Sleuth: A Trace-Based Root Cause Analysis System for Large-Scale Microservices with Graph Neural Networks

Yu Gan
mingshu.gy@alibaba-inc.com
Alibaba Group
Hangzhou, Zhejiang, China

Guiyang Liu
wuming.lgy@alibaba-inc.com
Alibaba Group
Hangzhou, Zhejiang, China

Xin Zhang
gavin.zx@alibaba-inc.com
Alibaba Group
Hangzhou, Zhejiang, China

Qi Zhou
jackson.zhouq@alibaba-inc.com
Alibaba Group
Hangzhou, Zhejiang, China

Jiesheng Wu
jiesheng.wu@alibaba-inc.com
Alibaba Group
Hangzhou, Zhejiang, China

Jiangwei Jiang
xiaoxie@alibaba-inc.com
Alibaba Group
Hangzhou, Zhejiang, China

ABSTRACT

Cloud microservices are being scaled up due to the rising demand for new features and the convenience of cloud-native technologies. However, the growing scale of microservices complicates the remote procedure call (RPC) dependency graph, exacerbates the tail-of-scale effect, and makes many of the empirical rules for detecting the root cause of end-to-end performance issues unreliable. Additionally, existing open-source microservice benchmarks are too small to evaluate performance debugging algorithms at a production-scale with hundreds or even thousands of services and RPCs.

To address these challenges, we present Sleuth, a trace-based root cause analysis (RCA) system for large-scale microservices using unsupervised graph learning. Sleuth leverages a graph neural network to capture the causal impact of each span in a trace, and trace clustering using a trace distance metric to reduce the amount of traces required for root cause localization. A pre-trained Sleuth model can be transferred to different microservice applications without any retraining or with few-shot fine-tuning. To quantitatively evaluate the performance and scalability of Sleuth, we propose a method to generate microservice benchmarks comparable to a production-scale. The experiments on the existing benchmark suites and synthetic large-scale microservices indicate that Sleuth has significantly outperformed the prior work in detection accuracy, performance, and adaptability on a large-scale deployment.

ACM Reference Format:
Yu Gan, Guiyang Liu, Xin Zhang, Qi Zhou, Jiesheng Wu, and Jiangwei Jiang. 2023. Sleuth: A Trace-Based Root Cause Analysis System for Large-Scale Microservices with Graph Neural Networks. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623278.3624758

1 INTRODUCTION

The recent emergence of cloud-native technologies, such as microservices architecture, containerization, container orchestration, service meshing, monitoring, and tracing, has significantly simplified the development and deployment of large-scale cloud applications [11, 15, 28, 47, 51, 72]. The leading cloud service providers, including AWS, Azure, Alibaba Cloud, and Google Cloud, have native support for the Kubernetes (K8s) engine, so application owners can easily deploy their cloud-native microservices to tens of thousands of nodes in data centers around the world with just one click [10, 12, 13, 31]. In addition, microservices offer greater scalability, flexibility, and agile iteration capabilities than traditional monolithic applications, which are especially important when business logic is developing rapidly. As a result, cloud-native microservices are becoming the one of most popular web application architecture [20].

As the microservice scale in size, it becomes increasingly challenging to meet the service-level objectives (SLOs, e.g., tail latency or availability targets). Root cause analysis (RCA) investigates underlying causes of performance issues in a microservice system. It is considered a difficult problem due to its scale and complex dependency graph. Distributed tracing is a popular tool to debug performance issues in large-scale microservices since it can visualize the path of a single request as it traverses across different components of a distributed system [3, 4, 9, 53, 62]. Although being straightforward, large-scale traces consisting of hundreds or even thousands of spans can be overwhelming and challenging to understand quickly by site reliability engineers (SREs). Thus, many trace-based RCA algorithms are proposed to determine the source of a performance issue automatically. They are based on a wide range of techniques, including static or dynamic rules [33, 67], heuristics [17, 43, 61, 64, 68], time-series analysis [35], graph analysis [15, 42], and machine learning (ML) [27, 29, 40, 44, 66, 70, 74].

As microservices scale in size, the effectiveness of heuristics and empirical rules employed in many RCA algorithms becomes diminished or even invalidated. As an example, we simulate a microservice system growing in size, with the latency of each microservice generated with heavy-tail log-normal distributions that we learned from a production microservice application. Figure 1 shows the trend of F_1 score and accuracy (defined in Section 6.1.5) by locating the root cause of *n-sigma* rule, a rule of thumb assuming normal data falls with in n standard deviation of the mean value.

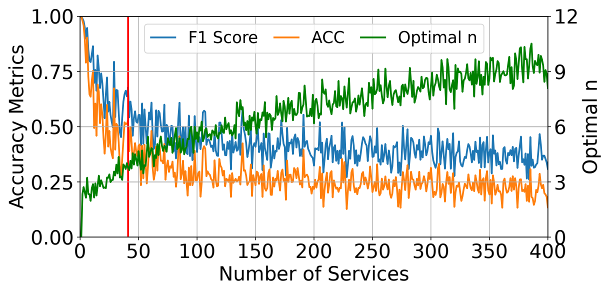


Figure 1: The F_1 score, accuracy (ACC, defined in Section 6.1.5), and the optimal n of root cause detection with the n-sigma rule as the number of microservices scales. The vertical line in red indicates the max number of services in the existing microservice benchmark suites.

The accuracy metrics drop sharply as the number of microservices increases. Such a trend can also be observed with other heuristic-based methods such as correlation thresholds, percentile thresholds, and workload-aware thresholds. Those methods perform worse at a larger scale because the impact of a single microservice on the entire trace is diluted as the scale increase. It is also worth noting that while *3-sigma* is often the magic number, it is no longer optimal when the service scales up because the service latency distribution is inherently heavy-tailed. Previous machine learning and statistical approaches surpass heuristic-based techniques in terms of accuracy by learning more refined patterns. Nonetheless, the network models employed in these approaches often lack generalizability for new applications and impose elevated overheads when implemented on a larger scale [17, 27, 29, 40].

We present Sleuth, an unsupervised trace-based system to identify the root causes of performance issues for large-scale microservices. Specifically, it captures the causal relationships of spans with a causal directed acyclic graph (DAG) constructed from the RPC dependency graph. It uses a graphical neural network (GNN) with redesigned aggregation layers to generate counterfactual examples for root cause localization. To prevent repetitive ML inferences with similar traces of the same anomaly type, we cluster the traces with a trace distance metric derived from Jaccard similarity, which accounts for the structure and state of a trace. Unlike the prior work, Sleuth is highly generalizable across different microservice applications. Because the architecture of the Sleuth model is independent of the RPC dependency graph, a pre-trained model can be applied to various microservices with only a few-shot fine-tuning. We evaluate Sleuth with existing microservice benchmarks, as well as a set of large-scale synthetic gRPC microservices which is comparable to the scale of many production microservice applications in the industry. The evaluation results show that Sleuth has significantly higher accuracy than the rule-based and heuristic-based RCA methods, and it is easier to train and more flexible with different microservice applications than other ML-based algorithms.

In summary, we have made three contributions in this paper:

- We design a GNN-based model to identify the root cause of SLO violations in microservices, which outperforms the prior work in accuracy, performance, and flexibility, particularly when applied to large-scale scenarios.
- We propose a light-weight distance metric to measure the similarity between two traces for clustering. The metric is capable of distinguishing traces with different call paths and execution patterns, including latency and error status.
- To quantitatively evaluate the proposed method in large-scale microservices, we design an evaluation methodology with synthetic benchmark generation. The benchmark can be generated at arbitrarily large scale and deployed in Kubernetes clusters to mimic the behaviors of production services in the real world.

2 RELATED WORK

2.1 Microservice Characterization and Benchmarking

As microservice design pattern gains popularity in the industry, there has been a large amount of work on designing microservice benchmarks and studying their characteristics. Acme Air [65] is a microservice benchmark for booking flight tickets with six services written in Java. μSuite [63] is a microservice benchmark suite consisting of several data-intensive (OLDI) services for key-value stores, set intersections, recommendations, and image similarity matching. DeathStarBench [28] contains several end-to-end microservices, including social networks, movie reviewing services, an e-commerce site, and a hotel reservation site, for studying implications of microservices across different hardware and software stacks. TrainTicket [73] is another microservice benchmark system with multiple programming languages and frameworks, containing 41 microservices for the research of performance debugging. Several demo projects from the industry are also used as microservice benchmarks by many prior studies in the field, including the SockShop [6] and the Online Boutique [32].

Although there are plenty of microservice benchmarks from the community, none of them has a scale comparable to the production microservices in the industry. For example, recent studies on microservice characterization with Alibaba Trace [47, 48] suggest that there can be hundreds of microservices in an application and more than thousands of calls in one RPC call graph. As the scale of production microservices continues to grow [72], the gap between the popular microservice benchmarks and the industry microservices is becoming larger, which makes them inapplicable to evaluate techniques developed for production-scale microservices.

