# KAIR: A Statistical and Causal Approach to Pinpointing Stragglers in Distributed Model Training

Yitang Yang[1], Junhong Liu[1], Jiapeng Chen[2], Xiaoyang Sun[3†], Tianyu Wo[1], Chunming Hu[1], Chengru Song[2], Jin Ouyang[2], Renyu Yang[1]
[1]Beihang University [2]Kuaishou Inc. [3]University of Leeds

***Abstract*—The distributed deep learning training process within large-scale clusters serves as the foundation of contemporary artificial intelligence. However, its inherent characteristics make it particularly sensitive to *stragglers*, specifically the presence of slow workers, which can significantly decelerate the entire procedure. Observability tools are essential for identifying stragglers within systems. However, the prevailing system profiling tools are either designed for single-node analysis, lacking visibility across multiple workers, or they recognize stragglers but only deliver high-level symptoms, providing engineers with insufficient insight into the underlying causes.**

**We design KAIR, a robust production-standard observability tool. KAIR uses an innovative hierarchical approach, transitioning from statistical anomaly detection to causal inference. It employs Kolmogorov-Smirnov statistics for the identification of statistically anomalous workers and implements a causal path tracing algorithm to accurately determine the specific operations, such as computation or communication, that are responsible for the delay. KAIR has been evaluated in a production cluster of 2,048 NVIDIA A800 GPUs and demonstrated high effectiveness in detecting latent stragglers at the framework level that are often overlooked by conventional tools. It offers precise suggestions that markedly reduce processing inefficiencies and engineering workload.**

***Index Terms*—Distributed Training, Performance Analysis, System Observability, and Straggler Detection**

## I. INTRODUCTION

The exponential increase in the size of deep learning models, scaling from millions to hundreds of billions of parameters [1], has made distributed large-scale training an indispensable technology. For instance, large language models (LLMs) [2] and deep learning recommendation models (DLRMs) [3], [4] necessitate extensive clusters comprising hundreds to thousands of GPUs to achieve training within a practical and feasible time horizon [5]. While distributed training solutions enhance scalability, they simultaneously introduce complexities within the system, such as communication overhead, synchronization delays, and challenges in failure recovery. Within this paradigm, the overall cluster throughput is constrained by the performance of its least efficient worker due to the presence of synchronization points, a phenomenon referred to as the *straggler effect* [6], [7]. This issue forms a significant performance bottleneck, contributing to increased operational costs and inefficiencies within real production environments, with stragglers documented to result in up to a 45% reduction in throughput within large-scale industrial clusters [8].

†Corresponding author: Xiaoyang Sun (X.Sun4@leeds.ac.uk)

The observability tool [9] represents an emerging methodology for comprehending system states and diagnosing failures through the utilization of logs, metrics, and trace data within microservices. Nonetheless, within the scope of distributed deep learning, conventional approaches [10], [11] encounter obstacles attributed to dynamic workloads [12], tightly coupled operations [13], and synchronized node patterns, thereby diminishing the efficacy of standard logging and tracing mechanisms. For example, TensorBoard [14] and PyTorch Profiler [15] are specifically developed to manage single-node conditions. Thus, these tools do not possess the capability to incorporate globally synchronized information within distributed environments. Other methods such as MegaScale [16] and Greyhound [17] are capable of identifying stragglers; however, they do not elucidate the underlying causes of these performance inefficiencies. Consequently, engineers observe the deceleration of the training job, yet they lack the analytical tools necessary to investigate the issue *why in a distributed context*.

To address this challenge, we present KAIR, a profiling tool designed for cluster environments, which enables engineers by furnishing *why*. This is achieved through the implementation of intra- and inter-worker runtime tracing and the application of a comprehensive set of robust statistical analysis algorithms. Furthermore, KAIR augments existing tools by providing empirical evidence essential for identifying the root causes of stragglers, such as unstable communication primitives or resource competition in computation-intensive kernels, thereby facilitating engineers in comprehending and optimizing the overall training process.

The paper makes the following contributions.

