TraStrainer: Adaptive Sampling for Distributed Traces with System Runtime State

HAIYU HUANG, Sun Yat-sen University, China
XIAOYU ZHANG, Huawei, China
PENGFEI CHEN, ZILONG HE, ZHIMING CHEN, GUANGBA YU, and HONGYANG CHEN, Sun Yat-sen University, China
CHEN SUN, Huawei, China

Distributed tracing has been widely adopted in many microservice systems and plays an important role in monitoring and analyzing the system. However, trace data often come in large volumes, incurring substantial computational and storage costs. To reduce the quantity of traces, trace sampling has become a prominent topic of discussion, and several methods have been proposed in prior work. To attain higher-quality sampling outcomes, biased sampling has gained more attention compared to random sampling. Previous biased sampling methods primarily considered the importance of traces based on diversity, aiming to sample more edge-case traces and fewer common-case traces. However, we contend that relying solely on trace diversity for sampling is insufficient, system runtime state is another crucial factor that needs to be considered, especially in cases of system failures. In this study, we introduce *TraStrainer*, an online sampler that takes into account both system runtime state and trace diversity. *TraStrainer* employs an interpretable and automated encoding method to represent traces as vectors. Simultaneously, it adaptively determines sampling preferences by analyzing system runtime metrics. When sampling, it combines the results of system-bias and diversity-bias through a dynamic voting mechanism. Experimental results demonstrate that *TraStrainer* can achieve higher quality sampling results and significantly improve the performance of downstream root cause analysis (RCA) tasks. It has led to an average increase of 32.63% in Top-1 RCA accuracy compared to four baselines in two datasets.

CCS Concepts: • **Software and its engineering → Cloud computing**; **Software reliability**; **Software performance**.

Additional Key Words and Phrases: distributed tracing, biased sampling, microservice

ACM Reference Format:
Haiyu Huang, Xiaoyu Zhang, Pengfei Chen, Zilong He, Zhiming Chen, Guangba Yu, Hongyang Chen, and Chen Sun. 2024. TraStrainer: Adaptive Sampling for Distributed Traces with System Runtime State. *Proc. ACM Softw. Eng.* 1, FSE, Article 22 (July 2024), 21 pages. https://doi.org/10.1145/3643748

1 INTRODUCTION

Industrial microservice systems operate in a highly unpredictable and dynamic setting [46, 53]. To effectively monitor and manage these systems, distributed tracing [37] has been widely adopted. By implementing frameworks like OpenTelemetry [30] and SkyWalking [38], end-to-end paths of requests through service instances can be recorded as trace data. These distributed traces offer

Authors' addresses: Haiyu Huang, huanghy95@mail2.sysu.edu.cn, Sun Yat-sen University, Guangzhou, China; Xiaoyu Zhang, zhangxiaoyu87@huawei.com, Huawei, Shenzhen, China; Pengfei Chen, chenpf7@mail.sysu.edu.cn; Zilong He, hezlong@mail2.sysu.edu.cn; Zhiming Chen, chenzhm37@mail2.sysu.edu.cn; Guangba Yu, yugb5@mail2.sysu.edu.cn; Hongyang Chen, chenhy95@mail2.sysu.edu.cn, Sun Yat-sen University, Guangzhou, China; Chen Sun, sunchen48@huawei.com, Huawei, Beijing, China.

ACM 2994-970X/2024/7-ART22
https://doi.org/10.1145/3643748

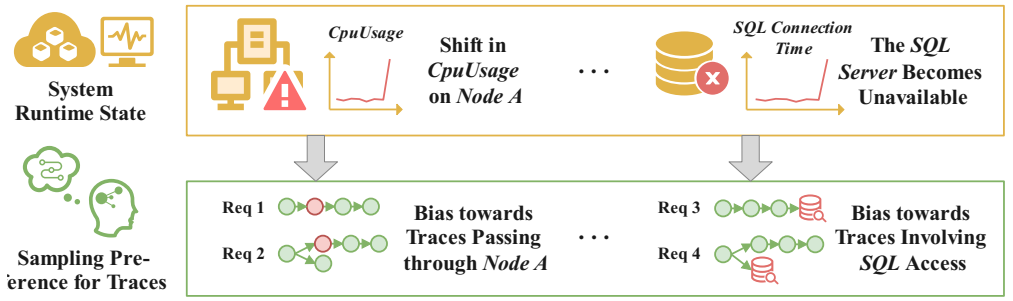


Fig. 1. The relationship between system runtime state and trace sampling preference.

valuable insights for assessing risks, troubleshooting, and comprehending the system's intricacies [7, 39]. Site Reliability Engineers (SREs) extensively leverage them to profile the environment [14], detect anomalies [25, 27, 51], and diagnose failures [10, 26, 35, 45].

Although distributed traces are helpful for analysis, their quantities and storage costs are significant [21]. Performing subsequent analysis on the complete set of traces is also computationally expensive [16]. To address this, trace sampling techniques [11, 16, 20, 21, 37, 52] have been introduced to selectively capture traces of interest.