2.2 Distributed Tracing and Cloud Observability

Distributed tracing and cloud performance monitoring tools are important for maintaining a highly-available microservice system. X-trace [26] is a distributed tracing framework to diagnose runtime performance issues in distributed systems. Pivot Tracing [49] presents a trace monitoring system that leverages the happen-before operator as well as other filter and group operations to help the engineers to troubleshoot the system. The Mystery Machine [19] infers causal relationships between requests by mining a large number of distributed traces. Observability is so important to managing cloud applications that many frameworks and tools have been designed by the community for collecting, visualizing, and analyzing the applications' traces, logs, and metrics. For example, Dapper [62], Zipkin [9], Jeager [3], and OpenTelemetry [53] are popular frameworks for distributed tracing; Prometheus [5] and Grafana [39] are opensource tools for collecting, storing and visualizing metrics; Fluentd [25] and Logstash [46] are designed for processing and storing the logs and events.

2.3 Cloud Application Performance Diagnosis

Data-driven methods have been popular tools for performance diagnosis due to the abundance of cloud monitoring data. CauseInfer [17] and Microscope [42] are based on PC-algorithm [36] for causal discovery from data. After a causal DAG is constructed, they locate the root cause with different threshold-based algorithms. However, learning the causal DAG from observational data is NP-hard [18], which makes the PC-algorithm-based algorithms impractical to be applied to large-scale microservice call graphs. Groot [67] is an event-graph-based RCA system used in eBay. It relies on multiple data sources and a range of techniques, including empirical rules, statistical models, and deep learning models to identify root causes. It requires domain-specific knowledge from the site reliability engineers (SREs) to refine the causal graphs and rules, which may not be easily transferable to other systems. TraceAnomaly [44], TraceRCA [41], MicroRCA [68], and MicroHECL [43] leverage empirical rules and correlations to locate root causes, which may fail as the system scale grows as is shown in Figure 1. Seer [29] and MEPFL [74] are two supervised learning models for microservice performance debugging. They need labels during training, which are often not available in production services. Sage [27] and CIRCA [40] are two causal inference-based models which construct causal DAGs from the system architecture and insights from the SREs. They infer the root cause by generating counterfactuals from a generative model and by interventions from a regression model, respectively. Since they train a separate model for each variable in the causal DAG, the models cannot be efficiently trained on GPU clusters and they are not generalizable to another microservice system with few-shot learning.

3 TECHNIQUES

3.1 Overview

Sleuth is a data-driven system to locate the root cause of anomaly traces. A set of services, pods, or nodes are defined as *root cause instances* for anomaly traces if restoring them to a normal state would prevent the trace from violating the SLO. Specifically, it fetches abnormal traces from the database, encodes the traces as tensors, and transforms the features. Then, traces are clustered by DBSCAN with our proposed trace distance metric. The geometric median of each cluster is selected as the representative, whose root causes are generalized to the entire cluster. The RCA system uses GNN to predict the duration and error status of the trace and determine the instances in the trace that result in anomalies.

We adhered to the following four design principles when developing Sleuth:

- **Unsupervised learning:** Labeled cloud telemetry data for microservice RCA is seldom available. This is due to the large volume of machine-generated data and the expertise required to interpret them. As a result, it is not possible to label cloud telemetry data using low-cost methods such as crowdsourcing, as has been done for images, natural languages, and videos [14]. Like many previous ML-based systems [27, 40, 68, 70], Sleuth leverages unsupervised learning to discover the root cause of anomaly traces without any labeled data during training.
- **Scalability and flexibility:** Sleuth is a trace RCA system designed for microservice applications running in production environments on public clouds. It is highly scalable and flexible, thanks to its GNN-based architecture, which does not rely on the structure of microservice RPC graphs. Sleuth can be efficiently trained on multi-GPU and multi-node clusters. To reduce the pressure of ML inference when a transient fault causes a large number of traces to fail, Sleuth defines a trace distance metric and uses it to cluster the abnormal traces before RCA.
- **Few-shot learning:** Sleuth is designed to be generalizable to different microservice applications, even with few or no fine-tuning samples. This is achieved by semantic-aware feature encoding and the use of a graph neural network layer that is specifically designed for modeling the temporal and causal relationships between microservices.
- **Efficiency:** Sleuth is a highly efficient system for troubleshooting microservice applications. It only requires a small fraction of tracing data, without requiring any additional data sources. Sleuth can be trained from scratch on a GPU server in just several to tens of hours, or even faster if fine-tuned from a pre-trained model. During inference, Sleuth can be deployed on a low-cost CPU server to complete an RCA query in less than one second for a thousand-span trace.

3.2 Trace Feature Engineering

The data engineering pipeline of Sleuth selects useful features and transforms them into a tensor for trace clustering and RCA.

3.2.1 *Feature Selection.*

Spans that conform to the most popular OpenTelemetry tracing specification [54] typically contain dozens of attributes, not all of which are equally important for root cause detection. Sleuth reconstructs the RPC dependency graph of a trace via `spanID` and `parentSpanID`. Instead of `spanID`, it collects `service` (the service where the RPC is called), `name` (operation name), and `kind` ("client" and "server" for synchronous RPCs, "producer" and "consumer" for asynchronous RPCs, and "local" for local function calls) as an identifier of a span. Although `spanID` is the unique ID of a span, its uniqueness and lack of semantics make it inappropriate for learning the relationships of spans which can be generalized to a wide variety of similar traces. It also collects `start` (start timestamp), `end` (end timestamp), `duration` and `statusCode` to represent the execution status of a span. To maximize the compatibility of Sleuth with traces generated by different sources, we do not leverage other attributes, logs, and metrics for trace RCA, because they are not required fields in the OpenTelemetry tracing convention and may not always be available.

3.2.2 *Feature Transformation.*

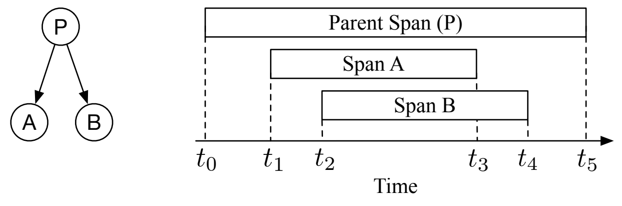


Figure 2: A trace example of a simple microservice system with the RPC dependency graph on the left.

- **Text feature encoding:** Well-designed `service` and `name` in the trace data typically contain semantic information that can be generalized to other traces or even to other microservice applications. For example, `GET` operations in Redis usually have similar latency behaviors and error patterns, even if they are from two different microservice applications. Therefore, to leverage the semantic information, we generate sentence embeddings with a pre-trained sentence-BERT model [59] for encoding `service` and `name`. It encodes the text with a 768-dimensional vector, and the distance of the embedding vectors quantitatively indicates the difference in meaning between the two texts. The text features are pre-processed to remove the special characters, separate camel case words, and replace long hex digits with placeholders. It is worth noting that Sleuth can handle billions of spans during training, which would require tens of terabytes of additional space if the embedding vectors for each span were kept on disk. Fortunately, since the number of different service names and operation names is much lower than the number of spans, we can store the embeddings of these different attributes separately, leaving only pointers to the embedding vectors for each span in the trace data, which significantly reduces storage and memory usage.
- **Exclusive duration and error calculation:** The duration of a non-leaf synchronous span consists of the time spent in that service and the wait time of its child spans. Disentangling the self-duration from the waiting time helps us to locate the root cause of a high-latency trace because the self-duration of a span can be used to predict the amount of time to be saved potentially if the performance issue related to that span is alleviated. While approximating the self-duration from traces is possible, it is difficult to calculate the exact value without extra annotations. For instance, in the trace shown in Figure 2, $t_1 - t_0$ and $t_5 - t_4$ can be considered as part of the self-duration of the parent span P. However, it cannot be directly inferred from the trace whether the parent span is simply waiting for responses from span A and span B or processing in parallel while waiting for the children from t_2 to t_3. Therefore, we propose the *exclusive duration* of a span as an indicator of the span's self-duration. Exclusive duration is the total time when the span does not overlap with any of its child spans, for example, $(t_1 - t_0) + (t_5 - t_4)$ for parent span P, $(t_3 - t_1)$ for span A, $(t_4 - t_2)$ for span B. Self-duration can be an immeasurable and non-intuitive function of the exclusive duration, which can be approximated via a deep learning model. Similarly, errors can propagate across spans as well. Thus, we define a span to have an *exclusive error* if it has an error of its own instead of originating from its child spans. Sleuth computes the exclusive duration and exclusive error of every span in the trace dataset in the run time.

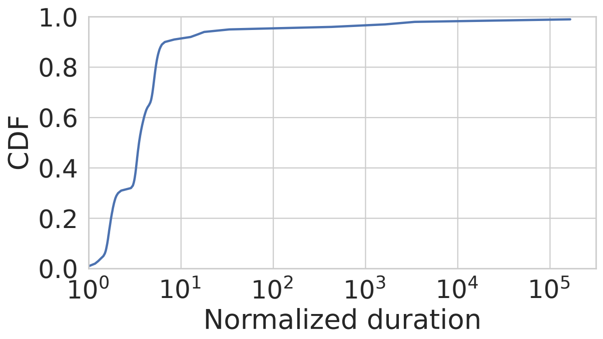


Figure 3: Cumulative distribution function of span durations in logarithm scale, normalized to the minimum duration.