- **A scalable, low-impact observability architecture**: We introduce a robust production framework tailored for large-scale industrial applications, incorporating adaptive sampling alongside decoupled asynchronous data aggregation. It facilitates the monitoring of thousands of workers with negligible impact on performance.
- **A multistage anomaly analysis and causal engine**: We propose a novel multistage methodology that transitions from statistical anomaly detection to causal analysis. Initially, it employs statistical tests to identify anomalies in *which worker* and outliers in *which of its operations*. Subsequently, it conducts causal path tracing to ascertain the fundamental root cause.
- **A comprehensive validation in production clusters**:

KAIR was implemented with 1,500 lines of Python and 500 lines of C++. Its effectiveness is evaluated in Kuaishou's production setting, utilizing 2,048 A800 GPUs. This deployment effectively identified over 90% of latent framework-level stragglers, issues that were previously unresolvable using existing tools, thereby, highlighting its practical value and substantial impact.

## II. BACKGROUND AND MOTIVATION

The training of frontier AI models has evolved into an enterprise of unprecedented magnitude [16], [18], carried out on distributed clusters comprising thousands of GPUs. Within this framework, where training hinges on tightly-coupled synchronous steps, the overall throughput is dictated by the slowest participant, referred to as the *straggler*. This phenomenon leads to considerable computational and financial inefficiency. A recent study [8] on large-scale training clusters demonstrated that stragglers cause 42.5% of all jobs to experience a slowdown of at least 10%. At its extremes, this inefficiency culminates in the waste of more than 45% of allocated GPU hours, posing a multi-billion-dollar challenge.

These stragglers can arise from a range of issues such as hardware degradation, network congestion, I/O jitter, or imbalances in data partitioning [8], thereby complicating the diagnostic process [17]. While optimizing training performance, engineers must not only identify slow-running workers but also pinpoint *where* and *why* their performance is suboptimal in relation to others.

The existing tools, including TensorBoard [14] and PyTorch Profiler [15], specialize in delivering a detailed view of execution within an individual worker, effectively visualizing operator timelines and GPU kernel activity. Nevertheless, these tools are architecturally incapable of detecting inter-worker performance fluctuations that are associated with the straggler problem. Modern straggler-aware tools [19]–[21] offer various approaches to handle anomalies in distributed settings, but each has drawbacks. Symptom-based detection [16] uses high-level metrics to identify problems, but requires manual intervention for root cause analysis. Disruptive validation [17] uses a two-phase method to identify faults, but its necessity to pause operations makes it impractical for immediate diagnosis in real-time environments. Offline performance modeling [22] aims at performance prediction and optimization rather than addressing straggler issues.

It is vital to develop a novel profiling paradigm to provide a *global perspective* and advance *beyond symptom detection to offer causal evidence*. Engineers require a tool capable of precisely identifying not only the straggler worker, but also the specific operations responsible for delays, without interrupting mission-critical training tasks. KAIR effectively addresses this essential shortcoming through an innovative, non-intrusive hierarchical approach that transitions from statistical anomaly detection to causal root-cause analysis.

## III. OUR APPROACH

The KAIR system has two main components illustrated in Figure 1: a *scalable data aggregator* to collect detailed runtime data from workers with minimal impact using dynamic sampling and a *multistage analysis engine* to convert these data into a unified analysis panel. This decoupled design is

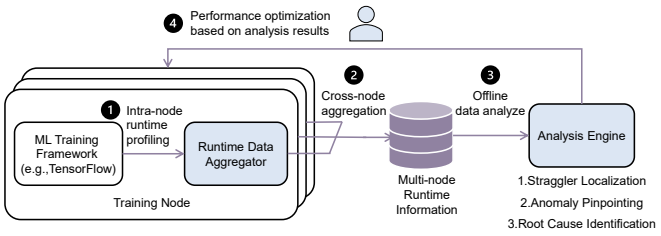


Fig. 1. The overview of our approach.

fundamental to achieving both clear visualization and low overhead across hundreds or thousands of workers.

The following section describes the architecture and the innovative hierarchical causal analysis, which transitions from high-level anomaly detection to the identification of root causes, particularly individual kernels.

### A. Scalable and Sampling-Driven Data Aggregator

Collecting detailed traces from thousands of workers simultaneously presents a significant challenge in data volume and performance. The KAIR's aggregation framework is engineered to address this through following key mechanisms, also shown in Figure 2.

**Feature1: Multi-level Intra-Worker Profiling**

KAIR captures performance metadata from two distinct levels of the software stack to provide a holistic view of system behavior. At the **hardware & kernels level**, it uses the NVIDIA CUPTI [23] library to obtain ground-truth data on GPU kernel durations and execution timestamps. At the training **framework level**, it connects to native profiling interfaces, such as TensorFlow [24] and PyTorch [25], to record the scheduling and execution of logical operators.