A common sampling approach used by tracing systems like Jaeger [17] and Zipkin [2] is uniform random sampling [21]. This method determines whether a trace should be sampled at the beginning of the trace, thus is also known as head-based sampling. However, different traces have varying analytical value within the system, and head-based sampling cannot preserve more valuable traces [16]. To address this, tail-based sampling has been proposed and adopted in previous works [11, 16, 20, 21]. This method makes sampling decisions at the end of the request, allowing for the capture of complete trace information to determine if it is valuable. Thus it is also known as biased trace sampling.

However, previous biased sampling methods [11, 16, 20, 21, 52] have not fully considered sampling preferences, especially neglect the influence of system runtime state. These methods were only based on a core intuition: setting sampling preferences based on the specificity and diversity of traces, tending to keep more edge-case traces and fewer common-case traces. This viewpoint makes sense because edge-case traces are more interesting and informative [21]. However, common-case traces are also worth analyzing, as demonstrated in §2.2. Moreover, methods solely based on trace diversity exhibit the same sampling preference (i.e., favoring rare traces) at any given point in system operation, disregarding the crucial context of system runtime state. In fact, the characteristics of a "valuable trace" also change when the system state changes, as shown in Fig. 1. For example, when there is an exception in the SQL server, SREs are more interested in the traces that access database because they are more likely to reflect issues and are more worth analyzing.

***TraStrainer* Approach.** To select more valuable traces while reducing storage and computational costs, we propose an online biased trace sampling approach called *TraStrainer*. The core idea behind *TraStrainer* is to take into account both system runtime state and trace diversity, adaptively providing a more comprehensive sampling preference. *TraStrainer* simultaneously takes system metrics and traces as inputs. *Trace Encoder* (§ 4.1) automatically encodes the coming trace into vector representations based on the system metrics. The generated vector representations contain both structural and state information of the trace, with each system metric corresponding to a specific dimension in the vector. *System Bias Extractor* (§ 4.2) determines the valuable dimensions based on the fluctuations of the system metrics and obtains a preference vector. *System-Biased Sampler* (§ 4.3) and *Diversity-Biased Sampler* (§ 4.4) online calculate sampling probability from two

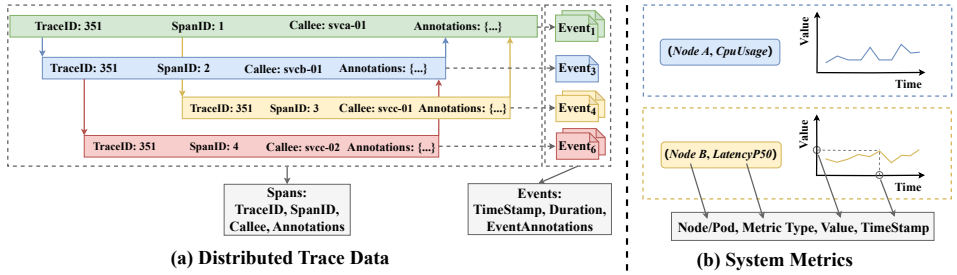


Fig. 2. Trace data and system metrics.

different perspectives. Finally, *Composite Sampler* (§ 4.5) utilizes a dynamic voting mechanism to make the final sampling decision.

To evaluate the effectiveness and efficiency of *TraStrainer*, we constructed two datasets, one from 13 real-world production microservice systems and the other from two widely-used microservices benchmarks, OnlineBoutique [12] and TrainTicket [9]. We conducted a comprehensive evaluation of *TraStrainer*, including the quality of sampling results, performance in downstream tasks, and sampling efficiency. The experimental results demonstrated that compared to four baseline methods, *TraStrainer* was able to identify more valuable traces within the same budget and improve the performance of downstream analysis. Furthermore, *TraStrainer* exhibited the better efficiency than two online biased sampling methods.

Contributions. In summary, this study makes the following contributions.

- We present *TraStrainer*, an online biased trace sampler. By considering both system runtime state and trace diversity, *TraStrainer* offers a more comprehensive approach to adaptively determine sampling preferences.
- We propose an interpretable and automated method for trace representation. The generated trace vector includes both the structural and state information of the trace, with each system metric corresponding to a specific dimension in the vector.
- We combined different sampling methods with several classical trace-based analysis approaches to investigate the impact of different sampling methods on downstream root cause analysis tasks.
- We have implemented *TraStrainer* and constructed two datasets to validate the quality of its sampled data and its effectiveness in improving downstream analysis tasks. Moreover, the experimental results show that our method outperforms state-of-the-art sampling methods.

2 BACKGROUND AND MOTIVATION

2.1 Background

Distributed traces. A trace corresponds to a series of actions triggered by a request within the system. It consists of a series of service operations (spans), and span events [32] have been incorporated in some tracing frameworks [30] to enhance the information associated with each span, as shown in Fig. 2(a). The structure of a trace reflects the order and hierarchy of the invocations among services. The status information such as invocation latency and event annotations on each span also provide crucial insights for subsequent analysis [15]. Despite the high analytical value of traces, the computational and storage overheads are also significant concerns [20].