- **Span duration transformation:** The distribution of span duration is extremely skewed. Figure 3 shows a cumulative distribution function (CDF) of normalized span durations of a production microservice system on a log scale. Although the duration of more than 90% of the spans is within 10x of the minimum duration, the duration dramatically increases to over 165,000x for the top 1% spans, making it difficult for an ML model to predict without proper scaling. Therefore, we scale the duration values with base-10 logarithm transformation. Moreover, we standardize them with a global mean of 4.0 and a standard deviation of 1.0 so that the model can be directly applied to all trace datasets without rescaling.

3.3 Trace Clustering

Trace clustering is vitally important to trace anomaly RCA in production microservices since it can reduce the number of expensive ML inference operations by orders of magnitude. In the case of a performance incident, hundreds or even thousands of anomaly traces can be categorized into a few failure modes with the same root causes. As the root causes can be generalized to other traces in the same category, the trace RCA system only needs to compute once for a representative trace in each cluster. The design of trace clustering consists of two steps. First, we determine the trace distance measure to quantitatively represent the similarity of two arbitrary traces. Second, we choose a clustering algorithm to efficiently classify the traces based on a trace distance metric.

3.3.1 Trace Distance Metric. The distance metric fundamentally determines the accuracy and performance of the clustering algorithm. The tree-edit distance (TED) [71] is a natural solution to calculate the trace distance, as traces can be represented as trees of function calls. However, TED does not scale well on large trees. The start-of-the-art implementation of TED, namely APTED [55, 56], has the average time complexity of $O(m^2 \log(m)^2)$ and worst-case time complexity of $O(m^4)$, where m is the number of nodes in a tree (i.e. number of spans in a trace). Therefore, it is not applicable for measuring the distances of two traces with hundreds or even thousands of spans each. Recently, in DeepTraLog [70], the researchers proposed a deep learning model with Gated GNN and support vector data description (SVDD) to learn the graph embedding of a trace and detect anomaly traces through clustering. Although the Euclidean distance of two vectors in the embedding space can be used as the distance of the two traces intuitively, it is not guaranteed that traces close to each other have the same root cause. In fact, as is shown in Section 6.2, DeepTraLog often maps traces with different root causes closely when reducing the radius of the hypersphere, which results in a lower detection accuracy after clustering with SVDD.

To address the limitations of tree-based distance metrics, we propose a new metric based on the Jaccard-index similarity. This metric converts the similarity of two trees into a similarity measure between two weighted sets of spans, with a time complexity of $O(m)$ to compute distances.

We encode a trace A into a weighted trace set with spans, denoted as S_A. Each element i of the set is a unique identifier for a span with its weight w_{A_i} being the duration of this span. The weight of an element demonstrates its significance to the trace as if it were stored w_{A_i} times in an ordinary set. The span identifier is a tuple of the service name, span name, span kind, error status, and the names of all its ancestor spans within a maximum distance of d_{max}. This allows us to represent the spans' calling paths in a trace. Different spans sharing the same identifier are merged in the set with the weight equal to the sum of their durations. When the structure and runtime characteristics of two traces are similar, elements with similar weights can be found in both trace sets.

We use the extended Jaccard index to calculate the similarity of trace sets. The distance between two traces A and B can be computed using Eq. 1, where $d(A,B)$ is an extended Jaccard distance on a weighted set normalized to $[0,1]$ by$|S_A \bigcup S_B|$. It is designed to be more sensitive to high-duration spans as they contribute more significantly to the entire trace than low-duration spans.

$$\begin{aligned} & d(A,B) = 1 - \frac{|S_A \bigcap S_B|}{|S_A \bigcup S_B|} = 1 - \frac{|S_A \bigcap S_B|}{|S_A| + |S_B| - |S_A \bigcap S_B|}, \\ & \text{where } |S_A| = \sum_{i \in S_A} w_{A_i},\ |S_B| = \sum_{i \in S_B} w_{B_i}, \\ & |S_A \bigcap S_B| = \sum_{i \in |S_A \bigcap S_B|} \min(w_{A_i}, w_{B_i}), \\ & \text{and } |S_A \bigcup S_B| = \sum_{i \in |S_A \bigcup S_B|} \max(w_{A_i}, w_{B_i}). \end{aligned} \tag{1}$$

3.3.2 *Trace Clustering Algorithm.* We use the HDBSCAN algorithm [50] with the distance defined above to cluster traces. The algorithm is a density-based clustering algorithm that does not require the number of clusters to be known a priori. The hyperparameters of the algorithm, namely, `min_cluster_size`, `min_samples` and `cluster_selection_epsilon`, are initialized with 10, 5, and 1, respectively. These hyperparameters are then adjusted according to the number and variation of the traces to be clustered. After clustering, we choose the trace with the minimum sum of distances to all other traces within the cluster (i.e. geometric median of that cluster) as the representative of this cluster. If the algorithm can accurately classify traces with different root causes into different clusters, then the root causes of the representative traces can be derived across the entire cluster.

3.4 Causal Modeling of Traces with GNN

Sleuth generates counterfactual queries to locate the root causes of abnormal traces [27]. It utilizes causal Bayesian networks (CBNs) [57] to model the causality relationships of spans, which indicates how errors flow in a trace. It is a commonly used causal model used in many previous microservices RCA papers because it can be derived directly from the RPC dependency graphs of microservices and can explicitly model the propagation of anomalies in the distributed system [17, 27, 40, 42, 68].

The CBN represents the causal relationships via a directed acyclic graph (DAG). In particular, Sleuth directly constructs the CBN from the RPC dependency graph from a trace and models the causal relationship of duration and error status from downstream spans to upstream spans since those two attributes are direct indicators of trace anomalies. While learning the structure of the CBN is straightforward, inferring the causal impact between two spans from the graph is non-trivial. First, the causality functions are usually non-linear. As an RPC can be called asynchronously or in parallel with other RPCs, the causality impact of that RPC on its parent is dependent on the states of its sibling RPCs.

Figure 2 illustrates the simplest possible example of why the non-linearity occurs. Only until $t_3 > t_4$ does span A start to exist in the critical path and contribute to the parent span's latency. Likewise, the latency of span B can be completely hidden by span A if $t_4 < t_3$. Therefore, it is impossible to accurately model the causal relationship with a linear model, such as linear structural equation modeling (SEM) [34]. Second, when a service sends tens or even hundreds of child RPCs to its back-end layer, their causality may be a function in a high-dimensional space. Due to the *curse of dimensionality*, the sparsity of the feature space increases exponentially on a fixed volume dataset, leading to over-fitting of the model. Third, the RPC dependency graphs of a microservice can be complicated and dynamic as the system evolves. The prediction model needs to be highly scalable to accommodate large graph sizes and flexible enough to work with a variety of RPC dependency graph topologies.

In this section, we introduce a model-based GNN which partially exploits domain knowledge of anomaly propagation in microservice traces to generate counterfactuals based on the RPC dependency graph.

3.4.1 *The Trace GNN Design.* The main advantage of using a GNN to learn causality in spans is that the GNN's structure is independent of the graph topology. This is important because the topology of a trace can vary from one request to another, due to the various types of operational flows that may be sent out from clients. Additionally, the topology of a trace can change significantly as microservices are frequently updated. A model whose structure depends on the graph topology, such as the graphical variational autoencoder (GVAE) in Sage [27], would need to update the connections of its neurons for different topologies in traces. However, the GNN aggregates messages from neighbors using permutation invariant functions, whose results do not rely on the order of the neighbors [37]. Therefore, the GNN can learn the status propagation functions of spans in traces with any graph topology without explicitly updating its model architecture.

In Sleuth, the CBN of a trace is defined as a DAG $G = (V, E, X)$ where V is a list of nodes (i.e. spans), E is the edge list and $X \in \mathbb{R}^{|V|\times d}$ is a matrix of node attributes (i.e. span attributes). Because of the Markov property of the causal DAG [57], Sleuth only needs one GNN layer to model the causality from child spans to the parent span.

We leverage the domain knowledge of the propagation of span duration and span errors to design the GNN layer. It predicts the duration by Eq. 2, where $a' = \text{power}(10,\ \sigma\cdot a+\mu)$, is unscaled value of a given variable a, d and d^* are duration and exclusive duration of a span, respectively. $\hat{d}_i$ is the predicted duration of a span i, which can be calculated by the sum of clipped ReLu functions of the duration of all its child spans (i.e. $j \in \mathcal{N}(i)$). Eq. 2 is based on the observation that a child span does not start to contribute to the parent span's latency until its own latency exceeds a minimum u'_j due to parallel executions, and it stops contributing to the parent span's latency at a maximum v'_j as requests can time out. u'_j and v'_j parameterized by two non-negative variables $h'_{j,1}$ and $h'_{j,0}$ to enforce $u'_j \le v'_j$. The equation is also capable of representing the causal relationship of asynchronous RPCs where the duration of publisher spans is not directly impacted by the subscriber spans when $u_j = v_j$. Because of the complexity of the microservice system, u_j and v_j will be learned from a neural network in Eq. 4.

The error status of a span can be predicted by Eq. 3, where e is the error status of a span, and e^* is the exclusive error status of a span. $\hat{e}_i \in [0, 1]$ is the maximum of its exclusive error and the duration and error status of its child spans which guarantees that the error related to the parent spans can be properly propagated along the causal DAG.