KAIR customizes two framework triggers `PerfScopeBegin` and `PerfScopeEnd`, which can be automatically integrated into the computation graph to categorize operations into meaningful high-level stages (e.g., forward pass, backward pass, optimizer step), allowing the profiling process to be aligned with the overall training procedure and facilitating hierarchical analysis.

KAIR exhibits considerable **compatibility and integration**. It is developed utilizing Kuaishou's proprietary distributed training framework, referred to as KAI. This framework seamlessly integrates profiling operations within the computational graphs of TensorFlow or PyTorch during the training phase. This integration enables thorough profiling, thereby improving the understanding of training dynamics and helping to identify performance issues.

**Feature2: Adaptive Sampling during Data Collection**

Rather than collecting all data continuously, KAIR employs a feedback-driven sampling strategy to minimize overhead. The system continuously monitors low-cost, high-level metrics, such as the variance in iteration time across the cluster. If this variance exceeds a predefined threshold, a strong indicator of a potential straggler, KAIR automatically triggers a more detailed, fine-grained trace collection at suspected nodes and a random subset of their peers for comparison. This intelligent approach ensures that KAIR operates with minimal overhead during normal execution but captures the rich data needed for analysis precisely when a problem arises.

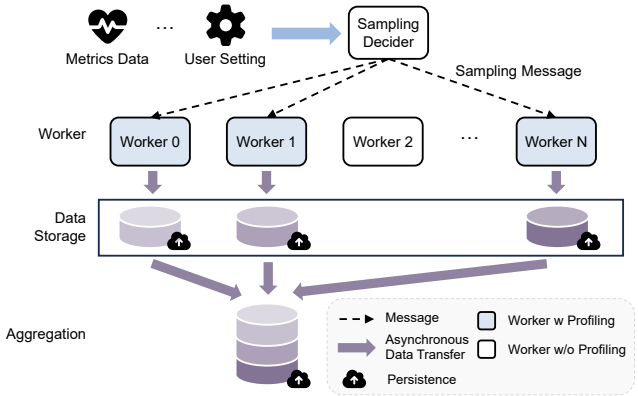


Fig. 2. Runtime data collection in KAIR.

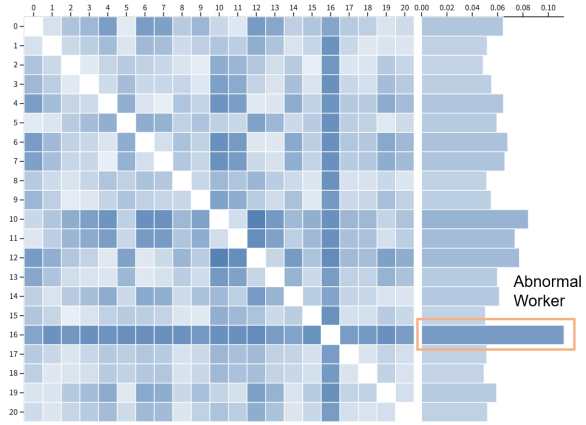


Fig. 3. Distribution divergence heatmap for worker anomaly detection.

**Feature3: Decoupled Asynchronous Aggregation**

To minimize the impact on the training process, KAIR's architecture decouples data collection from data persistence. On each worker, trace events are written to a lock-free in-memory ring buffer with minimal overhead. A separate, low-priority background process is responsible for reading asynchronously from this buffer. Once a complete training iteration is captured, the process compresses the trace data into a custom columnar format, optimized for both storage size and fast analytical queries, and transfers it to a shared distributed storage system. Within distributed storage, an additional coordination daemon runs independently to aggregate trace fragments from all nodes. This process enables unified cluster-level profiling views without interfering with ongoing training workloads. The design prevents training process interruptions from resource-heavy operations like I/O or network transmission, keeping profiling overhead minimal and constant, regardless of cluster size (i.e., total workers).

**Feature4: Cross-Worker Time Synchronization**

The *clock synchronization* is crucial for cross-worker analysis. KAIR achieves microsecond accuracy by aligning trace timestamps with a global clock and correcting clock offsets during profiling through the most concise heartbeat message.

### *B. Multistage Anomaly Analysis Engine*

The core of KAIR's innovation is its three-stage diagnostic engine, which implements a systematic *localize, then pinpoint* workflow. This structured top-down filtering makes the analysis of billions of events across thousands of workers computationally tractable and directs engineers from a cluster-wide slowdown to a practical/addressable root cause.