Trace sampling. Large-scale industrial microservice systems generate millions to billions of traces daily [18, 37]. Analyzing and storing all traces incurs significant costs, making it impractical to meet low-latency requirements [3, 33, 37, 55]. Additionally, not all traces hold valuable insights for analysis [21]. Hence, sampling traces to filter out those with lower analytical value becomes a crucial aspect of operations in microservice systems. The current sampling methods for traces can

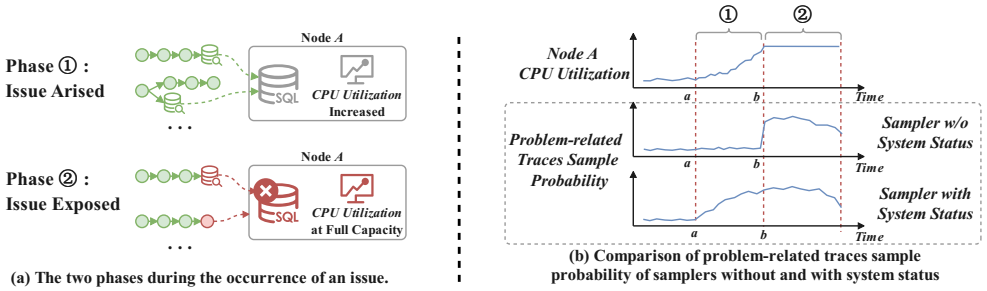


Fig. 3. An example of the importance of system runtime state in trace sampling preferences setting.

be broadly categorized into two types: head-based sampling [2, 17, 37] and biased sampling [11, 16, 20, 21, 52]. As mentioned in § 1, head-based sampling often results in lower quality traces being retained. Therefore, our focus lies on biased sampling.

System metrics. The system metrics are a series of time-series data that reflect the runtime state of the system [19]. They typically consist of four components: timestamp, node or pod affiliation, metric type, and value, as shown in Fig.2(b). During production, due to inevitable reliability and performance issues, the system state undergoes real-time changes [44], and these changes are often manifested by fluctuations in the system metrics [24]. For example, during an overload attack, Node *A*'s disk becomes paralyzed, which is manifested by a sharp increase in the *DiskUsage* metric on Node *A*. By analyzing the fluctuations of the metrics, we can infer the runtime state of the system.

2.2 Motivation

As mentioned in § 2.1, distributed traces are extensively helpful in profiling, diagnosing, and debugging [10, 26, 35, 45]. Therefore, the quality of the sampled traces largely determines the effectiveness of downstream analysis [47]. Prior to this, several biased samplers [11, 16, 20, 21, 52] were proposed to improve the quality of sampled traces. However, we found that the previous biased sampling approaches did not generate truly high-quality traces for downstream analysis, as demonstrated in our experimental findings (§ 5.5). After analyzing the experimental results, we concluded that this was primarily due to insufficient settings in the sampling preferences of the previous approaches. These approaches solely considered trace value based on diversity, favoring edge-case traces over common-cases.

Solely favoring edge-case traces is not sufficient. This is because: (1) Some common-case traces can be related to root causes. For instance, if there is an issue where a thread pool becomes exhausted, the cause may be excessive requests simultaneously issued to the thread pool. The traces associated to this issue may not exhibit any anomalies and are considered common-case traces. (2) Common-case traces are also helpful in downstream analysis methods. For example, trace-based root cause analysis algorithms [26, 45] typically need to collect a certain number of common-case traces to learn the normal invocation patterns and then identify and locate root causes based on this normal pattern. Some downstream approaches [23, 43] require common-case traces to conduct spectrum analysis [34]. (3) When encountering a large number of edge-case traces, it is necessary to determine which ones are more significant. In actual production scenarios, when system failures happen, there are often numerous abnormal traces [45]. If the anomaly rate exceeds the budget sampling rate, it becomes impossible to capture all the edge-case traces. In such scenarios, more specific rules are required to capture the more valuable portions of these traces.

System runtime state plays an important role in determining valuable traces. As a real-world issue case of Huawei is shown in Fig.3: ① During the time window [a, b], there was an increase in concurrent requests to the *MySQL* Server on Node *A*, including some full table queries,

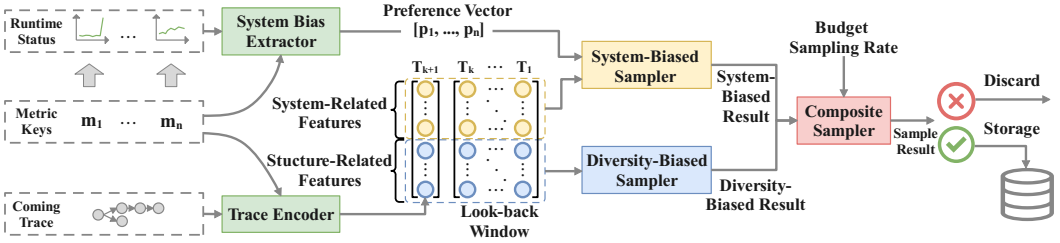


Fig. 4. An overview of *TraStrainer*.

which led to an increase in *CPU utilization* on Node *A*. ② After time b, the *CPU utilization* reached full capacity, causing Node *A* to be unavailable. As a result, all requests passing through Node *A* exhibited abnormal behavior, generating a large number of abnormal traces.