Eq. 2 and Eq. 3 are parameterized by $\mathbf{h}_j$ learned from a deep neural network with Eq. 4. $\mathbf{x}_i$ is the node attribute with duration and error status, $\mathbf{x}^*_i$ is the node attribute with exclusive duration and exclusive error status, and $\mathcal{S}(j)$ is the set of sibling spans of span j sharing the same parent span i in the RPC dependency graph. f_Θ is a variant of Graph Isomorphism Network (GIN), which has been shown to outperform other popular GNNs, such as GCN and GraphSAGE, in distinguishing the graph structures [69]. We use a GIN convolutional layer over all sibling spans to recognize the execution patterns of all functions called by the same parent in order to generate an intermediate vector $\mathbf{h}_j$ as parameters for Eq. 2 and 3.

The objective of training the GNN model is to minimize the reconstruction error of duration and error status across all spans. Therefore, the loss function can be expressed as the sum of the mean squared error (MSE) between the predicted and actual duration and the binary cross entropy (BCE) between the predicted and actual error status, which can be shown in Eq. 5.

3.5 Root Cause Analysis with Counterfactuals

Counterfactual is defined as the top level of the Causal Hierarchy for causal reasoning [57]. A counterfactual query asks for the expectation the result Y if X had been a different x, given $X = x'$ and $Y = y'$ in the actual world (i.e. $\mathbb{E}(Y_x \mid x', y')$) [58]. Similar to Sage, Sleuth uses counterfactual queries to locate the root instance of anomaly traces. Counterfactual queries ask for expectations of trace duration and error status in the case that a selected subset of spans is restored to a normal state (with duration equal to the median and without errors). First, it aggregates spans by `service` and `kind`. If the span is a client span, it is affiliated with the services of that span and all its child services; otherwise, it is only affiliated with the services of that span. The reason we differentiate between client spans is that certain types of failures located in child services, for example, network failures, can directly affect the state of the parent span without passing through its own spans. It sorts the services in descending order by the sum of exclusion errors and the amount of extra exclusion duration compared to the normal state of all their affiliated spans. Then, Sleuth locates the root-cause services by iteratively restoring the services to their normal states, predicting the counterfactual trace state, and checking whether the trace is predicted to be normal. The root-cause pods and nodes are where the root-cause services are running and they can be identified easily from span attributes with root-cause services are located.

$$\begin{aligned}
\hat{d'}_i &= \sum_{j\in\mathcal{N}(i)} \text{ClippedReLu}(d'_j,\ u'_j,\ v'_j) + d^{*'}_i \\
&= \sum_{j\in\mathcal{N}(i)} \left[\text{ReLu}\big(d'_j - u'_j\big) - \text{ReLu}\big(d'_j - v'_j\big)\right] + d^{*'}_i, \qquad (2) \\
u'_j &= h'_{j,1} - h'_{j,0}, \quad v'_j = h'_{j,1} + h'_{j,0}
\end{aligned}$$

$$\hat{e}_i = \max_{j\in\mathcal{N}(i)} \left(\text{sigmoid}(h_{j,2}\cdot e_j),\ \text{sigmoid}(h_{j,3}\cdot d_j),\ e^*_i\right), \qquad (3)$$

$$\begin{aligned}
\mathbf{h}_j &= [h_{j,0} \quad h_{j,1} \quad h_{j,2} \quad h_{j,3}] \\
&= f_\Theta\Big[\mathbf{x}^*_i \parallel (1+\epsilon)\cdot\mathbf{x}_j + \sum_{k\in\mathcal{S}(j)} \mathbf{x}_k\Big]. \qquad (4)
\end{aligned}$$

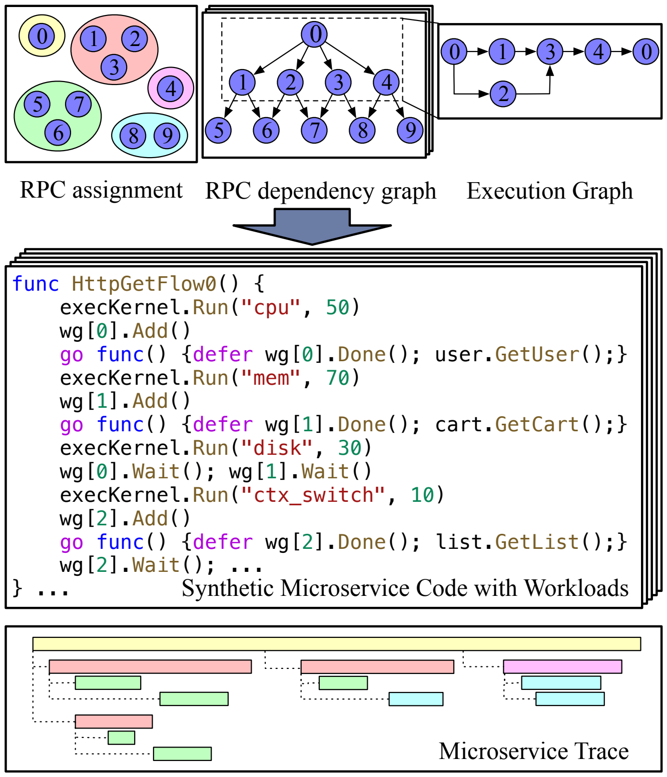
$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N}(\hat{d}_i - d_i)^2 - e_i\log(\hat{e}_i) - (1-e_i)\log\big((1-\hat{e}_i)\big). \qquad (5)$$

4 IMPLEMENTATION

Sleuth comprises several essential components, including trace collectors, a distributed trace storage engine, machine learning (ML) training and inference workers, and a model server.

For each microservices system, we utilize Kubernetes to deploy a cluster of OpenTelemetry collectors. The collectors support different trace protocols, including OpenTelemetry, Zipkin and Jaeger. They are capable of forwarding traces into the storage engine.

The distributed trace storage engine provides support for both storing and querying trace data at a scale of terabytes. The engine allows users to run parallel SQL-like queries with user-defined operators and functions. Therefore, we offload many computational-intensive workloads for data engineering, such as feature engineering, exclusive duration/error calculation, distance metric computation and clustering to SQL operations for better throughput and lower latency for both training and inference.



```
func HttpGetFlow0() {
    execKernel.Run("cpu", 50)
    wg[0].Add()
    go func() {defer wg[0].Done(); user.GetUser();}
    execKernel.Run("mem", 70)
    wg[1].Add()
    go func() {defer wg[1].Done(); cart.GetCart();}
    execKernel.Run("disk", 30)
    wg[0].Wait(); wg[1].Wait()
    execKernel.Run("ctx_switch", 10)
    wg[2].Add()
    go func() {defer wg[2].Done(); list.GetList();}
    wg[2].Wait(); ...
} ...
```

Figure 4: An example of synthetic microservice generation.

The GNN model of Sleuth is implemented with PyTorch Geometric [24]. The models are stored in a centralized object database and they can be pulled and updated by model training and inference workers. A centralized model server maintains the life cycle of the GNN model, including model creation, storage, update, inheritance and retirement.

5 SYNTHETIC MICROSERVICE GENERATION

Using the existing microservice benchmarks [6, 28, 63, 73] to evaluate the accuracy, performance, and scalability of the trace RCA algorithm has limitations since the largest microservice application in those benchmarks only has 41 services, which is not comparable to the typical scale of production microservices [47]. As a system scales, heuristics that have been validated on small-scale systems may gradually fail. This is because heuristics are often based on simplifying assumptions that no longer hold true as the system size increases. Additionally, some statistics- and machine learning-based algorithms have poor scalability and can become computationally intractable for large-scale microservice applications. In this section, we introduce a technique to automatically generate synthetic microservice benchmarks for quantitatively evaluating the accuracy and performance of microservice RCA algorithms and the scale of microservices grows.

An example of the generation process is illustrated in Figure 4. First, given the total number of services and RPCs, the generator randomly assigns a function to a service with a specified distribution. Then, a random DAG generator constructs an RPC dependency graph for each operation flow according to the inputs. If a function has child RPCs to invoke, the generator will also build an execution graph to represent the execution patterns of the child RPCs. Random execution kernels are injected between the RPC invocations to simulate local executions of a function based on an input distribution. After initializing those execution configurations for all microservices, the code is generated with the corresponding RPC calls and local executions, as well as the extra supportive code for cloud-native deployment and observability. The code generator currently supports gRPC [2] microservice framework in Go, Java, and C++ with Kubernetes [38] deployment, but it can be easily extended to other languages, programming frameworks, and deployment tools.

5.1 Initialization of Configurations

A configuration file containing all services, RPCs, RPC dependency graphs, execution graphs, and workload configurations is initialized for generating the microservice code.

5.1.1 RPC and Service Allocation. The service allocator assigns the RPCs to the microservices. It randomly labels a service to be a frontend, middleware, backend, and leaf tier, which has different locations and fanout degrees in the RPC dependency graph. We also collect a set of commonly used service and RPC names and assign them to the synthetic microservices and RPCs to make their semantics in traces more realistic.

5.1.2 RPC Dependency Graph. Since microservice applications typically process different types of requests, we construct an RPC dependency graph for each type of operation flow using a DAG generator. The DAG generator generates RPC dependency graphs based on parameters such as the number of nodes, maximum depth, width distribution of each layer, in-degree and out-degree distribution of each node. It then randomly assigns the nodes in the DAG to specific RPCs based on the match between the node location and the service label where the RPC is located. For example, RPCs in front-end services have a higher chance of being allocated to shallower nodes with more child RPCs, while RPCs in back-end services tend to be allocated to deeper nodes with more inbound connections.