**Stage1: Straggler Localization via Distribution Divergence.**

The first step is to identify which worker is behaving differently from its peers. Each worker has an *execution rhythm*, the rate at which it completes operations over time. A straggler's rhythm will be measurably distinct from that of healthy workers. KAIR quantifies this divergence by creating a temporal frequency distribution for each worker $\mathcal{F} = \{f_{t_1}, f_{t_2}, \ldots, f_{t_n}\}$, which counts the number of operators completed within a number of fixed-length time windows. From these data, the empirical cumulative distribution function (ECDF) is derived $F(t_k) = \sum_{i=1}^{k} f_{t_i}$.

To compare any two workers, $A$ and $B$, KAIR uses the pairwise **Kolmogorov-Smirnov (K-S)** test [26], a non-parametric statistical test that measures the maximum vertical distance between their ECDFs, $D_{A,B} = \max_t |F^A(t) - F^B(t)|$. The K-S statistic is ideal for this task as it is sensitive to differences in both the location and shape of the distributions, allowing it to detect both consistent delays and intermittent stalls.

An abnormality score $S_A = \frac{1}{N-1} \sum_{B=1, B\neq A}^{N} D_{A,B}$, is calculated for each worker by averaging its K-S divergence from all peers. Then, a heatmap of these scores immediately highlights the most anomalous worker as the primary straggler candidate, such as *worker#16* in Figure 3.

**Stage2: Anomaly Pinpointing via Statistical Outlier Detection**

Once a straggler worker is identified, the focus shifts to pinpointing which specific operation(s) within that worker are anomalous. In a healthy and synchronized cluster, the start time and duration of any given operator should be tightly concentrated around a mean value following a normal distribution. KAIR uses this assumption to detect outliers using the **z-score**, a standard statistical measure that quantifies how many standard deviations a data point has from the mean. For each operator instance $i$ with start time $s_i$ and duration $d_i$, KAIR computes its z-score relative to the cluster-wide mean ($\mu$) and standard deviation ($\sigma$).

An operation anomaly score, $a_i = \max\{\frac{|s_i - \mu_s|}{\sigma_s}, \frac{|d_i - \mu_d|}{\sigma_d}\}$, quantifies its deviation. Following the empirical 3-sigma rule [27], any operator with $a_i > 3$ is flagged as a statistically significant outlier, producing a ranked list of the most suspicious operations within the straggler.

**Stage3: Root Cause Identification through Causal Path Analysis**

The list of statistical outliers from Stage2 identifies *symptoms*, but might not be root causes. For example, an `AllReduce` operator may start late not because of a network issue but because it was the *victim* of a delay in the preceding `BackwardPass` computation. To distinguish culprits from victims, KAIR performs a novel causal analysis via **Backward Blame Attribution**.

First, KAIR reconstructs a step-specific execution dependency graph (a directed acyclic graph) from the collected traces. When an operator $O_i$ is flagged as anomalous, the attribution algorithm initiates a backward trace through this graph with the following logic:

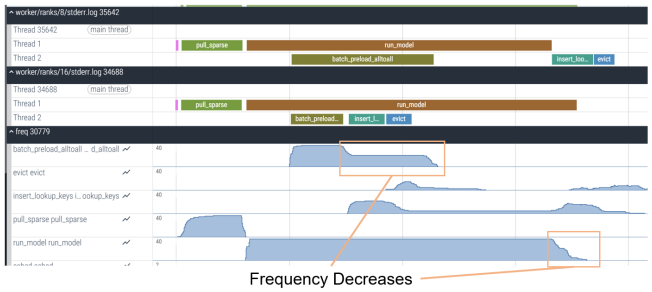


Fig. 4. Time-frequency view of sub-graph execution.

1) If $O_i$ has an abnormal execution time, it will be considered a candidate for the primary root cause, indicating a computational bottleneck in its worker. The backward blame attribution trace is terminated.
2) If $O_i$ has a normal execution time, but an anomalous start time, the algorithm examines its direct predecessor $O_{i-1}$ to check the reason for the delay in starting.
   - If $O_{i-1}$ itself had an anomalous execution time, the blame is propagated backward to $O_{i-1}$, as it is the likely cause of the delay.
   - If $O_{i-1}$ had a normal execution time, the above process repeats from $O_{i-1}$ to find its predecessor(s).
   - If $O_{i-1}$ completed on time but $O_i$ still started late, the *dependency edge* ($O_{i-1} \rightarrow O_i$) is flagged as the bottleneck. This indicates that the preceding computation was fine, but data transfer or synchronization between $O_{i-1}$ and $O_i$ caused this delay.

