SparseRCA: Unsupervised Root Cause Analysis in Sparse Microservice Testing Traces

Zhenhe Yao[1,5], Haowei Ye[2], Changhua Pei[3,6]✉, Guang Cheng[2], Guangpei Wang[2], Zhiwei Liu[2], Hongwei Chen[2], Hang Cui[3,7], Zeyan Li[4], Jianhui Li[3], Gaogang Xie[3], Dan Pei[1,5]
[1]Tsinghua University, yaozh20@mails.tsinghua.edu.cn, peidan@tsinghua.edu.cn
[2]Ant Group, {yehaowei.yhw, chengguang.cg, guangpei.wgp, biao.lzw, wei.chenhw}@antgroup.com
[3]Computer Network Information Center, Chinese Academy of Sciences, {chpei, lijh, xie}@cnic.cn
[4]Bytedance Inc., lizytalk@outlook.com
[5]Beijing National Research Center for Information Science and Technology
[6]Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences
[7]University of Chinese Academy of Science, cuihang24@mails.ucas.ac.cn

***Abstract*—Microservice architecture has become a predominant paradigm in the software industry. This architecture necessitates robust end-to-end testing to ensure seamless integration of all components before deployment. Rapidly pinpointing issues when test cases fail is crucial for enhancing software development efficiency. However, in testing environments, the available trace is often sparse, and the system is continuously upgrading, which renders existing microservice-based root cause analysis (RCA) ineffective. To address these challenges, we propose SparseRCA. By assessing the abnormality of the exclusive latency, SparseRCA directly determines the probability of the root cause, solving the challenge of not being able to fully obtain the fault propagation information, such as call relationships in sparse trace scenarios. At the same time, by reconstructing the exclusive latency using the decoupled atomic span units, it solves the problem of latency prediction for new traces caused by frequent upgrades. We evaluate SparseRCA on real-world datasets from a large e-commerce system's testing environment, where it demonstrates significant improvements over existing models. Our findings underscore the effectiveness of SparseRCA in addressing the challenges of RCA in microservice testing environments.**

***Index Terms*—root cause analysis, software testing, microservice traces**

I. Introduction

The microservice architecture, characterized by its design dividing large systems into smaller, self-contained components that communicate with each other, has emerged as a predominant architectural paradigm in the software industry, enhancing scalability and flexibility in large-scale applications [1]–[5].

To maintain the reliability of microservices systems, conducting end-to-end testing in microservices systems during rapid software development iterations is crucial for ensuring all components integrate seamlessly before full deployment [6]. During these tests under microservice architecture, it is a crucial step to perform root cause analysis (RCA) and localize the underlying faulty microservice to improve the accuracy and efficiency of subsequent interventions and maintain system integrity [7]–[10]. Typically, the process of localizing root causes in microservices hinges on the system knowledge possessed by operations engineers or on characteristics automatically captured by RCA algorithms. This knowledge or these characteristics are essential for diagnosing and troubleshooting newly emerged faults.

✉ Changhua Pei (chpei@cnic.cn) is the corresponding author.

Fig. 1: The traces are sparsely and unevenly distributed in testing scenarios.

To achieve efficient root cause localization, software engineers extensively utilize various data sources (such as metrics [11]–[18], logs [19]–[22], traces [6], [23]–[27], or a combination thereof [28]–[31]) to conduct automated, comprehensive incident analysis and fault diagnosis. While locating root causes among faulty microservices, tracing is a widely employed tool to monitor and record the intricate interactions within microservice architectures [6], [7], [27]. A trace corresponding to a single user request includes multiple spans, each recording information of a microservice call (e.g., the response time of each service), as shown in Fig. 2. Compared to other data sources, tracing offers distinct advantages, including providing a highly detailed and contextual visualization of a request's exact path and processing timeline through each microservice. Additionally, tracing facilitates real-time, dynamic analysis and effective fault isolation by delivering an end-to-end view of how requests traverse various distributed components, which is crucial for diagnosing cross-service issues in complex systems where single metrics or logs may not provide a complete narrative. Utilizing traces or metrics extracted from traces, many researchers have proposed numerous effective methods for identifying the root causes in microservice systems.

However, the previous methods mainly focus on performing RCA in production environments instead of the testing environment, which has some distinct characteristics:

TABLE I: Comparison of trace sparsity between production and testing scenarios. The dataset from the testing environment is collected from the datacenter of a large e-commerce company in this study. The traces in testing environments are significantly sparser than those in production environments.

Trace Dataset	Source	Traces	Duration	Services
AIOps-Challenge [32]	Production	2.5m	50 days	22
Alibaba [33]	Production	10b	7 days	20k
current study dataset	Testing	6k	37 days	507

- **Knowledge Obsolescence.** In the testing environment, system changes in microservices are more frequent. Previously accumulated knowledge about system characteristics can easily become outdated, while knowledge of the new system after changes is often not yet established or fully analyzed. Furthermore, one result of these frequent changes in characteristics is the frequent appearance of new system features in the testing environment that have never been seen in historical data, such as new and previously unseen trace structures.
- **Trace Sparsity.** In the testing environment, because of the high cost of constructing test cases [6], [34], traces of microservices are significantly sparser than in the production environment. Tab. I illustrates this difference, showing a stark contrast between the density of traces in production versus testing environments. Moreover, while some key time series metrics, such as response success rates and average microservice latency, are generally obtained from aggregating a large number of traces, the sparsity of traces in the testing environment results in two further consequences: (1) First, due to the sparsity of traces, these trace-based aggregated metrics are likely to become discontinuous, featuring many breakpoints and null values. Fig. 1 shows the distribution of traces collected from the testing environment of a large e-commerce company's data center, where traces are sparsely and unevenly distributed, with long periods without any traces, meaning that the mentioned trace-based aggregated metrics cannot be calculated in these parts. (2) Secondly, in the testing environment, the reduced number of observation samples due to trace sparsity increases the volatility of trace-aggregated metrics.

These two characteristics present unique challenges for root cause localization in the testing environment, distinct from those in the production environment:

Challenge 1: The infeasibility of human intervention for RCA in the testing environment. In the testing environment, actively introducing human intervention to assist with root cause localization becomes very difficult. The traditional methods of introducing human knowledge designed for the production environment become infeasible in the testing environment: (1) Manual Annotation: Annotating traces and microservices within a complex microservice architecture with numerous services to perform RCA [6], [25] has always been challenging. This problem becomes even more significant in the testing environment due to more frequent system changes than in a typical production environment. The information from direct manual annotation might quickly become outdated in testing environments, which increases the burden of maintaining an annotated dataset. (2) Integrating Human Knowledge: In the production environment, manually summarizing higher-level knowledge, like setting rules [35] or constructing causality graphs [30], [36], is a common solution. In the testing environment, however, the system characteristics and patterns summarized in the production environment may no longer apply after frequent software changes. Therefore, methods integrating human intervention become much less unreliable in testing scenarios.

Challenge 2: The limitations of traditional RCA methods in sparse trace environments. Previous RCA methods, which rely on aggregating substantial trace data to summarize knowledge automatically, encounter significant obstacles in testing environments due to trace sparsity. On the one hand, the substantial gaps and null segments in aggregated metrics hinder reliance on these metrics for RCA-related tasks, including causality mining and microservice classification. On the other hand, even if the RCA algorithms are successfully trained, the volatility of time-series metrics due to the sparsity of samples severely impacts the RCA algorithms' ability to capture and learn system characteristics and knowledge accurately. For example, a microservice's response success rate metric might remain stable when there are abundant traces, but in scenarios with fewer traces, even one or two failed responses can cause dramatic fluctuations in the success rate metric. This limitation greatly reduces the accuracy of the RCA models.