5.1.3 Execution Graph. The execution graph describes the order and dependencies of child RPCs invoked by the same parent RPC. It is a weakly connected DAG starting from the parent RPC. An edge from a parent RPC to a child RPC indicates that the child RPC can be invoked directly by the parent, without waiting for the result of any other child RPCs. An edge between two child RPCs suggests that the destination RPC cannot be invoked until the response of the source RPC is received and processed by the parent.

A synchronous child RPC always has descendant nodes in the execution graph, either its siblings or parent. In contrast, the response of an asynchronous child RPC is not immediately needed by the parent and thus has no descendants in the execution graph. The execution graph is constructed randomly by the DAG generator for each RPC per operation flow, with parameters configuring its depth, width, maximum parallelism, and whether the parent has asynchronous child RPCs.

5.1.4 Local Workloads. We implemented a set of pluggable and configurable microbenchmarks [8, 22] that simulate local workloads between invocations of child RPCs. Those workloads can stress different components of hardware and OS kernel, including CPU core, cache, memory, network, disk, file system, and the OS scheduler, with their duration and intensity controlled by a set of parameters. These microbenchmarks can be inserted at the beginning and end of a function, as well as before and after any child RPC invocations to simulate request and response processing on a server. We generate the local workload configurations by a probability distribution defined in the input.

5.2 Code Generation

The code generator builds microservices automatically according to the configuration file via code templates. We use gRPC for synchronous RPCs and Kafka for asynchronous messages. To support convenient deployment on managed Kubernetes services by major cloud service providers, we implement service discovery and client-side load balancing with Consul [21], distributed tracing with OpenTelemetry [53], system monitoring with Prometheus [5], and contain orchestration with Docker image registry [23] and Kubernetes [38].

6 EVALUATION

6.1 Methodology

6.1.1 Cloud Applications. We use multiple applications to evaluate the performance of Sleuth. Including SocialNetwork in DeathStarBench [28], the SockShop microservice [6] and four synthetic gRPC microservices generated by the approach from Section 5 at different scales. All applications are deployed in an 100-node Kubernetes (K8s) cluster.

- **SockShop** [6]: SockShop is a popular open-source microservice demo application. It contains 11 microservices implemented with Spring Boot [7], go-kit [30], and Node.js [52]. Users can browse an inventory of socks, add them to shopping carts, place an order, and make a payment. We use Locust [45] to generate a mixture of HTTP requests. `POST /orders` is the most complex API with 57 spans in total and a max depth of nine.
- **SocialNetwork** [28]: *SocialNetwork* is an end-to-end microservice in DeathStarBench with similar functionality to Twitter or Facebook to share posts, follow friends, view the followed posts, and browse the homepages of others. It consists of 26 individual services, including Nginx as the web server, tens of Thrift microservices for processing business logic, Memcached and Redis for caching, MongoDB for persistent storage, and RebbitMQ for asynchronous messaging. We load the system with Socfb-Reed98 Facebook social network dataset [60] as the social graph and use wrk2 [1] as the HTTP workload generator. Among all APIs, `ComposePost` has the most complex trace graph, with 31 spans in total and a max depth of nine.
- **Synthetic gRPC microservices**: We generate four synthetic gRPC microservice benchmarks, Synthetic-16, Synthetic-64, Synthetic-256, and Synthetic-1024, using the techniques introduced in Section 5, with each microservice benchmark consisting of 16, 64, 256, and 1024 RPCs, respectively. The statistics of RPCs and topology of RPC dependency graphs follow the distributions characterized in [47] so that the synthetic benchmarks has similar RPC call patters as the production microservices.

Table 1: Specifications of microservice benchmarks.

	Sock Shop	Social Net	Syn. 16	Syn. 64	Syn. 256	Syn. 1024
Services	11	26	4	16	64	256
RPCs	58	61	16	64	256	1024
Max spans	57	31	30	126	510	2046
Max depth	9	9	3	7	15	15
Max out degree	11	7	4	7	14	24

Table 1 presents a comparison of the specifications of all microservice benchmarks. Through the synthetic microservices generated automatically, we can quantitatively compare the accuracy, performance, and scalability of different trace RCA algorithms on microservices whose scale is close to or even beyond that of most current production services [47].

6.1.2 Baselines. We select the two rule-based trace RCA algorithms commonly used by the SREs, three start-of-the-art algorithms proposed from the previous papers as baselines. Those algorithms are based on heuristics, statistical models, and deep learning models. We also compare Sleuth against a different GNN model using GCN as aggregation layers.

- **Max duration**: The maximum duration algorithm discovers the root cause instances of high latency and error traces separately. It aggregates the exclusive duration of all spans in the trace by a certain level, such as span name, service name, pod name, and hostname and selects the instance with the highest exclusive duration as the root cause for high latency traces. In addition, it uses Deep First Search (DFS) in the RPC dependency graph to find instances that have errors not originating from their children as the root cause for error traces.
- **Threshold**: The threshold algorithm is similar to the max duration algorithm except that it sets duration thresholds based on percentiles for all spans to identify high-latency spans in high-latency traces. The instances with high-latency spans are root causes for high-latency traces. It also uses DFS to identify the root cause instances of error traces.
- **TraceAnomaly** [44]: TraceAnomaly is a trace anomaly detection and RCA system which relies on trace data only. It detects anomalies in traces with a variational autoencoder (VAE), identifies anomaly spans with three-sigma rules, and locates the root cause as the longest path with anomaly spans.
- **Realtime RCA** [15]: Cai et al. proposed a real-time microservice RCA system with distributed tracing data. It compares the anomalous trace with the historical normal trace, and if the span exceeds the 95% confidence interval, the span is flagged as anomalous. It calculates the contribution of each span to the variance of the end-to-end latency by using

Table 2: Platform specifications.

	K8s Cluster	GNN Training	RCA Inference
Nodes	100	1	1
Cores	8	96	16
RAM Size	16GB	768GB	32GB
GPU	-	8 * NVIDIA V100	-
Disk	80GB	2TB	120GB

a linear regression model and selects the most significant anomalous span as the origin of the anomaly.

- **Sage** [27]: Sage is an unsupervised microservice performance debugging system with a deep generative model. It constructs a CBN from the RPC dependency graph, trains a graphical variational autoencoder (GVAE) to generate counterfactuals, and uses the predicted counterfactuals to locate the root cause of performance incidents.
- **Sleuth-GCN**: Instead of using the network architecture we proposed in Section 3.4.1, Sleuth-GCN builds the GNN model a vanilla GCN implementation in [37].

To study the impact of the clustering algorithm on performance and accuracy, we also compare our proposed trace clustering algorithm with one start-of-the-art algorithm in the previous research.

- **DeepTraLog** [70]: DeepTraLog is an unsupervised trace clustering system based on GNN and SVDD. It learns the latent representation of a graph with Gated Graph Neural Networks (GGNN) that can be enclosed with a minimum hypersphere.

6.1.3 Platforms. We evaluate Sleuth with a cluster running on a public cloud, with the specifications shown in Table 2. We deploy all benchmark microservices with K8s on an 100-node cluster. The GNN model is trained on a server with 8 NVIDIA V100 GPUs and all trace RCA inferences are executed on a midsize CPU server to reduce the deployment cost.

6.1.4 Chaos Engineering. Although fault injection is not a necessary step in collecting Sleuth training data because it is an unsupervised model, we inject noise during the evaluation process and use the noise injection logs as ground truth to quantify accuracy. We use Chaosblade [16] to inject CPU, network, memory, and disk noises at container-, pod-, and node-level in the K8s cluster. We decide whether to inject noise into each instance (i.e., container, pod, or node) by a vector of independently distributed random variables that follow Bernoulli distributions with distinct small probabilities to simulate real-world failures.

6.1.5 Metrics. We use F_1 score and accuracy (ACC) to measure the accuracy of different algorithms in identifying the root cause instances of an abnormal trace. Each trace RCA query q predicts a set of root cause instances C_q^{pred}. We calculate true positives (TP), false positives (FP), and false negatives (FN) for each query by comparing the predicted set with the actual root cause instance set C_q^{real}. $\text{TP}_q = \{i | \forall i \in C_q^{\text{pred}} \wedge C_q^{\text{real}}\}$, $\text{FP}_q = \{i | \forall i \in C_q^{\text{pred}} \setminus C_q^{\text{real}}\}$, and $\text{FN}_q = \{i | \forall i \in C_q^{\text{real}} \setminus C_q^{\text{pred}}\}$. TP, FP, and FN are aggregated across all RCA queries to calculate F_1 score and accuracy in evaluation.

- **F_1 score**: $F_i = \frac{2|\text{TP}|}{2|\text{TP}|+|\text{FP}|+|\text{FN}|}$, which is a commonly used metric for binary classification that accounts for the impact of both FP and FN.
- **Accuracy (ACC)**: $\text{ACC} = \frac{1}{|Q|} \sum_{q \in Q} \mathbf{1}_{C_q^{\text{pred}} = C_q^{\text{real}}}$, which is a stricter metric than F_1. It requires the algorithm to completely match the predicted result of each query with the actual root cause to achieve a high ACC.