This causal tracing transforms the diagnostic output from a list of slow operators into a ranked list of probable root causes, either specific compute-bound kernels or slow inter-worker communication, providing engineers with precise and actionable optimizations for improvement.

**Option: Frequency-Based Visualization for Execution Analysis**

In addition to the three-stage detection workflow, KAIR introduces a complementary visualization method to further enhance interpretability. Specifically, KAIR extends the conventional timeline visualization by introducing the *frequency* metric to each operation to support the distributed training analysis, shown in Figure 4. The frequency metric captures the distribution of a given operator that occurs over time between different workers. By identifying regions with degraded or missing normal frequencies, KAIR effectively reveals inconsistencies in execution behavior among workers.

## IV. EVALUATION

KAIR was evaluated to answer three fundamental practical research questions:

- RQ1-**Efficiency**: How much is its runtime overhead introduced by tracing data collection?
- RQ2-**Scalability**: Is its data aggregation scalable?
- RQ3-**Availability**: Is it usable in real production-scale scenarios?

### A. RQ1: Runtime Overhead of Data Collection

**Experiment Setup**. The runtime overhead was evaluated on a small-sized cluster of four physical machines, each with two NVIDIA A800 GPUs and 2TB RAM. The inter-node bandwidth is 64 GB/s. The training framework is TensorFlow 2.13.0, integrated with KAIR to train a DLRM model.

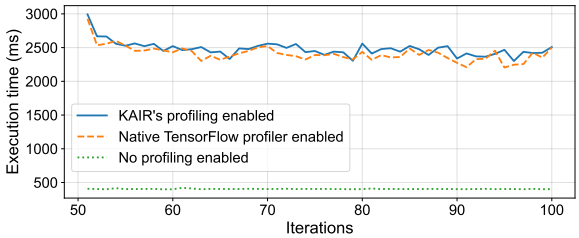


Fig. 5. Profiling overhead comparison between baselines and KAIR.

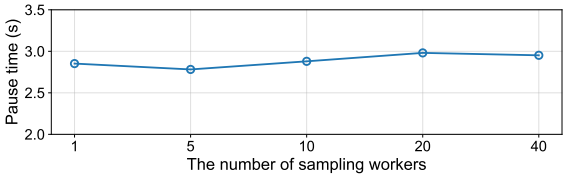


Fig. 6. Scalability of data aggregation in KAIR.

**Baselines**. Two traditional execution modes are set as baselines: (i) profiling disabled in the training framework and (ii) native TensorFlow profiling enabled.

**Metrics**. The iteration time is recorded after the warm-up phase (e.g., 50 iterations), and the final iteration time is calculated as the mean value for all workers.

**Results**. Figure 5 illustrates that profile operation causes quantifiable overhead. The native TensorFlow profiler, as well as KAIR, introduces a delay relative to the baseline configuration without profiling. Nevertheless, the KAIR approach maintains a performance level on par with TensorFlow's profiler while offering supplementary insights, including detailed operator execution events and subgraph-level timing. These insights are essential for diagnosing issues related to distributed performance.

Furthermore, KAIR supports step-level sampling, enabling users to decrease the frequency of data collection. By adjusting the sampling interval, developers can optimize the trade-off between profiling detail and computational overhead, enhancing the tool's applicability for extended training tasks.

### B. RQ2: Scalability of Multi-Worker Data Aggregation

**Experimental Setup**. The scalability overhead was evaluated on a cluster of forty NVIDIA A800 GPUs. Each machine has two GPUs and 2TB RAM. The inter-node bandwidth is 64 GB/s. The training framework is TensorFlow 2.13.0, integrated with KAIR to train a DLRM model.

**Metrics**. We assess *training pause time* to evaluate the negative effects of the profile on the overall training process. In order to evaluate performance of scalability, we employed various numbers of sampling workers.

**Results**. As illustrated in Figure 6, the *training pause time* remains nearly constant as the number of workers sampled increases. This is attributed to the asynchronous data aggregation mechanism of KAIR, which offloads I/O and inter-worker coordination to a background daemon process.

