs.microsoft.com/ai/openai-azure-supercomputer/`. Accessed Dec 6, 2022.

[40] Mingzhen Li, Wencong Xiao, Biao Sun, Hanyu Zhao, Hailong Yang, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, et al. EasyScale: Accuracy-consistent elastic training for deep learning. *arXiv preprint arXiv:2208.14228*, 2022.

[41] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 559–574, 2020.

[42] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 409–425, 2018.

[43] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. MLPerf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020.

[44] Yusuf Mehdi. Reinventing search with a new AI-powered Microsoft Bing and Edge, your copilot for the web. `https://blogs.microsoft.com/blog/2023/02/07/reinventing-search-with-a-new-ai-powered-microsoft-bing-and-edge-your-copilot-for-the-web/`. Accessed April 7, 2023.

[45] NVIDIA. CUDA templates for linear algebra subroutines. `https://github.com/NVIDIA/cutlass`. Accessed April 29, 2022.

[46] NVIDIA. CUDA toolkit archive. `https://developer.nvidia.com/cuda-toolkit-archive`. Accessed April 7, 2023.

[47] NVIDIA. NCCL tests. `https://github.com/NVIDIA/nccl-tests`. Accessed April 29, 2022.

[48] NVIDIA. NVIDIA A100 GPU memory error management - row-remapping. `https://docs.nvidia.com/deploy/a100-gpu-mem-error-mgmt/index.html#row-mapping`. Accessed Dec 6, 2022.

[49] NVIDIA. The NVIDIA container image for PyTorch. `https://docs.nvidia.com/deeplearning/frameworks/pytorch-release-notes/rel_20-12.html#rel_20-12`. Accessed April 29, 2022.

[50] OpenAI. Introducing ChatGPT. `https://openai.com/blog/chatgpt`. Accessed April 7, 2023.

[51] OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[52] Sundar Pichai. An important next step on our AI journey. `https://blog.google/technology/ai/bard-google-ai-search-updates/`. Accessed April 7, 2023.

[53] PyTorch. Torch distributed elastic. `https://pytorch.org/docs/2.0/elastic/quickstart.html`. Accessed April 7, 2023.

[54] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A Gibson, and Eric P Xing. Litz: Elastic framework for high-performance distributed machine learning. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 631–644, 2018.

[55] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric Xing. Fault tolerance in iterative-convergent machine learning. In *International Conference on Machine Learning*, pages 5220–5230. PMLR, 2019.

[56] Linux RDMA. InfiniBand verbs performance tests. `https://github.com/linux-rdma/perftest`. Accessed April 29, 2022.

[57] Baidu Research. Benchmarking deep learning operations on different hardware. `https://github.com/baidu-research/DeepBench`. Accessed April 29, 2022.

[58] Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. Estimating the support of a high-dimensional distribution. *Neural computation*, 13(7):1443–1471, 2001.

[59] Skipper Seabold and Josef Perktold. Statsmodels: Econometric and statistical modeling with python. In *Proceedings of the 9th Python in Science Conference*, volume 57, pages 10–25080. Austin, TX, 2010.

[60] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[61] Prasoon Sinha, Akhil Guliani, Rutwik Jain, Brandon Tran, Matthew D Sinclair, and Shivaram Venkataraman. Not all gpus are created equal: characterizing variability in large-scale, accelerator-rich systems. *arXiv preprint arXiv:2208.11035*, 2022.

[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

[63] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, pages 1–16, 2013.

[64] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[65] Rob Waters. Microsoft built a supercomputer to power OpenAI's ChatGPT - cybersecurity careers blog. `https://www.cybercareers.blog/2023/03/microsoft-built-a-supercomputer-to-power-openais-chatgpt/`. Accessed April 7, 2023.

[66] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT-175B logbook. `https://github.com/facebookresearch/metaseq/blob/main/projects/OPT/chronicles/OPT175B_Logbook.pdf`. Accessed May 19, 2022.

[67] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.