6.2 Root Cause Detection Accuracy

First, we compare the two accuracy metrics, F_1 score and ACC, with other baseline methods for RCA and clustering on five microservice benchmarks. For each microservice application, we randomly sample 144,000 traces from the data collected for 24 hours and split them into training, validation, and testing datasets for the training phase. During the evaluation, we randomly inject all kinds of errors with the methodology described in Section 6.1.4 and collect 100 anomaly samples for the evaluation of the detection accuracy.

Table 3 shows the two accuracy metrics, F_1 score and ACC, of all RCA algorithms and Sleuth with different clustering metrics on five microservice benchmarks. Sleuth without clustering outperforms all other techniques in the two microservice benchmarks and three synthetic microservices.

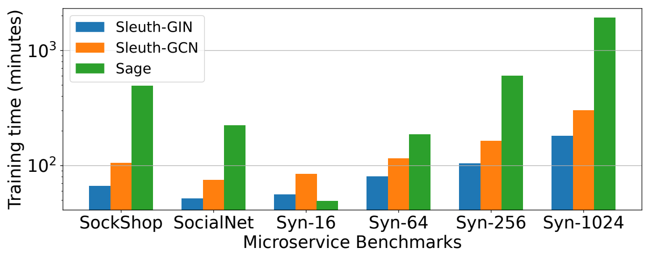
In particular, Sleuth has significantly higher accuracy than all empirical rule-based and correlation-based techniques (i.e. Max, Threshold, TraceAnomaly, and Realtime RCA), outperforming them by 84% - 16.2x in terms of accuracy. There are three major reasons for the higher accuracy of Sleuth. First, the thresholds are determined statically without considering the dynamic impact of spans in different traces. As a simple example, the same span may or may not cause a trace to violate its SLO, depending on the latency of the other spans. Second, it is difficult to find empirical rules or heuristics that can be applied to all spans or services. A parameter suitable for one service may cause false alarms in another. Third, counterfactual-based approaches, namely Sleuth and Sage, are less affected to the network failures described above than other approaches because they can compare the potential outcomes when recovering client and server states and determine which has a high probability of being the root cause. Compared to Seer and the Sleuth variant using GCN, the proposed Sleuth model exploits domain knowledge about latency and error propagation, so it outperforms them in terms of accuracy. Furthermore, by comparing the results for Synthetic-64, Synthetic-256, and Synthetic-1024, we find that the two GNN-based approaches are more robust to the growth of the microservice scale than the other approaches. Unlike other baselines, the GNN model does not rely on a specific causal DAG architecture or on any heuristic parameters that vary with microservice size, which contributes to a more robust accuracy at a large scale.

Clustering algorithms can lead to a reduce the overall accuracy of Sleuth because they introduce extra errors. In general, clustering with our proposed trace distance metric reduces the detection accuracy by 6.1%-9.5%. It has the highest error in the Synthetic-1024 because traces are more close to each other because of a large number of spans, which makes the HDBSCAN algorithm group traces with different root causes into one cluster. With the distances calculated by GGNN-SVDD from DeepTraLog, the clustering algorithm groups many traces located in the center of the hypersphere into

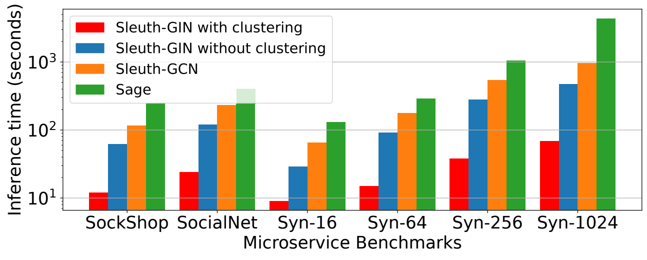
Table 3: F_1 score and accuracy of different RCA algorithms and Sleuth with different clustering metrics on five microservice datasets.

	SockShop		SocialNet		Syn. 64		Syn. 256		Syn. 1024	
	F_1	ACC	F_1	ACC	F_1	ACC	F_1	ACC	F_1	ACC
Max	0.59	0.48	0.62	0.54	0.65	0.54	0.69	0.56	0.60	0.46
Threshold	0.43	0.29	0.59	0.38	0.47	0.31	0.34	0.17	0.26	0.06
TraceAnomaly [44]	0.46	0.34	0.53	0.41	0.55	0.39	0.39	0.21	0.24	0.11
Realtime RCA [15]	0.23	0.15	0.31	0.22	0.30	0.19	0.21	0.13	0.07	0.05
Sage [27]	<u>0.89</u>	<u>0.85</u>	<u>0.91</u>	<u>0.87</u>	0.88	0.79	0.85	0.80	0.74	0.71
Sleuth-GCN	0.66	0.52	0.82	0.73	**0.90**	**0.85**	<u>0.89</u>	<u>0.86</u>	<u>0.86</u>	<u>0.82</u>
Sleuth-GIN										
- DeepTraLog [70]	0.33	0.30	0.46	0.38	0.38	0.30	0.39	0.30	0.27	0.22
- w/ clustering	0.86	0.81	0.88	0.84	0.86	0.78	0.87	0.83	0.85	0.81
- w/o clustering	**0.91**	**0.88**	**0.94**	**0.92**	<u>0.89</u>	<u>0.84</u>	**0.91**	**0.89**	**0.89**	**0.86**

one large cluster and suggests that they have the same root cause, resulting in low detection accuracy.



(a) Training Time in seconds.



(b) Inference time for 1000-trace batch in seconds.

Figure 5: (a). The training time and (b). inference time of Sleuth and Sage as microservice application scales. Both figures are plotted in log scale.

6.3 Performance

We also compare the performance of Sleuth against another deep-learning-based RCA system - Sage, with the same setting described in Section 6.2. Figure 5 shows the training and inference time of Sleuth with GIN, Sleuth with GCN, and Sage when the size of the microservice application scale. The training and inference time of Sleuth-GIN and Sleuth-GCN scale sublinearly with the microservice size, increasing only around 3.3x and 16.3x when the dataset scales from Synthetic-16 to Synthetic-1024. However, the training and inference time of Sage increase by 39.5x and 33.9x, respectively, when the microservice application scales. The difference in scalability between Sleuth and Sage is mainly a result of the model size. While the model size of Sleuth does not change, the GVAE model size increases from 1.3MB to 84MB, which is approximately linear to the application size. The specifically designed GIN outperforms the GCN, with speedups of 1.8x and 1.9x in training and inference, because the network architecture of GIN is simpler. Clustering can significantly improve the inference time from 6.4x on Synthetic-16 to 15.3x on Synthetic-1024 as it reduces the number of traces for ML inference. The speedup is greater on a larger-scale microservice application because the inference is more expensive.

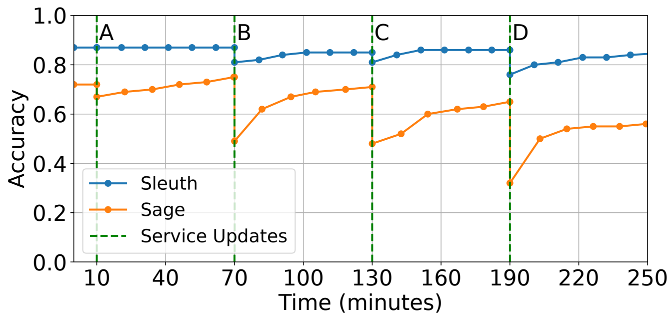


Figure 6: Detection accuracy of Sleuth and Sage on Synthetic-1024. The green dash lines indicate when microservices get updated.

6.4 Service Updates

We compare Sleuth against Sage to examine their real-time detection accuracy when microservices update in the largest Synthetic-1024 benchmark. The updates are rolled out every hour, and after the service is updated, both models start to train every ten minutes when the batch data streams in. The updates include:

- **A**: Increase the average processing time of one microservice on the third level in the RPC dependency graph by 10x.

- **B**: Remove the updated service from the system.
- **C**: Add another service on the second level in the RPC dependency graph.
- **D**: Add three microservice chains, each of which consists of three microservices, in the middle of the RPC dependency graph.

Figure 6 illustrates the detection accuracy between Sleuth and Sage on Synthetic-1024 when microservices get updated. The network architecture of Sage relies on the RPC dependency graph, so we need to dynamically insert or remove neurons in the GVAE in Sage. Since update A only impacts the duration of spans and does not change the RPC dependency, the accuracy of Sage just drops 7%. However, when update D adds nine new services in the middle of the system, the detection accuracy of Sage drops sharply to 0.29 due to too many new neurons added to the GVAE. Sleuth, on the contrary, is less sensitive to service updates because the model can generalize to new services with the prior information learned from the rest part of the system. In addition, since it takes a shorter time to train the GNN than GVAE, Sleuth converges faster than Sage.

6.5 Transfer Learning

Figure 7 demonstrates the ability to transfer a pre-trained Sleuth model to another unseen microservice application. In this experiment, we use two pre-trained Sleuth models: one trained on a synthetic dataset of 256 microservices and the other trained on a larger dataset of 50 real-world microservice applications. We then fine-tune these models and evaluate their root cause detection accuracy on two new benchmarks: SockShop and Synthetic-1024. We compare the results with Sage, which does not generalize to completely different microservice applications, and another Sleuth model trained with the evaluated application from scratch.