When profiling data are flushed, the training process only incurs a brief pause to hand off the data locally. The subsequent aggregation and persistence are handled independently, without stalling the model training process. This design

ensures that, even as the profile scale increases, the overhead imposed on training remains minimal, demonstrating the scalability and efficiency of our profile architecture.

### C. RQ3: Availability in Production-Scale Clusters

KAIR was implemented based on Kuaishou's internal training framework, with 1,500 lines of Python and 500 lines of C++ programs. It has been deployed and evaluated in Kuaishou production environments, over a cluster of 2048 NVIDIA A800 GPUs.

KAIR performs effectively to help engineers optimize the bottleneck in the overall training process. An example of a real-world scenario is identifying delays in the worker-level execution stage.

In the process of upgrading the training framework version, engineers observed a persistent lag between training workers. Standard logging and TensorFlow's profiler failed to identify inter-worker issues. Employing the anomaly analysis of KAIR on a large-scale training task running on 26 machines, each with 8 GPUs (208 GPUs in total), they detected a graph compiler optimization delay affecting operators at an early stage. This discovery necessitated a refinement of the compiler pipeline, effectively resolving the problem. This scenario illustrates the capability of KAIR to identify stragglers, analyze inter-worker alignment, and ascertain root causes, thereby facilitating data-driven diagnostics in complex distributed systems and minimizing reliance on intuition or manual verification.

## V. RELATED WORK

**Distributed Training Profilers and Observability Tools**. Several specialized frameworks and tools [28]–[31] have emerged to provide visibility into distributed deep learning workloads. Notably, dPRO [22] is a framework-agnostic profiler that collects fine-grained computation and communication traces across multiple nodes and constructs global data flow graphs to support replay and optimization. While dPRO excels in modeling and predicting performance, as well as applying optimizations for computation and memory in heterogeneous frameworks (TensorFlow [24], MXNet [32]), it does not provide high-level subgraph timing or node anomaly detection via statistical methods—capabilities that KAIR addresses.

**Performance Modeling and Simulation Tools**. Previous work such as Daydream [33], DistSim [34], and dPRO have developed simulation-based or analytical models to predict distributed training performance. These works focus on estimating execution time based on operator-level cost models or trace replay simulation. Our approach differs in KAIR provides fine-grained real-time observability at runtime, rather than offline or predictive performance modeling.

**Observability in Distributed Training Systems**. Several systems like TensorBoard and PyTorch Profiler offer profiling functionality mainly for single nodes. Advanced tools such as ByteDance's Hostping [35] and Alibaba's C4D [36] provide diagnostics for RDMA-level [37] performance and collective communications [38] in distributed environments though they lack subgraph-level observability and cross-node statistical anomaly detection. In contrast, KAIR builds upon multi-layer profiling (hardware, operator, subgraph) and adds quantitative detection of both operator-level and node-level anomalies.

## VI. DISCUSSION

KAIR identifies stragglers by analyzing the behavioral divergence at synchronization points. Since virtually all distributed training tasks, regardless of the model architecture (e.g., Transformer [39]) or domain-specific applications (e.g., computer vision [40], natural language processing [41]), rely on periodic synchronization, such as the communication primitives `all-reduce` and `all-gather`, the straggler problem is a universal bottleneck. Our approach, by targeting this common failure issue, is not confined to a specific workload. The methodology of KAIR' is extensible to mainstream frameworks, such as PyTorch [25] and JAX [42], requiring only minimal adaptation to their specific event tracking APIs.

The primary industrial contribution of KAIR is its ability to efficiently and accurately pinpoint which worker/operation is the straggler preventing the progress of the cluster. Deployed as a frontline troubleshooting tool in Kuaishou's production clusters, KAIR underpins the timely detection of fail-slow manifestations across thousands of daily model training jobs, directly addressing the review criterion of industrial relevance and impact.

Although KAIR excels at localization, an even more intricate challenge is automating the final step of root-cause analysis. Our future work, now under active development, focuses on this by synthesizing logs, system metrics, and other observability data to move from pinpointing a bottlenecked operator to suggesting a specific root cause, such as a memory leak or a specific hardware fault.