Challenge 3: The necessity for robust generalization and data efficiency in dynamic testing environments. Frequent changes in testing environments pose higher demands on the generalization abilities and data efficiency of RCA models. In these environments, previously unencountered trace structures may suddenly appear following system updates, subsequently being processed by the RCA algorithms. Conventional methodologies [27], [29]–[31], [37] largely fail to address the emergence of new topological changes or structural modifications within microservices. This demands that RCA algorithms exhibit robust generalization abilities, enabling them not only to elucidate internal patterns among existing microservices but also to effectively perform analysis in zero-shot tasks where they encounter completely novel trace structures.

To address the above challenges, we introduce SparseRCA. This model boasts several distinctive features tailored to tackle the issues previously highlighted. (1) SparseRCA is a trace-based, unsupervised RCA model trained with historical traces that carry microservice invocation topology and response times recorded automatically by monitoring systems, eliminating the need for manual fault annotation and trace classification. (2) SparseRCA analyzes traces at the span granularity, processing and analyzing each microservice's response time recorded by the spans within individual traces instead of relying on a certain volume of trace data flow. It treats the trace spans as atomic units for RCA and measures the anomaly levels of

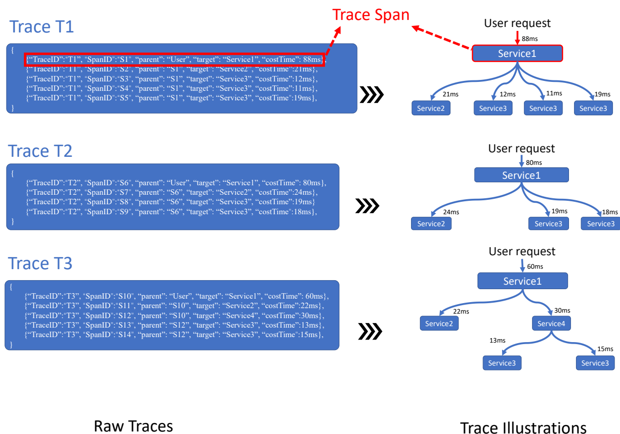
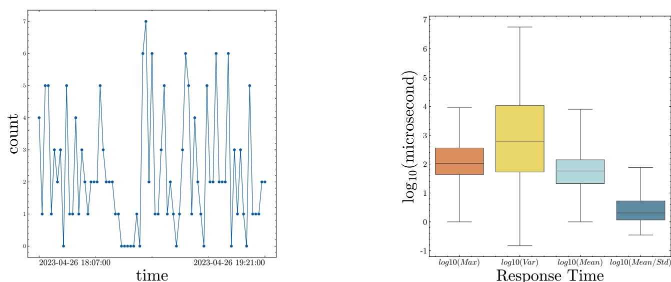


Fig. 2: Examples of traces. Each trace corresponds to a specific user request, consisting of a series of trace spans recording the microservice calls during the request.



(a) Trace Sparsity: number of traces visiting a specific microservice per minute over time our dataset.

(b) Performance Variability: varying response times of microservices across different traces in our dataset.

Fig. 3: Illustration of some challenges for RCA in trace-sparse scenarios from a real-world dataset. During certain periods, the microservice is not accessed, making statistical metrics like error rate or average latency unstable or discontinuous. Additionally, the microservice latencies, particularly the inclusive latencies, vary across different traces.

these spans based on the span-specific latencies rather than relying on statistical information across hundreds of accumulated traces within a time window, addressing the problem of possible sparse trace data flow in testing scenarios. (3) By analyzing the exclusive latency which is closely related to the probability of being root causes [38], SparseRCA investigates consistent performance components across traces varying in terms of trace structure and microservice performance, which effectively minimizes the impact of variability in microservice performance. (4) Furthermore, when encountering entirely new trace structures, SparseRCA adeptly extrapolates the parameters for these new configurations from the most similar existing trace structures. This design ensures reliable root cause analysis, even for trace structures that have not been previously observed.

In summary, this paper makes the following contributions:

- We introduce an unsupervised model, SparseRCA, which performs root cause analysis based on traces in microservice testing environments without needing manual annotation or classification of traces.
- SparseRCA performs root cause analysis at the span granularity, enabling effective root cause localization in testing scenarios where the traces are sparse.
- SparseRCA investigates the exclusive latency components of microservices, thereby enhancing robustness as performance metrics and call structures evolve dynamically. Notably, the model demonstrates a sophisticated capability to estimate exclusive latency when confronted with previously unencountered microservice call structures. Furthermore, it incorporates topological information to enhance the precision of root cause localization.
- Evaluation on datasets collected from the testing environment of a large e-commerce system shows that SparseRCA outperforms the baselines, achieving a 4.9%-46.8% improvement in top-1 accuracy and a 15.1%-48.6% improvement in top-5 accuracy. Ablation studies confirm the effectiveness of model components.

II. Background

This section introduces the architecture and key concepts, formulates the problem of trace-based RCA, and discusses the related works.

A. Architecture and Key Concepts

1) Microservice Architecture: **Microservice architecture** is an architectural style that structures an application as hundreds or thousands of loosely coupled functional units called **microservices**. The microservices are highly maintainable and testable, independently deployable, and usually managed by different small teams. This architecture is widely utilized in large web companies like Twitter [4], Microsoft [38], and Alibaba [33], [39].

2) Software Changes: Software changes refer to modifications made to a software system, including bug fixes, enhancements to existing features, or adding new features. Such changes can potentially alter the behavior of microservice interactions, such as introducing new calls unseen in history, making models trained with data from history before these changes are generally less reliable [40], [41].

3) Testing Environment: In software development, there are Development Environment (DE), Testing Environment (TE), and Production Environment(PE). Integration testing and acceptance testing are usually conducted in TE, where requests are fed into the software to test whether Service Level Objectives (SLOs) are satisfied end-to-end. For example, if the user authentication microservice is updated, testing in TE is needed to ensure this microservice works seamlessly with all other microservices in an end-to-end manner.

4) Traces, Spans, and Contexts: In microservice systems, a **trace** represents the end-to-end journey of a **user request** as it propagates through various services and components in a system. A trace is composed of multiple **spans**, each

representing a single microservice call within a trace. As illustrated in Fig. 2, each span includes the start and end time of the call and other metadata, such as the callee microservice, tag, and parent ID, providing detailed insight into the specific segment of the request's journey. We refer to the callee microservice as the corresponding microservice to the span.

A span records its parent span ID, facilitating the reconstruction of the trace topology. In this paper, given a specific span in a trace, we refer to the list of all ancestor microservice nodes and the microservice of the span as the **context of the span**. Please note that while a service might be called for multiple times, we distinguish these calls by different spans although they share the same callee microservice.

5) Inclusive and Exclusive Latency: The latency of a span refers to the time taken for a call to a microservice to be processed. There are two types of latency measurements:

- **Inclusive Latency (InL)**: The total time taken to complete a microservice call, including the time spent in child calls. It provides an overall measurement of the call. InL is usually directly collected by comparing the start and end timestamps of a microservice call.
- **Exclusive Latency (ExL)**: The time taken to complete a microservice call, excluding the time spent in child calls. It measures the execution time of a microservice and helps identify performance bottlenecks [27], [38].