The results show that the Sleuth model pre-trained with the Synthetic-256 dataset can infer the root cause of performance degradation in an unseen microservice application with only a few-shot fine-tuning. The accuracy increases immediately after the model is fine-tuned with 1000 samples for just tens of seconds. After fine-tuning the model with 10000 samples for hundreds of seconds, the accuracy is comparable with the detection accuracy of a GNN model trained from scratch for several hours.

The Sleuth model pre-trained on a larger and more diverse dataset can be applied directly to unseen microservice applications without any fine-tuning, at the cost of a 4%-5% loss of accuracy on our evaluation datasets. The detection accuracy also converges much faster and closer to the GNN when using the pre-trained model learning from a larger dataset.

In contrast, the Sage model needs to be retrained from scratch for an unseen microservice application with a different RPC dependency graph. This is because the neurons of the GVAE in Sage cannot be shared across different services and RPCs. Therefore, it is impossible to build a pre-trained model from a large dataset which can be generalized to any microservice.

The ability of few-shot or even zero-shot learning is primarily a result of using the GNN model. The same GNN model in Sleuth is reused to learn the information flow of all nodes, unlike the GVAE in Sage, where separate VAEs are used to predict the outcomes of different nodes. Therefore, Sleuth is capable of learning the knowledge that is generalizable among different microservices, which makes it more adaptable to different microservice applications.

6.6 Sensitivity to Semantic Information in Spans

To study the sensitivity of two pre-trained models introduced in section 6.5 to trace semantic information, we randomly assign service and RPC names in microservices and evaluate the detection accuracy of the models. We duplicate the traces in the testing set into two copies. In the first copy, we keep the service and RPC names as their original. In the second copy, the names are randomly generated from another list that has no intersection with the training set. The accuracy is then evaluated both without fine-tuning and after fine-tuning using 10,000 samples.

We observe that semantic information affects the accuracy of pre-trained models using a single data source, but has little effect on models pre-trained from more diverse data sources. As is shown in Figure 8, misleading semantic information results in up to 19% accuracy drop on Synthetic-1024. Therefore, a model pre-trained with a single dataset can overfit to specific semantic information. However, the difference in accuracy due to incorrect semantic information drops when the model is pre-trained using a larger dataset with traces collected from more diverse microservice applications. We suspect that the model learns to focus on the performance metrics rather than memorize semantic information in traces as the diversity increases in the training set. After fine-tuning with 10,000 trace samples, both pre-trained models overcome the distraction of misleading semantic information and have similar accuracy on the two test replicas.

7 DISCUSSIONS

7.1 Difference between Sleuth and Sage Models

The ML model used in Sleuth is designed differently from the one used in Sage. Sleuth takes into account several important factors, such as the nature of large-scale microservices, the diversity of traces, and zero-shot learning. As a result, Sleuth is more adaptable, performant, and scalable than Sage, and can therefore handle large-scale microservice traces more efficiently.

- **Adaptability:** While Sage was originally designed to support microservices with a relatively consistent topology, real-world microservice applications can contain a large number of trace graphs. The GNN used by Sleuth exhibits better flexibility and generalizability, enabling it to manage various types of RPC graphs using a single model with a moderate number of parameters. As a result, Sleuth is a more versatile tool for managing complex microservice applications with a diverse and dynamic RPC graph topology.
- **Performance:** Sage exhibits significantly prolonged training and inference time in contrast to Sleuth. The network architecture of Sleuth enables direct inference on new datasets via a pre-trained model with few-shot fine-tuning or even zero-shot learning. This feature makes Sleuth more suitable for production deployment, as external users can readily

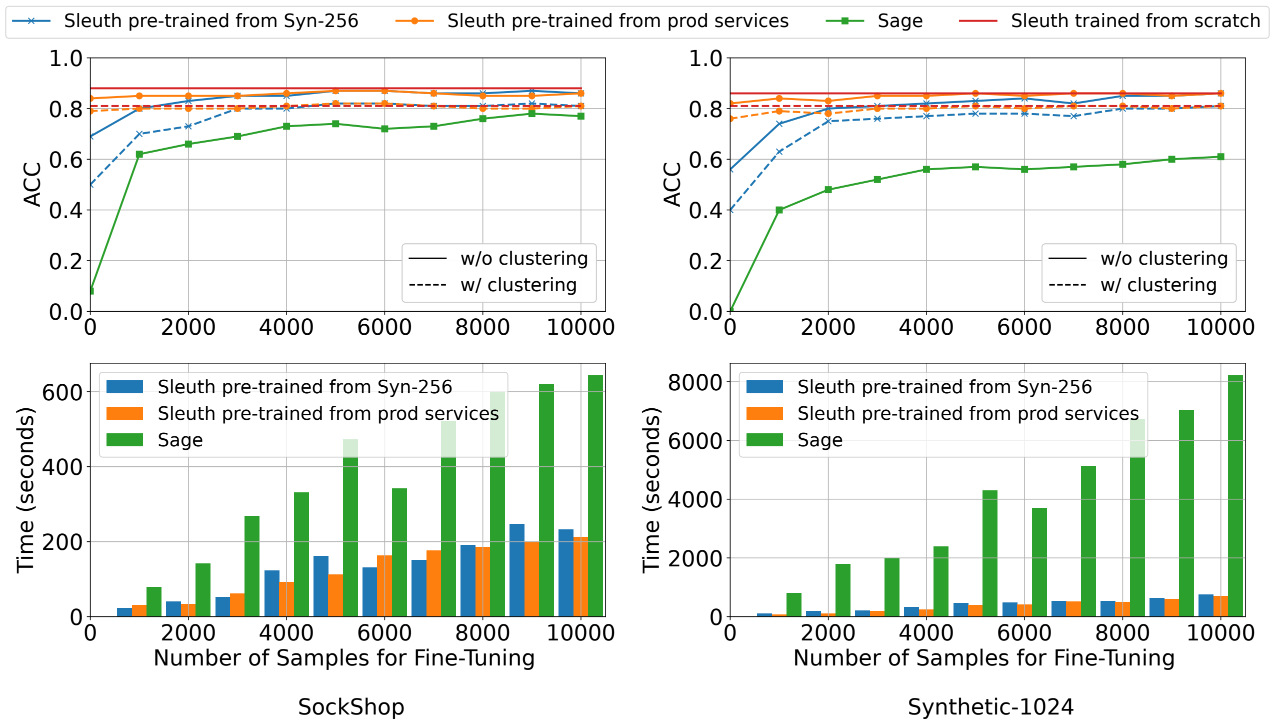


Figure 7: The accuracy and retraining time of the Sleuth model with an increasing number of samples. The Sleuth model is pre-trained from Synthetic-256 and 50 production microservices, and fine-tuned and tested on SockShop and Synthetic-1024. In contrast, the Sage model is retrained from scratch because it is not adaptable to different RPC dependency graphs. The red reference lines shows the accuracy of Sleuth model trained from scratch using the same application.

utilize the pre-trained model while maintaining detection accuracy.

- **Scalability:** The scalability of Sage is hindered by its reliance on a growing number of VAEs as the scale of applications increases. In contrast, Sleuth maintains a fixed neural network architecture that enables better scalability. We observe that the accuracy of Sage drops as the complexity of the applications increases and suspect that it is because of the use of multiple VAEs that accumulate errors as the scale grows. Furthermore, the expansion of the model size in Sage exacerbates its training and inference time.

7.2 Synthetic Microservice Benchmarks for Evaluation

The quantitative evaluation of RCA algorithms using real-world large-scale microservices is ideal, but it presents significant challenges.

First, the development and open-sourcing of production-scale microservices are impeded by intellectual property concerns and the considerable amount of work involved. This makes it difficult to create standardized large-scale microservice benchmarks. Second, anomalous traces in production systems are sometimes not correctly diagnosed by site reliability engineers (SREs), making it difficult to obtain ground truth for quantitative evaluation. Third, the results collected from internal production services may lack reproducibility and comparability, making it difficult to draw meaningful conclusions about the performance of RCA algorithms.

To address these challenges, we propose employing synthetic microservices combined with chaos engineering. This approach offers a standardized benchmark for researchers to assess the accuracy and performance of RCA algorithms across diverse studies. These synthetic microservices simulate real-world application behavior in the presence of common issues in the cloud. By using various execution kernels to stress CPU, memory, disk, and network components, they replicate responses akin to real hardware disruptions. Furthermore, synthetic microservices are built in popular programming languages, RPC and messaging frameworks, and are deployed and operated via Kubernetes, mimicking the setup of actual production services. They communicate based on specified RPC topology graphs characterized from real-world applications.

While we acknowledge that synthetic microservices do not entirely mirror the complexity of production microservices, we believe that their inherent superiority in terms of scale and diversity surpasses the available open-source microservice benchmarks.

7.3 Limitation and Future Work

The causal graph utilized in Sleuth is primarily constructed from microservice traces, without integrating insights from additional application- or system-level logs and metrics. This limitation confines the system's identified root causes to the exact instances (i.e.