## VII. CONCLUSION

In this paper, we present KAIR, an observability system designed to address significant limitations in current tools by transcending mere symptom identification to deliver comprehensive and causal diagnostic insights. Employing a low-overhead, scalable data collection architecture along with an innovative hierarchical analysis engine, KAIR can autonomously determine not only which worker is experiencing delays but also pinpoint the specific computational or communication bottleneck causing the inefficiency. Its efficacy and practicality have been corroborated through its deployment in a substantial industrial setting at Kuaishou. KAIR constitutes a pivotal advancement toward data-driven, efficient debugging of intricate performance challenges, thereby augmenting the reliability and performance of the infrastructure underlying contemporary AI systems.

## ACKNOWLEDGMENTS

This work is supported in part by National Key R&D Program of China (Grant No. 2024YFB4505901), in part by the National Natural Science Foundation of China (Grant No. 62402024), in part by the Beijing Natural Science Foundation (Grant No. L241050), in part by the Fundamental Research Funds for the Central Universities, and, last but not the least, by Kuaishou Research Fund. For any correspondence, please refer to Dr. Xiaoyang Sun (X.Sun4@leeds.ac.uk).

## REFERENCES


[1] P. Villalobos, J. Sevilla, T. Besiroglu, L. Heim, A. Ho, and M. Hobbhahn, "Machine learning model sizes and the parameter gap," *arXiv preprint arXiv:2207.02852*, 2022.

[2] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, vol. 1, no. 2, 2023.

[3] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.

[4] J. Deng, S. Wang, K. Cai, L. Ren, Q. Hu, W. Ding, Q. Luo, and G. Zhou, "Onerec: Unifying retrieve and rank with generative recommender and iterative preference alignment," *arXiv preprint arXiv:2502.18965*, 2025.

[5] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[6] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Addressing the straggler problem for iterative convergent parallel ml," in *Proceedings of the seventh ACM symposium on cloud computing*, 2016, pp. 98–111.

[7] M. F. Aktaş and E. Soljanin, "Straggler mitigation at scale," *IEEE/ACM Transactions on Networking*, vol. 27, no. 6, pp. 2266–2279, 2019.

[8] J. Lin, Z. Jiang, Z. Song, S. Zhao, M. Yu, Z. Wang, C. Wang, Z. Shi, X. Shi, W. Jia, Z. Liu, S. Wang, H. Lin, X. Liu, A. Panda, and J. Li, "Understanding stragglers in large model training using what-if analysis," in *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI '25)*, 2025.

[9] M. Usman, S. Ferlin, A. Brunstrom, and J. Taheri, "A survey on observability of distributed edge & container-based microservices," *IEEE Access*, vol. 10, pp. 86 904–86 919, 2022.

[10] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu, "Enjoy your observability: an industrial survey of microservice tracing and analysis," *Empirical Software Engineering*, vol. 27, no. 1, p. 25, 2022.

[11] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," 2010.

[12] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "Pipedream: Fast and efficient pipeline parallel dnn training," *arXiv preprint arXiv:1806.03377*, 2018.

[13] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 463–479.

[14] Google, "Tensorboard," https://www.tensorflow.org/tensorboard, 2025.

[15] Meta, "Pytorch profiler," https://docs.pytorch.org/docs/stable/profiler.html, 2025.

[16] Z. Jiang, H. Lin, Y. Zhong, Q. Huang, Y. Chen, Z. Zhang, Y. Peng, X. Li, C. Xie, S. Nong, Y. Jia, S. He, H. Chen, Z. Bai, Q. Hou, S. Yan, D. Zhou, Y. Sheng, Z. Jiang, H. Xu, H. Wei, Z. Zhang, P. Nie, L. Zou, S. Zhao, L. Xiang, Z. Liu, Z. Li, X. Jia, J. Ye, X. Jin, and X. Liu, "MegaScale: scaling large language model training to more than 10,000 GPUs," in *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI '24)*, 2024.

[17] T. Wu, W. Wang, Y. Yu, S. Yang, W. Wu, Q. Duan, G. Yang, J. Wang, L. Qu, and L. Zhang, "GREYHOUND: Hunting Fail-Slows in Hybrid-Parallel training at scale," in *2025 USENIX Annual Technical Conference (USENIX ATC '25)*, 2025, pp. 731–747.

[18] X. Sun, W. Wang, S. Qiu, R. Yang, S. Huang, J. Xu, and Z. Wang, "Stronghold: fast and affordable billion-scale deep learning model training," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–17.

[19] A. Ravikumar and H. Sriraman, "Dpro-sm–a distributed framework for proactive straggler mitigation using lstm," *Heliyon*, vol. 10, no. 1, 2024.