B. Root Cause Analysis on Traces

In microservices systems, there are many Service Level Objective (SLO) violations [42]. These violations include end-to-end performance and reliability violations, such as high user latency or service unavailability, and infrastructure metric violations, such as continuously increasing memory usage and abnormally frequent garbage collection. When SLO violations are identified, the monitoring system reports the corresponding traces as abnormal, triggering the Root Cause Analysis (RCA) process to identify the microservice most likely responsible for the violation. This is a common practice as in previous studies [38], [43]–[45]. Trace-based RCA uses abnormal traces as input, considering all microservice nodes within a trace as potential root cause candidates. The process then outputs a ranking of the suspected microservices, enabling operators to perform targeted remediation and reducing the time needed for SLO recovery [17].

While a microservice might be called multiple times in a trace and, therefore, recorded in multiple spans, the highest root cause score of these spans is usually taken as the root cause score for this microservice. This service-level score is then used in the service root cause ranking or performance bottleneck analysis [27].

III. Related Works

It has been a long-standing and popular research topic to perform RCA within software microservice architectures, utilizing various data sources which include metrics [11]–[18], logs [19]–[22], [50], traces [6], [23]–[27], and a combination thereof [28]–[31]. Among these, traces have emerged as a particularly important and advantageous method due to their detailed and sequential nature, capturing the flow of requests through services. Some metric-based methods also rely on metrics aggregated from traces, such as error rates and average latency [17].

TABLE II: Related works on root cause analysis and their disadvantages under testing scenarios. *HI* indicates the need for direct human inspection of raw data. *HE* indicates the need for human expertise in configurations and rules. *TT* indicates that the model requires continuous stable traces during training. *TI* indicates that the model requires multiple traces during inference. *UT* indicates that the model cannot process untrained trace topology. Checkmarks represent the models that have these **disadvantages**.

Works	Challenge 1		Challenge 2		Challenge 3
	HI	*HE*	*TT*	*TI*	*UT*
Diagnosing [25]	✓	✗	✗	✗	✓
Manual [6]	✓	✗	✗	✗	✓
PDiagnose [35]	✗	✓	✗	✓	✗
Groot [30]	✗	✓	✗	✗	✓
KGroot [36]	✗	✓	✗	✗	✓
CoE [37]	✓	✗	✗	✗	✓
TraceDiag [38]	✓	✗	✗	✗	✗
MicroRCA [15]	✗	✗	✓	✗	✓
AutoMap [28]	✗	✗	✓	✗	✓
TraceRCA [46]	✗	✗	✓	✓	✓
AlertRCA [46]	✗	✗	✓	✓	✗
CIRCA [47]	✗	✗	✓	✓	✓
RCD [48]	✗	✗	✓	✓	✗
MicroHECL [16]	✗	✗	✓	✓	✗
tprof [27]	✗	✗	✓	✗	✓
Dejavu [49]	✓	✗	✓	✓	✓
CMDiagnostor [17]	✗	✗	✓	✓	✓
Nezha [31]	✗	✗	✓	✓	✓
Eadro [29]	✗	✗	✓	✓	✗
Microscope [24]	✗	✗	✗	✗	✓
MicroRank [23]	✗	✗	✗	✓	✗
SparseRCA (ours)	✗	✗	✗	✗	✗

Several prior studies have proposed trace-based root cause localization methods that require human intervention. Some approaches [6], [25] rely on manual classification and labeling of traces before training the models. PDiagnose [35] transforms original multi-modal data into multiple time series features and uses manually designed threshold-based rules to detect and diagnose issues. Some other works [30], [36] rely on an event-causal graph constructed with manual configurations for event-level root cause inference. TraceArk [51] involves human engineers for anomaly evaluation to optimize alert generation strategies. TraceDiag [38] adopts reinforcement learning to train a decision-maker using labeled root cause information, constructing and pruning a service dependency graph that is subsequently used for root cause localization. However, the frequent system updates and trace sparsity challenge the effectiveness of these traditional RCA methods, which often require substantial trace data and stable system characteristics that are not always available in rapid testing cycles.

Another category of RCA methods uses statistical metrics,

such as average latency per minute and error rate per minute, extracted or aggregated from a continuous trace data flow to localize root causes. Some of these methods use predefined rules or correlation mining algorithms (e.g., Pearson correlation) to construct a weighted causality graph representing fault propagation probabilities, which is then used for root cause inference [15], [28], [48]. Some other algorithms directly utilize these statistical metrics for classification and prediction, determining whether microservices are in an anomalous state and providing a comprehensive root cause ranking based on the overall topology and anomaly states of all microservices [16], [17], [24], [47], [49]. Furthermore, Nezha [31] converts multi-modal data into events and computes event patterns based on their co-occurrence ratios, comparing the patterns under normal and fault conditions to indicate root causes. The aforementioned methods generally rely on the assumption that the extracted statistical metrics form a stable and continuous time series for further mathematical operations like correlation, convolution, or ratio calculation. However, when traces are sparse, such as in testing scenarios, these statistical metrics can fluctuate significantly or contain numerous null values (as shown in Fig. 3a), affecting the accuracy and feasibility of these mathematical operations.

Some researchers have focused on topology-based and causality-based RCA algorithms with various search policies. These methods mostly utilize performance indicators from individual traces. Microscope [24] uses derived thresholds (e.g., the 3-σ principle) based on the SLO characteristics (such as latency) of the microservices to tell whether they are anomalous and then identify the root cause microservice from the anomalous subset with the topology information. MicroRank [23] establishes a linear regression model based on end-to-end latency and the number of various microservice calls within a trace, setting end-to-end latency thresholds for anomaly classification and using spectral analysis to rank root cause microservices. However, both methods rely on the assumption that the inclusive latency of the same microservice remains consistent across different traces and invocations, which actually contradicts the dynamic nature of microservice inclusive latency in microservice environments [27], [38], [51], as shown in Fig. 3b. Some other researchers propose tprof [27] which decomposes spans into subspans based on the starting and ending timestamps of child spans, and perform RCA by analyzing the subspan durations. This method assumes that the order of a microservice calling its child microservices remains consistent from the past to the future, which might not be true in evolving microservices. Furthormore, when encountering previously unseen call structures, tprof cannot perform calculations due to the lack of corresponding historical subspans. For example, tprof fails to estimate the corresponding subspan duration expectations if a new trace includes ”Service A calling B, C, and D” while historically only ”Service A calling B and C” and ”Service A calling B and D” exist.

Inspired by tprof [27], we designed a trace-based, data-efficient root cause localization algorithm named SparseRCA. Similar to the third and fourth layers in tprof, our SparseRCA

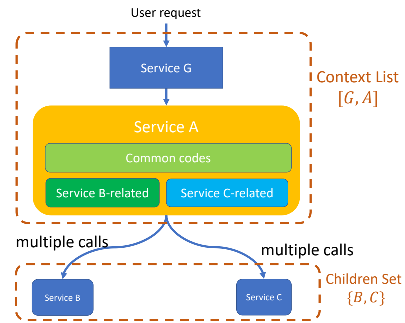


Fig. 4: An example illustrating the ExL components. The context of Service A is a list from the initial service to itself, and the children set is the set of microservices called by Service A. The exclusive latency of Service A (the block in orange) is contributed by (1) the common components (in light green) and (2) the components related to the call number of child microservices (in dark green and light blue).

also decomposes trace span durations based on the relationship between spans and related children spans. However, SparseRCA makes several improvements: (1) SparseRCA decomposes exclusive latency instead of inclusive latency to ensure robustness and does not restrict the order of child span invocation timestamps; (2) we propose a method in SparseRCA to estimate the span ExL for unseen call structures; (3) a personalized PageRank is employed to incorporate topological information to refine the performance-based RCA.

IV. Empirical Study

While the degree of abnormality in ExLs is often closely related to the probability of a microservice being the root cause [38], the ExL of spans is usually influenced by the control flow of a microservice, including the upstream relevance (context) and downstream relevance (child set). Empirical studies are conducted to verify these relevances.