The manual analysis process conducted by SREs for this failure is as follows: When receiving a system alert at time b, SREs examined the system state over a recent period and discovered the increased *CPU utilization* on Node *A* during the time window [a, b]. They further filtered and analyzed the traces passing through Node *A*, eventually identifying the root cause as an overload attack on the *MySQL* Server during the [a, b] time period.

For previous samplers that only considered trace diversity [16, 21], the structure and status information of the traces passing through Node *A* during the [a, b] time window actually appeared normal. Since these traces were considered common-cases, their sampling probability remained low during this time period, resulting in insufficient traces available for analysis and potentially missing the traces related to the root cause event that manual analysis can be aware of. This scenario shows major shortcomings of existing state-of-the-art samplers. It motivates us to consider the system state to dynamically increase the sampling probability of traces related to issues within the time window [a, b], based on the fluctuation of *CPU utilization* in Node *A*.

Goal. Given the discussion above, our goal is to implement biased trace sampling in a more comprehensive way, which considers not only the trace diversity, but also the system runtime state.

2.3 Problem Formulation

We define the problem of biased trace sampling with system runtime state as follows. Given the current system state metrics denoted as $\mathcal{M}$ and the traces collected over a period of time represented by $\mathcal{T}$, we aim to determine a sampling function $Sp(\beta, \mathcal{M}, \mathcal{T}, t)$ for a given budget sampling rate β. This function maintains an average rate of β and calculates a biased sampling probability ρ for each trace t,

$$Sp(\beta, \mathcal{M}, \mathcal{T}, t) \rightarrow \rho, \mathcal{T}'. \tag{1}$$

Subsequently, the decision to retain trace t is based on the sampling probability.

3 OVERVIEW

This section presents the design of *TraStrainer*, an adaptive online biased trace sampler. The objective of *TraStrainer* is to capture higher-quality traces based on system runtime state and trace diversity, and sample them with a higher probability. The overall architecture of *TraStrainer* is illustrated in Fig. 4. *TraStrainer* consists of two main phases: runtime data preprocessing and comprehensive sampling. In the runtime data preprocessing phase, for each coming trace, *Trace Encoder* (§ 4.1) automatically encodes it in an interpretable manner guided by metrics. The generated vector representation includes both structural and status information of the trace, with each system metric corresponding to a specific dimension in the vector. Simultaneously, *System Bias Extractor* (§ 4.2) dynamically calculates the anomaly degree of each system metric. This degree serves as the

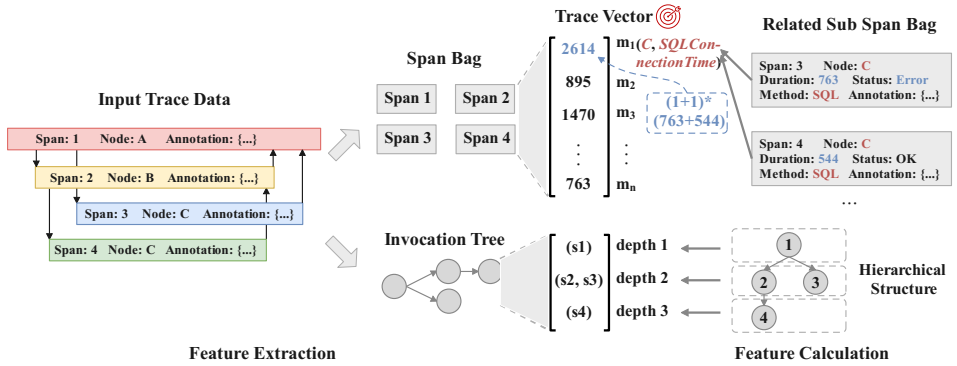


Fig. 5. The process of encoding traces in two parts: status-related and structure-related.

preference weight for the corresponding metric dimension. The larger the anomaly degree, the higher the preference weight. The preference weights for each dimension form a preference vector that determines which dimensions are more valuable.