[20] X. Liu, Y. Li, R. Ranjan, and D. N. Jha, "Pluggable ai-based real-time stragglers detection framework in hadoop," *High-Confidence Computing*, p. 100341, 2025.

[21] E. Ozfatura, S. Ulukus, and D. Gündüz, "Straggler-aware distributed learning: Communication–computation latency trade-off," *Entropy*, vol. 22, no. 5, p. 544, 2020.

[22] H. Hu, C. Jiang, Y. Zhong, Y. Peng, C. Wu, Y. Zhu, H. Lin, and C. Guo, "dPRO: A generic performance diagnosis and optimization toolkit for expediting distributed dnn training," in *Proceedings of Machine Learning and Systems (MLSys)*, 2022.

[23] Nvidia, "Cuda profiling tools interface," https://docs.nvidia.com/cupti/, 2025.

[24] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: a system for Large-Scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[25] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[26] V. W. Berger and Y. Zhou, "Kolmogorov–smirnov test: Overview," *Wiley statsref: Statistics reference online*, 2014.

[27] R. Lehmann, "3 $\sigma$-rule for outlier detection from the viewpoint of geodetic adjustment," *Journal of Surveying Engineering*, vol. 139, no. 4, pp. 157–165, 2013.

[28] C. Shin, G. Yang, Y. Yoo, J. Lee, and C. Yoo, "Xonar: Profiling-based job orderer for distributed deep learning," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. IEEE, 2022, pp. 112–114.

[29] Q. Zhao, H. Wu, Y. Hao, Z. Ye, J. Li, X. Liu, and K. Zhou, "Deepcontext: A context-aware, cross-platform, and cross-framework tool for performance profiling and analysis of deep learning workloads," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2025, pp. 48–63.

[30] A. Sharma, V. M. Bhasi, S. Singh, R. Jain, J. R. Gunasekaran, S. Mitra, M. T. Kandemir, G. Kesidis, and C. R. Das, "Analysis of distributed deep learning in the cloud," *arXiv preprint arXiv:2208.14344*, 2022.

[31] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM symposium on operating systems principles*, 2019, pp. 1–15.

[32] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[33] H. Zhu, A. Phanishayee, and G. Pekhimenko, "Daydream: Accurately estimating the efficacy of optimizations for DNN training," in *2020 USENIX Annual Technical Conference (USENIX ATC '20)*, 2020, pp. 337–352.

[34] G. Lu, R. Chen, Y. Wang, Y. Zhou, R. Zhang, Z. Hu, Y. Miao, Z. Cai, L. Li, J. Leng, and M. Guo, "Distsim: A performance model of large-scale hybrid distributed DNN training," in *Proceedings of the 20th ACM International Conference on Computing Frontiers (CF '23)*, 2023, p. 112–122.

[35] 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)*, 2023, pp. 15–29.

[36] J. Dong, B. Luo, J. Zhang, P. Zhang, F. Feng, Y. Zhu, A. Liu, Z. Chen, Y. Shi, H. Jiao, G. Lu, Y. Guan, E. Zhai, W. Xiao, H. Zhao, M. Yuan, S. Yang, X. Li, J. Wang, R. Men, J. Zhang, C. Zhou, D. Cai, Y. Xie, and B. Fu, "Boosting large-scale parallel training efficiency with c4: A communication-driven approach," *arXiv preprint arXiv:2406.04594*, 2024.

[37] A. Gangidi, R. Miao, S. Zheng, S. J. Bondu, G. Goes, H. Morsy, R. Puri, M. Riftadi, A. J. Shetty, J. Yang *et al.*, "Rdma over ethernet for distributed training at meta scale," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 57–70.

[38] J. Romero, J. Yin, N. Laanait, B. Xie, M. T. Young, S. Treichler, V. Starchenko, A. Borisevich, A. Sergeev, and M. Matheson, "Accelerating collective communication in data parallel training across deep learning frameworks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 1027–1040.

[39] 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*, vol. 30, 2017.

[40] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis, "Deep learning for computer vision: A brief review," *Computational intelligence and neuroscience*, vol. 2018, no. 1, p. 7068349, 2018.

[41] K. Chowdhary, "Natural language processing," *Fundamentals of artificial intelligence*, pp. 603–649, 2020.

[42] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/jax-ml/jax