To understand **whether the contexts influence the ExL of the spans**, we analyze the ExL and the different context categories of each call. We conducted ANOVA tests [52] on calls with different contexts to examine the relevance between the context categories of a certain same call and the span latencies, with the null hypothesis that the context categories do not affect the ExL of the same call. The results show that more than 44.8% of the calls have p-values below 0.05, rejecting the null hypothesis and indicating that the latencies of these calls are significantly related to their context. We define the spans with the same context as context-aware spans.

Similarly, to further verify **whether the child set influences the ExL of the spans**, we conducted relevance tests. To eliminate the previously confirmed influence by context, we perform the relevance tests across all context-aware spans with different downstream call numbers. In addition to the ANOVA test, we employed the Kruskal-Wallis H test [53] to provide a more robust analysis in scenarios where sample sizes are smaller and ANOVA's variance of a certain category is not

reliable. The results show that 59.6% (ANOVA) and 63.4% (Kruskal-Wallis) of the context-aware span ExLs reject the null hypothesis with a p-value lower than 0.05, demonstrating relevance with the number of downstream calls.

Therefore, our design of SparseRCA is inspired by the following insights:

- **Insight 1:** Both the context and children set should be considered to accurately model the ExL of spans.
- **Insight 2:** When a microservice performs a task, some components of the ExL are **related** to the number of downstream calls. This overhead may come from executing pre-processing codes, information synchronization codes, and post-processing codes after the calls to child microservices.
- **Insight 3:** When a microservice performs a task, some components of the ExL are **unrelated** to the number of downstream calls. This overhead may come from code blocks unrelated to calling child microservices or from those code blocks always executed as common pre-processing and post-processing, regardless of the number of calls to child microservices.

Please note that **Insight 2** and **Insight 3** are general assumptions that apply to both asynchronous and synchronous architecture. An asynchronous architecture will only change the portion of downstream-call-number-related components in the ExL. We assume that the asynchrony of a specific call between two microservices remains essentially unchanged (i.e., there will not be both synchronous and asynchronous calls between microservice A and microservice B during the same period).

V. DESIGN

Based on the insights from previous empirical studies, this section presents the details of the SparseRCA model. We begin with an overview of the workflow. Next, we introduce the key components of the workflow in detail.

A. Overview

SparseRCA is an unsupervised model designed for trace-based root cause analysis. It leverages the normal trace data to learn the distribution of ExL. The workflow is in Fig. 5.

The training and testing workflow for SparseRCA begins with preprocessing the traces. This step maps each span of a specific trace to three observation metrics: (1) the ExL derived from the InLs of the spans, (2) the pattern of the span, and (3) the number of calls to child microservices.

During both the training and root cause inference stages, SparseRCA uses the pattern parameters and the number of child calls in each span to estimate the ExL distribution of each span. While the span pattern set is constructed based on the training set, a span of the testing trace may not match any existing pattern. To address this, we designed a pattern prediction module that predicts pattern parameters for these spans based on similar recorded patterns.

In the training stage, the estimated ExL distribution is compared with the ExL observations to update the parameters. In the root cause inference stage, this estimated ExL distribution is compared with the ExL observations to generate the ExL anomaly score for each span. This score is then fed into a personalized PageRank algorithm to produce the final service RCA scores.

B. Preprocessing

As the first step of both the training and root cause inference stages, the SparseRCA takes the traces as input and outputs the span pattern-based features.

In this step, while each span records its direct parent and corresponding microservice, the process involves three extractions for a span S_i, which include:

1) The span ExL, $\mathrm{ET}(S_i)$, is extracted by calculating the difference between the InL of the span and the non-overlapping InLs of its child spans.
2) The pattern of span, $P(S_i)$, is extracted by calculating the span context and the set of child microservices.
3) The child calls are categorized by the targeted child microservices and counted. The call number array is denoted as $\mathbf{N}(S_i)$

Following the **Insight 1** in Section IV, We define the **pattern of a span** as the combination of its context and children set, indicating both the upstream (context list) and downstream (children set) microservice control flow.

Different spans within the same span pattern have the same context list and children set. Empirically, their ExL distribution changes with the number of downstream calls similarly because they execute the same code blocks and share similar latency-contributing components (like queueing and network delay). If the observed ExL of a specific span does not follow this distribution, the span might be executing an abnormal branch, experiencing unexpected abortion, or encountering unwanted delays, which means this span is suspicious to be the root cause of SLO violation.

Based on the previous discussion, we designed the preprocessing module to extract the three types of measurements from the training of testing traces. For example, if a span records the InL of microservice B as 15ms, with the context as $A \to B$, and there are 2 child calls from microservice B to microservice D and 5 child calls from B to E which take 5ms in total, we extract the ExL of the span as 10ms, the pattern of this span as $([A, B], \{D, E\})$, and the number of child calls as $[2, 5]$ to microservice D and E.

C. Span Exclusive Latency Modeling in Training

The Span ExL Modeling serves as the second step of the training stage, as shown in Fig. 5.

Based on the **Insight 2** and **Insight 3** in Section IV, we model the ExL of a microservice as two components: the ExL influenced by or uninfluenced by the number of calls to child microservices, which can be expressed as:

$$\mathrm{ET}(S_i) = R(\theta(P(S_i))) + \mathbf{C}(\theta(P(S_i))) \cdot \mathbf{N}(S_i) \qquad (1)$$

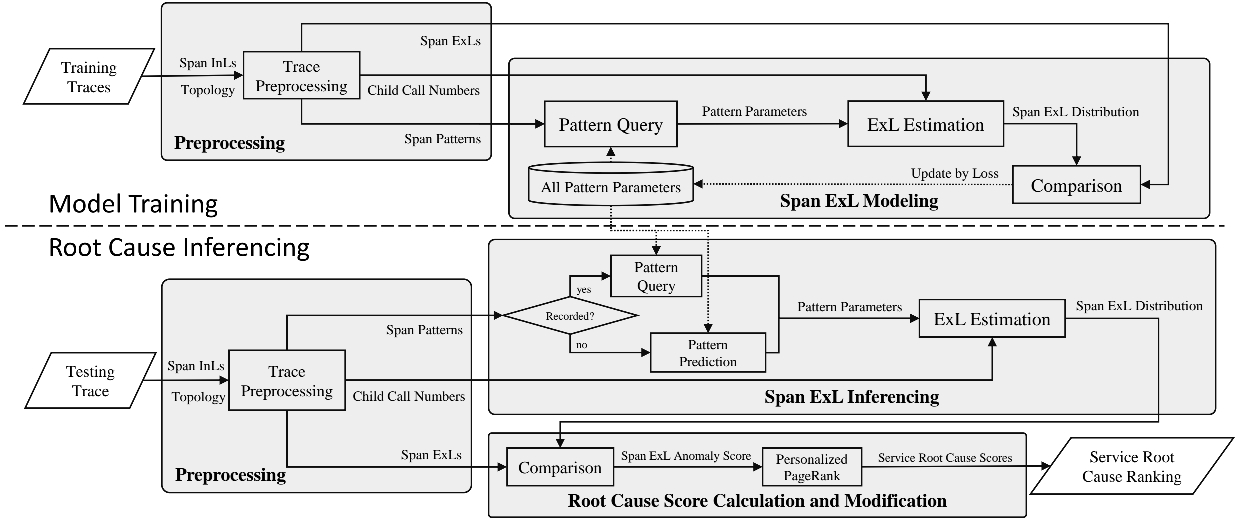


Fig. 5: The overview of components in SparseRCA

where $\theta(P(S_i))$ is the pattern parameters of a specific span pattern, R is the distribution of latency component unrelated to the number of calls to child microservices, and C is the distribution of latency component increasing with $\mathbf{N}(S_i)$, the number of calls to child microservices.