During the online comprehensive sampling phase, *TraStrainer* takes into account the current coming trace vector and a look-back window of previous trace vectors. It makes a sampling decision that considers both the system bias and the diversity of the traces. On one hand, *System-Biased Sampler* (§ 4.3) calculates the sampling probability for the coming trace based on the preference vector obtained from *System Bias Extractor*, resulting in a system-biased sampling outcome. On the other hand, *Diversity-Biased Sampler* (§ 4.4) determines the diversity-biased sampling outcome by considering the structural and state differences between the coming trace and the previous traces. Finally, *Composite Sampler* (§ 4.5) uses a dynamic voting mechanism to make the final sampling decision based on the sampling budget and the current sampling frequency.

4 DETAILED DESIGN

4.1 Trace Encoder

Trace is not originally machine-readable and requires encoding into a machine-friendly format (e.g., vector) for next-step analysis (e.g., clustering, sampling) [16]. However, previous approaches suffered from insufficient manual intervention and a lack of interpretability. In order to address these issues, *Trace Encoder* proposes an automated trace encoding method guided by system metrics, which ensures that the dimensions of the resulting trace vector are interpretable.

As mentioned in § 2.1, a trace is a tree-like topology structure that contains valuable information in both its status and structure. Therefore, we can divide the encoding of a trace into two parts: status-related and structure-related, as shown in Fig. 5.

Encode the status-related part of the trace. When disregarding the structure of a trace, we can perceive it as a bag of spans, denoted as $S = \{s_1, ..., s_n\}$. Each span, denoted as s, represents a unit of work or operation within the trace. It includes essential information such as duration, status code, the associated node or pod, as well as additional details provided by event annotations. The information appended by event annotations can be complex and diverse. Previous approaches [16, 26] mostly relied on manual selection of relevant information based on human experience. However, this manual approach is costly and requires redefining rules when new fields are introduced. We leverage system metrics to automatically determine the features we need.

Definition 1 (Related sub span bag S_m). *For a metric $m(m.node, m.type)$ and a trace t, let $S = \{s_1, ..., s_n\}$ to be the span bag of t. We use $S_m = \{s_{m1}, ..., s_{mn}\}$ to denote the related sub span bag of m, where s_{mi} must satisfy that $s_{mi}.node = m.node$ and $m.type$ related to $s_{mi}.annotation$.*

Definition 2 (Feature value f_m). *For the related sub span bag $S_m = \{s_{m1}, ..., s_{mn}\}$ of the metric m, we use $S_a = \{s_{a1}, ..., s_{an}\}$ to denote the abnormal span in S_m, where s_{ai} must satisfy that $s_{ai}.status$*

is abnormal. The feature value f_m of m is calculated as follows:

$$f_m = (|S_a| + 1) * \sum_{i=1}^{n} s_{mi}.duration. \tag{2}$$

As mentioned in § 2.1, each key of the system state metric m includes the type of the metric and the node it belongs to (e.g., *SQL connection time* on node C), which can be represented by a tuple $(m.node, m.type)$. For a trace t and its corresponding span bag S, we select a subset of spans that are relevant to the metric m as the related sub span bag S_m. S_m represents the set of all behaviors in trace t that are related to metric m (e.g., span 3 and 4 are the related spans of m_1 in Fig. 5). Therefore, we calculate the statistical analysis of the spans in S_m to obtain the feature value f_m of t in the corresponding dimension of m. We measure f_m by calculating the total duration and the number of exceptions in the spans of S_m.

Encode the structure-related part of the trace. The trace structure shows the order and hierarchy of service operations. We call it an invocation tree. Because there are many asynchronous calls in the production environment, the order of spans at the same level on the trace can change [15]. Therefore, we focus on the invocation hierarchy of spans on the trace, rather than the strict order of spans at the same level. For a trace t, each layer of its invocation tree is encoded as a feature in the vector representation. Each span in a layer is represented by its parent span, method name, and argument (referred to as *pma*). If there is no span for a trace at a certain depth, the corresponding position is filled with null.

Dimension scalability and reduction. When new metrics are added or additional traces occur, *Trace Encoder* can automatically expand the vector dimensions without the need for additional manual intervention. For example, for newly introduced metrics, it only requires automatically identifying the related sub span bag as mentioned earlier and performing the necessary statistics. However, the vector expansion mechanism has a side effect of continuously increasing the dimension of the trace vector. To address this, we have adopted a dimension reduction strategy. For the status-related part, if it is observed that two metrics consistently have the same values for their dimensions over a period of time, those dimensions are merged. For the structure-related part, the reason for dimension redundancy is that a certain trace has a very deep call hierarchy but no longer occurs in the future. To address this, we only focus on recent traces. If the dimensions with greater depth have cleared corresponding values across all recent traces, those dimensions are removed.

4.2 System Bias Extractor

As mentioned in § 2.2, the runtime state of the system has a significant impact on sampling preferences. *System Bias Extractor* adaptively determines which metrics are more worthy of attention. It assesses the anomaly degree of various metrics by comparing the current system metric values with the expected values. It then generates a dimension preference vector that represents the preference score of each metric at the present moment.