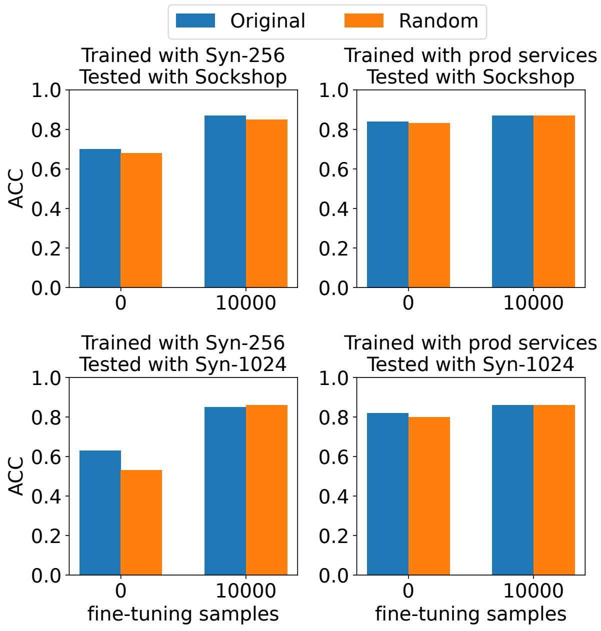


Figure 8: Model detection accuracy on SockShop and Synthetic-1024 with different semantic information using two pre-trained models in section 6.5

nodes, pods, services, spans) linked to anomaly traces. To achieve a more detailed root cause analysis, a broader heterogeneous causal graph at a larger scale, informed by domain expertise from SREs and datacenter operation engineers, might be imperative. While GNNs offer promise in dissecting these diverse causal graphs, additional research is needed to fully investigate this avenue.

8 CONCLUSION

We design Sleuth, an unsupervised root cause analysis system for large-scale traces in microservices. It uses graph neural networks to model the propagation of spans' status and locate the root cause of abnormal traces. Sleuth is also built with a trace distance metric to cluster anomaly traces and reduce inference time. In experiments using existing open-source microservice benchmarks and large-scale synthetic microservices generated according to the characteristics of production microservices, Sleuth achieves high accuracy identifying root causes in anomaly traces with low overheads. Sleuth is flexible and generalizable, making it easy to be applied across microservice applications with few-shot or even zero-shot learning.

ACKNOWLEDGMENTS

We sincerely thank Cheng Zhang, Mingxia Cai, Changjun Xiao, and the anonymous reviewers and shepherd for their support and feedback on earlier versions of this manuscript.

REFERENCES

[1] giltene/wrk2. https://github.com/giltene/wrk2.
[2] grpc: A high performance open-source universal rpc framework. https://grpc.io/.
[3] Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/.
[4] Opentracing. https://opentracing.io/.
[5] Prometheus. https://prometheus.io/.
[6] Sockshop: A microservices demo application. https://www.weave.works/blog/sock-shop-microservices-demo-application.
[7] Spring framework. https://spring.io/projects/spring-framework.
[8] stress-ng. https://wiki.ubuntu.com/Kernel/Reference/stress-ng.
[9] Zipkin. http://zipkin.io.
[10] Alibaba cloud container service for kubernetes (ack). https://www.alibabacloud.com/product/kubernetes.
[11] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51. IEEE, 2016.
[12] Amazon elastic kubernetes service (eks). https://aws.amazon.com/eks.
[13] Azure kubernetes service (aks). https://azure.microsoft.com/en-us/products/kubernetes-service/.
[14] Daren C Brabham. *Crowdsourcing*. MIT Press, 2013.
[15] Zhengong Cai, Wei Li, Wanyi Zhu, Lu Liu, and Bowei Yang. A real-time trace-level root-cause diagnosis system in alibaba datacenters. *IEEE Access*, 7:142692–142702, 2019.
[16] chaosblade io. chaosblade-io/chaosblade: An easy to use and powerful chaos engineering experiment toolkit., Sep 2022.
[17] P. Chen, Y. Qi, P. Zheng, and D. Hou. Causeinfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pages 1887–1895, 2014.
[18] Max Chickering, David Heckerman, and Chris Meek. Large-sample learning of bayesian networks is np-hard. *Journal of Machine Learning Research*, 5:1287–1330, 2004.
[19] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 217–231, Berkeley, CA, USA, 2014. USENIX Association.
[20] Common web application architectures. https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures.
[21] Consul. https://www.consul.io/, Sep 2022.
[22] Christina Delimitrou and Christos Kozyrakis. iBench: Quantifying Interference for Datacenter Workloads. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC)*. Portland, OR, September 2013.
[23] Docker. https://www.docker.com/, 2022.
[24] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric, 2019.
[25] Fluentd. Fluentd. https://www.fluentd.org/.
[26] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.
[27] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: Practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 135–151, New York, NY, USA, 2021. Association for Computing Machinery.
[28] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayantara Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.
[29] Yu Gan, Yanqi Zhang, Kelvin Hu, Yuan He, Meghna Pancholi, Dailun Cheng, and Christina Delimitrou. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.
[30] Go kit - a toolkit for microservices. http://gokit.io/, 2014.
[31] Google kubernetes engine (gke). https://cloud.google.com/kubernetes-engine.
[32] GoogleCloudPlatform. Online boutique. https://github.com/GoogleCloudPlatform/microservices-demo.
[33] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 156–166, 2012.
[34] Joop J Hox and Timo M Bechger. An introduction to structural equation modeling. 1998.
[35] Vimalkumar Jeyakumar, Omid Madani, Ali Parandeh, Ashutosh Kulshreshtha, Weifei Zeng, and Navindra Yadav. Explainit! – a declarative root-cause analysis

engine for time series data. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 333–348, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Markus Kalisch and Peter Bühlmann. Estimating high-dimensional directed acyclic graphs with the pc-algorithm. *Journal of Machine Learning Research*, 8(Mar):613–636, 2007.

[37] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[38] Kubernetes. https://kubernetes.io/, Sep 2022.

[39] Grafana Labs. Grafana. https://grafana.com/.

[40] Mingjie Li, Zeyan Li, Kanglin Yin, Xiaohui Nie, Wenchi Zhang, Kaixin Sui, and Dan Pei. Causal inference-based root cause analysis for online service systems with intervention recognition. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '22, page 3230–3240, New York, NY, USA, 2022. Association for Computing Machinery.

[41] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, et al. Practical root cause localization for microservice systems via trace analysis. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, pages 1–10. IEEE, 2021.

[42] JinJin Lin, Pengfei Chen, and Zibin Zheng. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In *International Conference on Service-Oriented Computing*, pages 3–20. Springer, 2018.

[43] Dewei Liu, Chuan He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshun Wu. Microhecl: High-efficient root cause localization in large-scale microservice systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 338–347. IEEE, 2021.

[44] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, et al. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 48–58. IEEE, 2020.

[45] Locust. https://locust.io/, 2022.

[46] Logstash. Logstash. https://www.elastic.co/logstash.

[47] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 412–426, New York, NY, USA, 2021. Association for Computing Machinery.

[48] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, and Chengzhong Xu. An in-depth study of microservice call graph and runtime performance. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3901–3914, 2022.

[49] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 378–393, New York, NY, USA, 2015. Association for Computing Machinery.

[50] Leland McInnes, John Healy, and Steve Astels. hdbscan: Hierarchical density based clustering. *The Journal of Open Source Software*, 2(11), mar 2017.

[51] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016.

[52] Node.js. Node.js. https://nodejs.org/en/, 2022.

[53] Opentelemetry. https://opentelemetry.io/, 2022.

[54] OpenTelemetry. Opentelemetry specification, 2022.

[55] Mateusz Pawlik and Nikolaus Augsten. Efficient computation of the tree edit distance. *ACM Trans. Database Syst.*, 40(1), mar 2015.

[56] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, 2016.

[57] Judea Pearl. *Causality: Models, Reasoning and Inference.* Cambridge University Press, New York, NY, USA, 2nd edition, 2009.

[58] Judea Pearl. Structural counterfactuals: A brief introduction. *Cognitive science*, 37(6):977–985, 2013.

[59] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

[60] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[61] Huasong Shan, Yuan Chen, Haifeng Liu, Yunpeng Zhang, Xiao Xiao, Xiaofeng He, Min Li, and Wei Ding. ?-diagnosis: Unsupervised and real-time diagnosis of small- window long-tail latency in large-scale microservice platforms. In *The World Wide Web Conference*, WWW '19, page 3215–3222, New York, NY, USA, 2019. Association for Computing Machinery.

[62] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[63] A. Sriraman and T. F. Wenisch. μ suite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12, 2018.

[64] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. Sieve: Actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, page 14–27, New York, NY, USA, 2017. Association for Computing Machinery.

[65] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. Workload characterization for microservices. In *Proc. of IISWC*. 2016.

[66] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterfield, and K. Schwan. Statistical techniques for online anomaly detection in data centers. In *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*, pages 385–392, 2011.

[67] Hanzhang Wang, Zhengkai Wu, Huai Jiang, Yichao Huang, Jiamu Wang, Selcuk Kopru, and Tao Xie. Groot: An event-graph-based approach for root cause analysis in industrial settings. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 419–429. IEEE, 2021.

[68] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. Microrca: Root cause localization of performance issues in microservices. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9, 2020.

[69] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[70] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 623–634, New York, NY, USA, 2022. Association for Computing Machinery.

[71] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.

[72] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 149–161, New York, NY, USA, 2018. Association for Computing Machinery.

[73] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2018.

[74] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 683–694, 2019.