Empirically, we assume the noises in R and $\mathbf{C}$ both follow the Gaussian distribution, then:

$$R \sim \mathcal{N}(t_R(P(S_i))), \sigma_R^2(P(S_i))) \quad (2)$$

$$\mathbf{C} \sim \mathcal{N}(\mathbf{t_C}(P(S_i)), \boldsymbol{\sigma}^2_{\mathbf{C}}(P(S_i))) \quad (3)$$

where t_R, σ_R^2, $\mathbf{t}_C$ and σ_C^2 are the Gaussian parameters collectively referred to as the previously introduced $\theta(P(S_i))$.

To learn a distribution, a popular solution is to utilize Variational Autoencoder (VAE) [54]. However, in our practice, the trace sparsity in the testing scenario makes it difficult to train a reliable VAE model. The varying density of the observations under each span pattern often makes it overfit or non-convergence. To learn the $\theta(P(S_i))$ in SparseRCA, the Gaussian expectation can be easily obtained using the least squares method [55], while the variance can be approximately obtained using the EM algorithm [56].

D. Span Exclusive Latency Inferencing in Testing

During the root cause inference stage, the traces are preprocessed in the same manner as in the training stage. Subsequently, the ExL distributions of the spans are estimated.

1) Recorded Span Patterns: For the span patterns in the testing trace, if a pattern exists in the training traces, we can directly derive the ExL distribution through Eq. (1) with the pattern parameters obtained during the training stage.

2) Unrecorded Span Patterns: There are scenarios where a testing span pattern does not match any patterns in the record. To address this, we have designed a pattern prediction module to estimate the parameters for unseen span patterns.

The pattern prediction is illustrated in Fig. 6. The algorithm is in Alg. 1. In this module, we first identify recorded span patterns related to the testing span pattern (line 2). Related patterns refer to recorded span patterns that (1) share the same microservice (as the orange block in Fig. 6) and (2) share any child microservices (e.g., P_1 and P_3 both share the child microservice C). When the testing pattern has no child microservices, we only keep the first constraint. After identifying the related recorded patterns, we calculate the similarity of the patterns to the testing pattern by the reciprocals of the Levenshtein edit distances and derive the filtered patterns, $\mathbf{P}_f$, with the distance-based weights, ω_f (line 3 - 4). Finally, we use these threshold-filtered Levenshtein edit distances as weights to estimate the parameters of the testing pattern by weighted average (line 5). In practice, the filtering percentile parameter q in Alg. 1 is set as five by experience, which means we filter out the top-5% most dissimilar patterns if there are two or more patterns that can be referenced for a specific parameter.

Algorithm 1: Unseen Pattern Prediction

```
1 def EstimatePattern(P_new, P_rec, q):
      Input  : P_new, the new pattern; P_rec, the set
               of recorded patterns; q, filtering
               percentile parameter.
      Output : the pattern parameters θ(P_new)
2     P_related ← FilterRelated(P_new, P_rec);
3     ω ← SimilarityByEditDistance(P_new, P_related);
4     P_f, ω_f ← FilterByWeights(P_new, P_related, ω, q);
5     θ(P_new) ← WeightedMean(θ(P_f), ω_f);
6     return θ(P_new);
```

After the pattern prediction, the span ExL distribution is derived from Eq. (1), utilizing the predicted parameters and the number of calls extracted during the preprocessing step.

E. Root Cause Score Calculation and Modification

In the last step of the root cause inference stage, the root cause scores of the spans and services are derived.

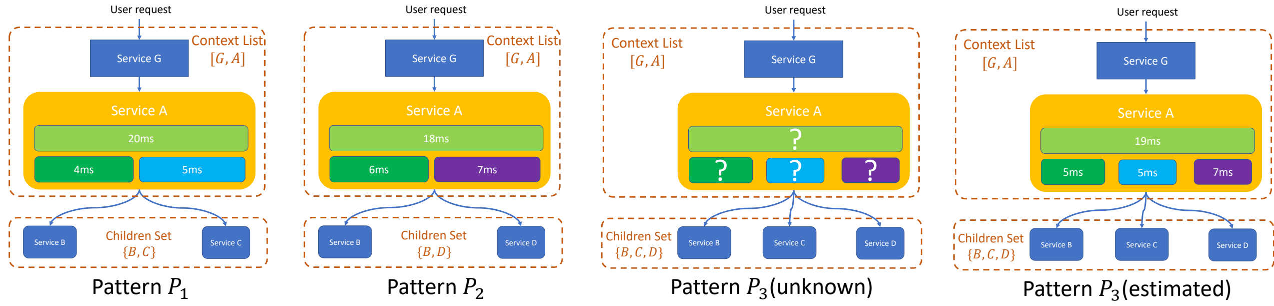


Fig. 6: Example of estimating parameters for an unseen pattern. We take the expectation parameters as an example. Here, patterns P_1 and P_2 are trained on historical data, while P_3 is a newly discovered pattern. The distribution parameters of P_1 and P_2 are used to estimate the distribution parameters for P_3.

1) Span ExL Anomaly Score: According to the Eq. 1, the ExL of a specific span S_i follows a Gaussian distribution with the given $N(S_i)$. The expectation and variation of this distribution are denoted as $\mu(S_i)$ and $\sigma^2(S_i)$, respectively. The ExL anomaly score, $\mathrm{Y}_{raw}(S_i)$, is then calculated as the product of standardized deviation and relative deviation:

$$\mathrm{Y}_{raw}(S_i) = \frac{(\mathrm{ET}(S_i) - \mu(S_i))^2}{(\mu(S_i) + \epsilon) * (\sigma(S_i) + \epsilon)} \tag{4}$$

where ϵ is a smoothing factor. An ablation study is conducted to justify the design of the ExL anomaly score as in Eq. (4) over other metric (e.g., relative deviation only).

2) Personlized PageRank: As performance abnormalities might propagate from the downstream root cause microservice to the parent microservices, making the root cause microservices not necessarily the microservices with the highest ExL abnormality [38]. To address this, we designed a topology-based modification step to optimize the root cause scores. **The goal of this step is to reallocate a portion of the root cause scores from ancestor nodes back to their child nodes.** This adjustment reduces the influence of performance anomalies propagating from ground truth root cause microservices to their parents and ancestors. For the whole span set $\mathbf{S}$ in a trace, we denote the modified and final root cause score array as $\mathbf{Y}_{mod}(\mathbf{S})$ and $\mathbf{Y}_{final}(\mathbf{S})$, respectively:

$$\mathbf{Y}_{mod}^{(t+1)}(\mathbf{S}) = \alpha \mathbf{Y}_{raw}(\mathbf{S}) + (1-\alpha)\mathbf{M} \times \mathbf{Y}_{mod}^{(t)}(\mathbf{S}) \tag{5}$$

$$\mathbf{Y}_{mod}^{(0)}(\mathbf{S}) = \mathbf{Y}_{raw}(\mathbf{S}) \tag{6}$$

$$\mathbf{Y}_{final}(\mathbf{S}) = \mathbf{Y}_{mod}^{(L)}(\mathbf{S}) \tag{7}$$

Here, α is the transition probability, and $\mathbf{M}$ is the normalized directed adjacency matrix of the spans in the trace. By experience, α is assigned as 0.9, and L is 3 since most faulty non-root-cause anomalies are observed to be near this length to the ground truth root cause in our dataset and datasets from other researches [46]. The root cause score of a specific microservice is determined by the highest final span root cause score of the spans corresponding to this microservice.