Assess the anomaly degree of metrics. As described in § 2.1, each metric m consists of a series of time-series data, represented as $m = \{(t_1, v_1), ..., (t_n, v_n)\}$. Our goal is to calculate online the anomaly degree α of each metric at the current time t_k, based on the historical time-series data within a look-back window $[t_1, ..., t_{k-1}]$. We consider two alternative approaches: (i) statistical methods and (ii) neural network models. Given that business operations in real production environments often exhibit periodic patterns, it is crucial to learn the historical variations of the metrics in order to better assess the current degree of fluctuations. Statistical methods such as boxplot [8] and 3-sigma detection [57] can only provide certain statistical measures based on historical data, which

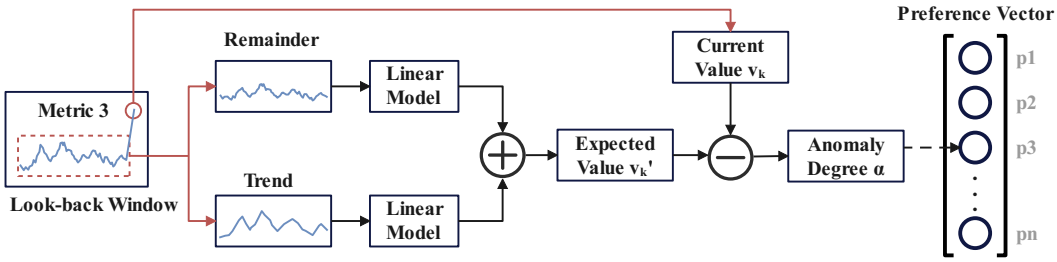


Fig. 6. An example of assessing the anomaly degree of metrics.

fail to learn the historical patterns. Therefore, we opt for the neural network model approach, which can better learns the waveform patterns in historical data.

Due to the online nature of the System Bias Extractor, complex models such as LSTM [36] and Transformer [41] variants are unable to meet the low latency requirements [11]. Therefore, we need a lighter model to fulfill this task. Taking inspiration from recent advancements in time series forecasting (TSF) [5], we have opted for a linear forecasting model. Zeng et al. [49] have demonstrated that linear models outperform Transformer-based methods in time series prediction tasks. We have adopted the DLinear algorithm [49], which combines a Decomposition scheme used in Autoformer [42] and FEDformer [54] with linear layers, and made adjustments for our specific task. This model takes a historical time series data window of a certain metric as input and outputs the expected value of the current time point, denoted as v_k'. The difference between the actual value v_k and the expected value v_k' is used to measure the degree of the anomaly, represented as $\alpha = \frac{|v_k' - v_k|}{max(v_k', v_k)}$. An example of assessing the anomaly degree of metrics is shown in Fig. 6.

Form the preference vector. After obtaining the current anomaly degree α_i for each metric m_i, we consider α_i as the preference score p_i for that metric. The preference scores of all metrics $\mathcal{M}$ form the preference vector $\mathcal{P}$ for the current moment, which is represented as $\mathcal{P} = [p_1, ..., p_n]$.

4.3 System-Biased Sampler

The intuition behind *System-Biased Sampler* is that we prioritize traces that are more relevant to the fluctuations within the system. To achieve this, we maintain a look-back window consisting of recently collected traces. Based on the statistical measures of the traces within the look-back window on each dimension, we calculate an attention score vector for the coming trace. We then take the dot product of this attention score vector with the current system preference vector to determine the system-biased sampling probability.

Calculate the attention score vector. During the sampling process, we maintain a dynamic look-back window, denoted as $\mathcal{W} = [t_1, ..., t_k]$, which consists of the most recent k trace vectors. Here, we only consider the status-related part of each trace vector, denoted as $t_i = [f_{1i}, ..., f_{ni}]$. For each dimension i, we calculate the mean μ_i and standard deviation σ_i of the values taken by the previous k trace vectors in that dimension. When considering the coming trace t_{k+1}, the attention score a_i measures the increase in resource utilization in dimension i compared to the historical traces, using Equation 4.3. The attention scores for all dimensions on the coming trace form the attention score vector, denoted as $\mathcal{A} = [a_1, ..., a_n]$.

$$a_i = \frac{|f_{ik+1} - \mu_i|}{\sigma_i}. \tag{3}$$

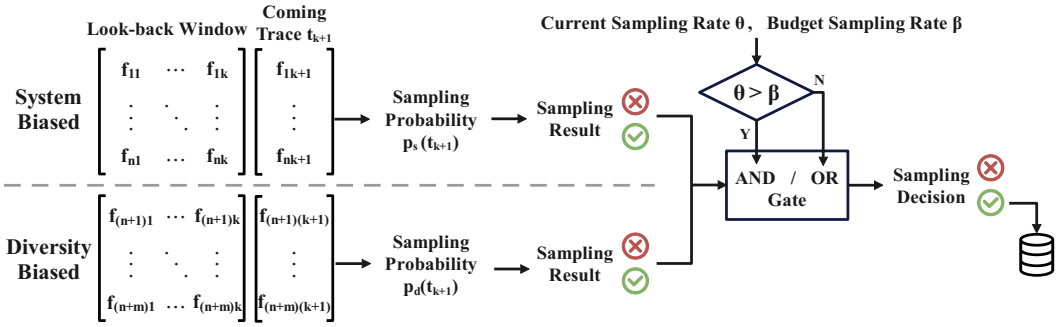


Fig. 7. The process of the dynamic voting mechanism that takes the budget into account to combine the sampling results from the two previous samplers.

Calculate the system-biased sampling probability. When a coming trace obtains a higher attention score in dimensions with higher preference scores, it is more deserving of being kept. Once we obtain the attention score vector $\mathcal{A}(t_{k+1})$ for the coming trace t_{k+1}, we perform a dot product with the current preference vector $\mathcal{P}$, and then apply a *tanh* function to map the dot product result to the $[0, 1]$ range. This yields the system-biased sampling probability $p_s(t_{k+1})$.

$$p_s(t_{k+1}) = \frac{2}{1 + e^{-2\mathcal{P} \cdot \mathcal{A}(t_{k+1})}} - 1. \tag{4}$$

4.4 Diversity-Biased Sampler

The goal of *Diversity-Biased Sampler* is to identify edge-case traces and assign them a higher sampling probability. To achieve this, we cluster the trace vectors within the look-back window and calculate the mass of each cluster. When a new trace arrives, we locate the closest cluster and calculate the diversity-biased sampling probability for the coming trace based on its similarity to the closest cluster and the mass of the cluster.

Cluster traces within the look-back window. To calculate the uncommon degree of the coming trace, we first establish patterns for previous traces. For a look-back window $\mathcal{W} = [t_1, ..., t_k]$, we cluster the trace vectors within it. Assuming we obtain n clusters $C = \{c_1, ..., c_n\}$ after clustering, we calculate the number of traces included in each cluster, which serves as the mass of each cluster, denoted as $\mathcal{M}a = \{ma_1, ..., ma_n\}$.

Calculate the diversity-biased sampling probability. For the coming trace t_{k+1}, we calculate its Jaccard similarity [29] with each cluster and consider the cluster with the highest similarity as the closest cluster, denoted as c'_{k+1}. The mass of c'_{k+1}, denoted as ma'_{k+1}, reflects the commonality of c'_{k+1}. The Jaccard similarity $si(t_{k+1})$ between t_{k+1} and c'_{k+1} reflects their matching degree. Therefore, the smaller the value of ma'_{k+1} and $si(t_{k+1})$, the larger the uncommonness of the coming trace t_{k+1}. We use the following equation to calculate the diversity-biased sampling probability $p_d(t_{k+1})$.

$$p_d(t_{k+1}) = \frac{\frac{1}{ma'_{k+1} * si(t_{k+1})}}{\sum_{i=1}^{k+1} \frac{1}{ma'_i * si(t_i)}}. \tag{5}$$

4.5 Composite Sampler

Composite Sampler provides a more comprehensive measure of the importance of a trace by taking into account both system state and trace diversity. It also considers the impact of the budget on the sampling strategy and utilizes a dynamic voting mechanism to make the final sampling decision.

Table 1. Experiment Datasets Overview.

Dataset	Microservice Benchmark	Trace Number	Metric Number	Batch Number	Uncommon Label Rate	Problem-related Label Rate	Fault Types Number
$\mathcal{A}$	13 Production Microservice systems	909,797	121	62	2.5%	2.5%	6
$\mathcal{B}$	OnlineBoutique and TrainTicket	112,000	32	56	5.0%	5.0%	5

After *System-Biased Sampler* and *Diversity-Biased Sampler* provide their respective sampling probabilities $p_s(t)$ and $p_d(t)$ for the coming trace t, *TraStrainer* generates a random number between $[0, 1]$. It then compares this random number to $p_s(t)$ and $p_d(t)$ separately. If the sampling probability $p(t)$ for a certain aspect is greater than the random number, the sample result for that aspect is considered "*True*", otherwise it is considered "*False*".

Dynamic Voting Mechanism. In order to align the overall sampling rate with the expected budget, we have implemented a dynamic voting mechanism to combine the sampling results from the two previous samplers. Using a look-back window, we dynamically track the recent sampling frequency, denoted as θ, and compare it with the budget sampling rate, denoted as β. If θ is greater than β, we need to enforce stricter sampling rules. In this case, we utilize an *AND* gate as the voting mechanism, meaning that *TraStrainer* will only sample the trace when both samplers yield a "*True*" result. On the other hand, if θ is smaller than β, we can relax the sampling rules. In this scenario, we employ an *OR* gate as the voting mechanism, indicating that *TraStrainer* will sample the trace as long as at least one sampler produces a "*True*" result, as shown in Fig. 7. The sampling decision ultimately determines whether the coming trace t will be stored or discarded.