#Traces (train)	#Traces (test)	Overall (Duration)	#Services	AvgSrv (per trace)	AvgSpn (per trace)
6,080	120	29 days	507	10.6	38.7

TABLE III: Dataset Description. AvgSrv represents average service number while AvgSpn represents average span number.

VI. Evaluation

In this section, we will verify and analyze the performance of SparseRCA on the testing scenario dataset. Our primary research questions are as follows:

- **RQ1: Effectiveness.** How does SparseRCA improve the accuracy of RCA compared to other baseline algorithms on the real-world testing scenario dataset?
- **RQ2: Ablation Study.** What is the individual and combined contribution of some designs in SparseRCA?
- **RQ3: Trace Sparsity Robustness.** How does our model perform in scenarios with even sparser traces?

A. Experiment Design

1) Datasets: We trained and evaluated SparseRCA using a dataset comprising approximately 6 thousand traces collected from the test environment in a data center of a large e-commerce company, as described in Table. III. The dataset comprises traces collected over 29 days and covers around 507 microservices within the test environment of the data center. Each trace records information including the service call topology, service call InL, success status codes, and the physical host IP of the microservices. Among the traces, 120 anomalous traces identified by SLO violation were labeled by experts with root causes and were exclusively used as the test set, while the remaining unlabeled traces in the dataset are the full training set for SparseRCA.

In Section VI-B and Section VI-C, the entire training set was employed to train both complete and incomplete versions of the SparseRCA model. In Section VI-D, the size of the training set was incrementally increased, with only a subset of the training set used for SparseRCA training.

2) Evaluation Metric: Following existing works like [16], [23], we use top-k accuracy to evaluate the RCA accuracy. This metric indicates the probability of the true root cause

being among the top-k recommended options provided by the RCA algorithm. We utilize the random strategy to break the ties. Assuming there are m candidates having root cause scores tied with the ground truth root cause, and the best root cause ranking of these tied candidates is n, then the evaluation metrics are as follows:

$$a_i = \begin{cases} min(m, k-n+1)/m & \text{if } n \le k \\ 0 & \text{if } n > k \end{cases}$$

$$\text{A@k} = \frac{1}{T}\sum_{i=1}^{T} a_i \times 100\%$$

Here, a_i represents the top-k accuracy of the RCA algorithm on the i-th trace, and T is the total number of traces in the test set. A@k is the overall top-k accuracy of the algorithm.

3) Baselines: This section will briefly introduce several *trace-based unsupervised* RCA algorithms selected as baselines for our study.

Microscope [24]. Microscope is a typical causal-graph-based RCA algorithm with specified searching rules, depending on a causal dependency graph to discover candidate nodes and then detect connected abnormal nodes violating the SLO metrics.

MicroHECL [16]. MicroHECL is a typical topology-graph-based RCA algorithm with specified searching rules. It constructs a fault propagation topology based on the interrelationships of metrics and locates the root cause by sequentially detecting the presence of faults in the topology.

AutoMap [28]. AutoMap is a typical causal-graph-based RCA algorithm with a random walk strategy, depending on conditional independence tests of monitoring metrics to determine the causal relationships between microservice nodes.

MicroRank [23]. MicroRank is an RCA algorithm combining end-to-end InL and spectral analysis techniques, which maps the number of calls to various microservices to the end-to-end InL to classify the traces and then performs spectral analysis. While the vanilla MicroRank requires dense traces to perform spectrum analysis, we perform the first step of MicroRank on sub-traces when the traces are not rich enough in a specific period.

Other algorithms that utilize logs or non-performance metrics were not selected as baselines, as they do not conform to the format of our input data [19]–[22]. Additionally, certain algorithms were excluded from our comparison because they necessitate additional expert configuration or manual annotation of the root causes [30], [35], [37]. Other trace-density-dependent methods fail to train on our sparse trace dataset [31], [47]. In detail, Nezha [31] fails to learn the event support for many patterns in its construction and testing stage, and CIRCA [47] fails to access continuous data traffic during structural graph construction and hypothesis testing. Some deep learning models fail to converge in time or produce reliable non-null output [29], [57].

4) Environment: Our experiments run with 32GB RAM and an AMD Ryzen 7 5800H with Radeon Graphics CPU, featuring a base clock speed of 3,201 MHz and 4,096 KB of L2 cache. SparseRCA takes 12 minutes to train on the full trainset and about 0.2 seconds per trace in the inference stage.

TABLE IV: Performance of SparseRCA and Baselines

	Model	Category	A@1 (%)	A@3 (%)	A@5 (%)
Baselines	MicroHECL	Stat-based Topology	19.3	26.4	39.5
	AutoMap	Stat-based Causality	40.7	50.6	61.5
	MicroScope	Stat-based Causality	40.3	66.3	73.0
	MicroRank	Trace InL & Spectrum	61.2	67.6	73.0
Ours	SparseRCA	Trace ExL & Topology	**66.1**	**86.4**	**88.1**

B. RQ1: Effectiveness

The experiments were conducted under the previously introduced settings, and Table IV illustrates the RCA performance of SparseRCA and other baseline algorithms on the test set.

All the models presented in Table IV are unsupervised. Among these, SparseRCA exhibited the highest accuracy, significantly outperforming the baseline algorithms, which include MicroScope [24], MicroHECL [16], AutoMap [28], and MicroRank [23].

Notably, the baseline methods that rely on statistical-metric-based causality discovery or topology elimination showed poorer performance. This result suggests that causality derived from statistics based on sparse trace data is rather inaccurate. As a comparison, models that estimate the performance metrics for a single trace (InL for MicroRank and ExL for SparseRCA) generally achieved better results.

Further, MicroHECL's poorest performance indicates that the service correlation topology and service abnormality classifiers might be learned incorrectly in the testing environment with sparse traces.

C. RQ2:Ablation study

In this section, an ablation study is performed to help analyze the effects of the following components in SparseRCA:

1) The Pattern-Based Modeling of the span ExLs (instead of the call-based), denoted as **PBM**.
2) The Distribution-Based Anomaly score of ExL (instead of expectation-based), denoted as **DBA**.
3) The topology-based Root Cause Modification through personalized PageRank, denoted as **RCM**.

To set up the comparison, removing **PBM** refers to modeling the ExL of all spans of the same call type without distinguishing its context and child set; removing **DBA** refers to utilizing only the expectation to generate the span ExL anomaly score, which replaces the Y_{raw} in Eq. (4) with Y'_{raw} in Eq. (8); removing **RCM** refers to replacing the Y_{final} with the Y_{raw} in Eq. (4).

$$Y'_{raw}(S_i) = \frac{|\text{ET}(S_i) - \mu(S_i)|}{\mu(S_i) + \epsilon} \tag{8}$$

TABLE V: The Performance of Complete and Partial SparseRCA Models

	Model	PBM	DBA	RCM	A@1	A@3	A@5
Complete	SparseRCA	✓	✓	✓	**66.1**	**86.4**	**88.1**
Partial	w/o (RCM)	✓	✓	×	49.2	72.9	72.9
	w/o (DBA)	✓	×	✓	59.3	81.4	84.7
	w/o (PBM)	×	✓	✓	61.0	84.7	88.1
	w/o (DBA,RCM)	✓	×	×	42.4	69.5	72.9
	w/o (PBM,DBA)	×	✓	×	44.1	69.5	74.6
	w/o (PBM,DBA)	×	×	✓	47.5	72.9	84.7
	w/o (PBM,DBA,RCM)	×	×	×	27.1	61.0	72.9

Here, we discuss the results presented in Table. V: It's important to note that whether or not certain steps in SparseRCA are removed, modeling of execution time is performed on the microservices. The experimental results are detailed in Table V. The results suggest that the precision in ranking the top-ranked candidate spans is predominantly shaped by the **RCM**, enhancing mainly top-1 and top-3 accuracy, demonstrating the effective utilization of topology relationships to address inconsistencies arising from the separate calculation of fault scores for each node. Furthermore, the experiment suggests that employing **DBA** to generate the initial root cause score and updating the score through the **PBM** leads to improvements in accuracy across all metrics, including top-1, top-3, and top-5. This is attributed to the **PBM** step consolidating similar call pattern instances, enabling the model to learn from a more extensive pattern dataset and mitigating the limitations associated with sparse incidents. The enhancements achieved by using **DBA** underscore the significant fluctuation characteristics in execution time, aligning with the previous observations.

D. RQ3: Trace Sparsity Robustness

In this section, we examine the robustness of the model under scenarios with sparser traces.

Our investigation centers on SparseRCA's performance with varying scales of training data. We trained the model on different-sized subsets randomly sampled from the training dataset's related traces. These subsets varied in the number of traces. We evaluated the top-k accuracy of the models trained on these training subsets. This process was repeated three times, and for each specific training dataset size, the accuracy was averaged over the three rounds of experiments.

The results are presented in Table VI. For comparison, the best baseline (MicroRank) is also included in the table and figure. Experiments show that SparseRCA achieves better A@1, A@3, and A@5 accuracy than the best baseline utilizing only about 40%, 15%, and 10% of the training traces, respectively. These results suggest that SparseRCA maintains satisfying top-k accuracy even as the trainset size decreases and the traces become sparser.

TABLE VI: Accuracy of SparseRCA Under Sparser Traces.

trainset used (%)	Model	A@1	A@3	A@5
100	MicroRank	61.2	67.6	73.0
100	SparseRCA	66.1	86.4	88.1
50	SparseRCA	66.1	78.0	84.7
40	SparseRCA	66.1	79.7	83.1
25	SparseRCA	55.9	72.9	79.7
20	SparseRCA	59.3	71.2	79.7
15	SparseRCA	54.2	69.5	78.0
10	SparseRCA	54.2	66.1	74.6
5	SparseRCA	42.4	52.5	69.5

VII. Threats to Validity

External Validity SparseRCA has demonstrated promising results in the testing environment of a large e-commerce company's microservice system, suggesting potential applicability across various industrial domains. However, further evaluation is needed to confirm its generalizability to other domains and non-microservice architectures.

Additionally, the root cause labels in our test set, manually annotated by operations engineers, reflect their own practical expertise. Future work should consider incorporating universal labeling standards to enhance the generalizability of the findings across different fields.

Internal Validity In SparseRCA, we assume the ExL of a span increases approximately linearly with the number of children calls. This assumption aligns well with our current data scenario and has shown effective results. However, this model assumption may not capture all the complexities of real-world microservice interactions

Construct Validity In our scenario, the root cause is defined as the specific microservice most responsible for a particular trace being identified as abnormal due to SLO violations. While this definition is robust and appropriate for our analysis and widely adapted in many previous studies [38], [43]–[45], we acknowledge that it may not capture every nuance of root causes in a diverse range of microservice architectures. However, it provides a clear and actionable framework for our current study.

VIII. Conclusion

This paper introduces SparseRCA, a trace-based RCA algorithm that addresses the unique challenges of software end-to-end testing scenarios characterized by frequent system updates and trace sparsity. SparseRCA performs RCA at span granularity by analyzing the performance characteristics of trace spans through exclusive latency decomposition, effectively addressing the limitations posed by discontinuous metrics typically aggregated from traces. Additionally, SparseRCA is equipped to estimate pattern parameters for new trace span patterns that have not been previously encountered, thereby enabling it to infer exclusive latency anomalies with enhanced accuracy.

Experiments conducted on a dataset collected from the testing environment of a large e-commerce company's data center, which comprises over 6,000 traces, have demonstrated that SparseRCA outperforms existing baseline algorithms in terms of top-k accuracy, reflecting its superior capability to adapt to and effectively manage the dynamic nature of modern microservice testing environments.

ACKNOWLEDGMENTS

This work was supported by Alibaba Group through Alibaba Research Intern Program, the National Natural Science Foundation of China under grants 62072264 and 62202445, the Beijing National Research Center for Information Science and Technology (BNRist) key projects, and the State Key Program of the National Natural Science Foundation of China under Grant 62321166652. We also thank other AntGroup employees for their help in this work.

REFERENCES

[1] J. Soldani and A. Brogi, "Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey," *ACM Computing Surveys (CSUR)*, vol. 55, no. 3, pp. 1–39, 2022.
[2] S. Newman, *Building Microservices*. " O'Reilly Media, Inc.", 2021.
[3] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.
[4] E. Musk, "There are 1200 "microservices" server side," Nov. 2022.
[5] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.
[6] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems," *Communications of The Acm*, vol. 59, no. 8, pp. 32–37, Jul. 2016.
[7] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2018.
[8] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017.
[9] R. Pietrantuono, S. Russo, and A. Guerriero, "Testing microservice architectures for operational reliability," *Software Testing, Verification and Reliability*, vol. 30, no. 2, p. e1725, 2020.
[10] M. Waseem, P. Liang, G. Márquez, and A. Di Salle, "Testing microservices architecture-based applications: A systematic mapping study," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2020, pp. 119–128.
[11] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 378–393.
[12] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng, J. Chen, Z. Wang, and H. Qiao, "Unsupervised Anomaly Detection via Variational Auto-Encoder for Seasonal KPIs in Web Applications," in *Proceedings of the 2018 World Wide Web Conference*, ser. WWW '18. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, Apr. 2018, pp. 187–196.
[13] M. Ma, W. Lin, D. Pan, and P. Wang, "MS-rank: Multi-metric and self-adaptive root cause diagnosis for microservice applications," in *2019 IEEE International Conference on Web Services (ICWS)*. IEEE, 2019, pp. 60–67.
[14] Y. Meng, S. Zhang, Y. Sun, R. Zhang, Z. Hu, Y. Zhang, C. Jia, Z. Wang, and D. Pei, "Localizing failure root causes in a microservice through causality inference," in *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. IEEE, 2020, pp. 1–10.
[15] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root Cause Localization of Performance Issues in Microservices," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–9.
[16] D. Liu, C. He, X. Peng, F. Lin, C. Zhang, S. Gong, Z. Li, J. Ou, and Z. 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)*. IEEE, 2021, pp. 338–347.
[17] Q. Yu, C. Pei, B. Hao, M. Li, Z. Li, S. Zhang, X. Lu, R. Wang, J. Li, Z. Wu, and D. Pei, "CMDiagnostor: An Ambiguity-Aware Root Cause Localization Approach Based on Call Metric Data," in *Proceedings of the ACM Web Conference 2023*, ser. WWW '23. New York, NY, USA: Association for Computing Machinery, Apr. 2023, pp. 2937–2947.
[18] L. Wu, J. Tordsson, J. Bogatinovski, E. Elmroth, and O. Kao, "MicroDiag: Fine-grained Performance Diagnosis for Microservice Systems," in *2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)*, May 2021, pp. 31–36.
[19] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. Sosp '03. New York, NY, USA: Association for Computing Machinery, 2003, pp. 74–89.
[20] H. Zawawy, K. Kontogiannis, and J. Mylopoulos, "Log filtering and interpretation for root cause analysis," in *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–5.
[21] V. Nair, A. Raul, S. Khanduja, V. Bahirwani, Q. Shao, S. Sellamanickam, S. Keerthi, S. Herbert, and S. Dhulipalla, "Learning a hierarchical monitoring system for detecting and diagnosing service issues," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 2029–2038.
[22] S. Lu, B. Rao, X. Wei, B. Tak, L. Wang, and L. Wang, "Log-based abnormal task detection and root cause analysis for spark," in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 389–396.
[23] G. Yu, P. Chen, H. Chen, Z. Guan, Z. Huang, L. Jing, T. Weng, X. Sun, and X. Li, "MicroRank: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments," in *Proceedings of the Web Conference 2021*, ser. WWW '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 3087–3098.
[24] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *International Conference on Service-Oriented Computing*. Springer, 2018, pp. 3–20.
[25] R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.
[26] L. Meng, F. Ji, Y. Sun, and T. Wang, "Detecting anomalies in microservices with execution trace comparison," *Future Generation Computer Systems*, vol. 116, pp. 291–301, 2021.
[27] L. Huang and T. Zhu, "Tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 76–91.
[28] M. Ma, J. Xu, Y. Wang, P. Chen, Z. Zhang, and P. Wang, "Automap: Diagnose your microservice-based web applications automatically," in *Proceedings of the Web Conference 2020*, 2020, pp. 246–258.
[29] C. Lee, T. Yang, Z. Chen, Y. Su, and M. R. Lyu, "Eadro: An End-to-End Troubleshooting Framework for Microservices on Multi-Source Data," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 1750–1762.
[30] W. Hanzhang, W. Zhengkai, J. Huai, H. Yichao, W. Jiamu, K. Selcuk, and X. Tao, "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)*, 2021, pp. 419–429.
[31] G. Yu, P. Chen, Y. Li, H. Chen, X. Li, and Z. Zheng, "Nezha: Interpretable Fine-Grained Root Causes Analysis for Microservices on Multi-modal Observability Data," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 553–565.
[32] NetManAIOps, "AIOps challenge 2020 data," 2023.
[33] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 412–426.
[34] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino

et al., “An orchestrated survey of methodologies for automated software test case generation,” *Journal of systems and software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[35] C. Hou, T. Jia, Y. Wu, Y. Li, and J. Han, “Diagnosing Performance Issues in Microservices with Heterogeneous Data Source,” in *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, 2021, pp. 493–500.

[36] T. Wang, G. Qi, and T. Wu, “KGroot: Enhancing root cause analysis through knowledge graphs and graph convolutional neural networks,” 2024.

[37] Z. Yao, C. Pei, W. Chen, H. Wang, L. Su, H. Jiang, Z. Xie, X. Nie, and D. Pei, “Chain-of-event: Interpretable root cause analysis for microservices through automatically learning weighted event causal graph,” in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion ’24)*. New York, NY, USA: ACM, 2024, p. 12.

[38] R. Ding, C. Zhang, L. Wang, Y. Xu, M. Ma, X. Wu, M. Zhang, Q. Chen, X. Gao, X. Gao, H. Fan, S. Rajmohan, Q. Lin, and D. Zhang, “TraceDiag: Adaptive, Interpretable, and Efficient Root Cause Analysis on Large-Scale Microservice Systems,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1762–1773.

[39] K. Wang, Y. Li, C. Wang, T. Jia, K. Chow, Y. Wen, Y. Dou, G. Xu, C. Hou, J. Yao, and L. Zhang, “Characterizing job microarchitectural profiles at scale: Dataset and analysis,” in *Proceedings of the 51st International Conference on Parallel Processing*, ser. Icpp ’22. New York, NY, USA: Association for Computing Machinery, 2023.

[40] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, “Learning under Concept Drift: A Review,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 12, pp. 2346–2363, Dec. 2019.

[41] Q. Xiang, L. Zi, X. Cong, and Y. Wang, “Concept Drift Adaptation Methods under the Deep Learning Framework: A Literature Review,” *Applied Sciences*, vol. 13, no. 11, p. 6515, Jan. 2023.

[42] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. ” O’Reilly Media, Inc.”, 2016.

[43] R. Xin, P. Chen, and Z. Zhao, “CausalRCA: Causal inference based precise fine-grained root cause localization for microservice applications,” *Journal of Systems and Software*, vol. 203, p. 111724, Sep. 2023.

[44] R. Rouf, M. Rasolroveicy, M. Litoiu, S. Nagar, P. Mohapatra, P. Gupta, and I. Watts, “InstantOps: A joint approach to system failure prediction and root cause identification in microserivces cloud-native applications,” in *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, ser. Icpe ’24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 119–129.

[45] Y. Gan, G. Liu, X. Zhang, Q. Zhou, J. Wu, and J. Jiang, “Sleuth: A trace-based root cause analysis system for large-scale microservices with graph neural networks,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ser. Asplos ’23. New York, NY, USA: Association for Computing Machinery, 2024, pp. 324–337.

[46] Z. Li, J. Chen, R. Jiao, N. Zhao, Z. Wang, S. Zhang, Y. Wu, L. Jiang, L. Yan, Z. Wang, Z. Chen, W. Zhang, X. Nie, K. Sui, and D. Pei, “Practical Root Cause Localization for Microservice Systems via Trace Analysis,” in *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, 2021, pp. 1–10.

[47] M. Li, Z. Li, K. Yin, X. Nie, W. Zhang, K. Sui, and D. 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*, ser. KDD ’22. New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 3230–3240.

[48] A. Ikram, S. Chakraborty, S. Mitra, S. Saini, S. Bagchi, and M. Kocaoglu, “Root Cause Analysis of Failures in Microservices through Causal Discovery,” in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 31 158–31 170.

[49] Z. Li, N. Zhao, M. Li, X. Lu, L. Wang, D. Chang, X. Nie, L. Cao, W. Zhang, K. Sui, Y. Wang, X. Du, G. Duan, and D. Pei, “Actionable and interpretable fault localization for recurring failures in online service systems,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 996–1008.

[50] S. Zhang, J. Zhu, B. Hao, Y. Sun, X. Nie, J. Zhu, X. Liu, X. Li, Y. Ma, and D. Pei, “Fault Diagnosis for Test Alarms in Microservices through Multi-source Data,” in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, Jul. 2024, pp. 115–125. [Online]. Available: https://dl.acm.org/doi/10.1145/3663529.3663833

[51] Z. Zeng, Y. Zhang, Y. Xu, M. Ma, B. Qiao, W. Zou, Q. Chen, M. Zhang, X. Zhang, H. Zhang, X. Gao, H. Fan, S. Rajmohan, Q. Lin, and D. Zhang, “TraceArk: Towards Actionable Performance Anomaly Alerting for Online Service Systems,” in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, May 2023, pp. 258–269.

[52] R. A. Fisher, “Statistical methods for research workers,” in *Breakthroughs in Statistics: Methodology and Distribution*. Springer, 1970, pp. 66–70.

[53] W. H. Kruskal and W. A. Wallis, “Use of ranks in one-criterion variance analysis,” *Journal of the American Statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.

[54] D. P. Kingma and M. Welling, “Auto-Encoding Variational Bayes,” Dec. 2022.

[55] G. A. Seber and A. J. Lee, *Linear Regression Analysis*. John Wiley & Sons, 2012.

[56] A. P. Dempster, N. M. Laird, and D. B. Rubin, “Maximum likelihood from incomplete data via the EM algorithm,” *Journal of the royal statistical society: series B (methodological)*, vol. 39, no. 1, pp. 1–22, 1977.

[57] Z. Xie, C. Pei, W. Li, H. Jiang, L. Su, J. Li, G. Xie, and D. Pei, “From point-wise to group-wise: A fast and accurate microservice trace anomaly detection approach,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. Esec/Fse 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1739–1749.