5 EXPERIMENT EVALUATION

We conducted experiments to evaluate how well *TraStrainer* performs in biasing sampling towards problem-related and edge-case traces. We also assessed its effectiveness for downstream root cause analysis tasks and analyzed the contribution of considering both sampling factors simultaneously. Through these studies, we aim to answer the following research questions(RQs).

- **RQ1:** How does the quality of the traces sampled by *TraStrainer* compare to the baseline approaches?
- **RQ2:** How effective is *TraStrainer* in downstream trace-based root cause analysis compared with baseline approaches?
- **RQ3:** How much does considering both system runtime state and trace diversity contribute to the effectiveness of *TraStrainer*?
- **RQ4:** How efficient is the sampling of *TraStrainer*?

5.1 Datasets

We evaluated *TraStrainer* using two datasets, namely $\mathcal{A}$ and $\mathcal{B}$. Dataset $\mathcal{A}$ consists of real-world data generated from 13 production microservice systems of Huawei. Dataset $\mathcal{B}$ is derived from two widely-used microservices benchmarks, OnlineBoutique [12] and TrainTicket [9]. Table 1 shows the detailed information of the two datasets.

Dataset $\mathcal{A}$ Setup. The dataset is generated from 13 real microservice systems provided by Huawei. It involves 284 services and 1327 nodes. We collected trace data and metric data from 62 incidents that occurred between April 2023 and August 2023, resulting in 62 batches. The incidents include various types of failures such as high CPU load, network delay, slow SQL execution, failed third-party package calls, code logic anomalies, and thread pool exhausted. The dataset comprises a total of 909,797 traces and 121 metrics. SREs and technical experts annotated the uncommon

and problem-related traces, with an average label rate of 2.5%. They also annotated the actual root causes for each batch to evaluate the accuracy of downstream root cause analysis.

Dataset $\mathcal{B}$ Setup. The dataset is generated from two open-source microservice benchmarks: OnlineBoutique [12] and TrainTicket [9], which have been widely used in previous studies [6, 22, 45, 48, 50, 56]. We deployed them on a Kubernetes [1] platform consisting of 12 virtual machines. Each virtual machine is equipped with an 8-core 2.10 GHz CPU, 16GB memory, and runs on Ubuntu 18.04 OS. To collect traces, we utilized Opentelemetry Collector [31] and stored them in Grafana Tempo [13]. In order to simulate latency or reliability issues in the microservice system, we injected a total of 56 faults using Chaosblade [4] into these two microservice benchmarks. We collected corresponding trace data and metric data, resulting in 56 batches. The fault types included CPU contention, CPU consumed, network delay, code exception, and error return. The dataset comprises a total of 112,000 traces and 32 metrics. We annotated the uncommon traces and problem-related traces based on the injected fault positions and types, with an average label rate of 5.0%.

5.2 Baselines

Baseline Sampling Approaches. To evaluate the effectiveness and efficiency of *TraStrainer*, we compare it with four baseline sampling methods as follows.

- Random is the head-based sampling approach that decides whether to capture each trace with equal probability.
- HC [20] is an offline tail-based sampling approach using hierarchical clustering, which group traces by label counting to conduct sampling.
- Sifter [21] is an online tail-based sampling approach, which approximates the distributed system's common-case behavior and samples new traces based on how well represented they are.
- Sieve [16] is an online tail-based sampling approach, which uses robust random cut forest (RRCF) to detect uncommon traces and sample them with a high probability.

Baseline Downstream Analysis Approaches. Root cause analysis (RCA) is a common task performed in downstream analysis. To evaluate the impact of sampling methods on the analysis results, we combined the following three state-of-the-art trace-based RCA methods with different sampling approaches.

- TraceAnomaly [26] leverages deep learning to learn normal trace patterns offline, and then detects abnormal traces and identifies root causes online.
- TraceRCA [23] utilizes spectrum analysis to identify root cause services by analyzing the proportion of normal and abnormal invocations.
- MicroRank [45] identifies and ranks root causes by combining personalised Pagerank method and spectrum analysis.

Variants of *TraStrainer*. To evaluate the impact of combining system state and trace diversity, we create the following two variants and conduct ablation experiments.

- *TraStrainer* w/o $\mathcal{M}$ is a variant that only considers trace diversity to set sampling preferences. We achieve this by using the sampling results from the Diversity-Biased Sampler as the final sampling outcome.
- *TraStrainer* w/o $\mathcal{D}$ is a variant that considers only the system runtime state to set sampling preferences. We achieve this by using the sampling results from the System-Biased Sampler as the final sampling outcome.

5.3 Evaluation Metrics

To evaluate the quality of the sampling results, we rely on two metrics: proportion and diversity. Both of them have been widely utilized in previous studies [11, 16, 51] and are defined as follows.

- **Proportion** (*PRO*) reflects the ability of an approach to bias towards valuable traces. Let T be the total number of labeled traces, and t denote the number of labeled traces that are sampled. The proportion of labeled traces can be calculated as: $PRO = \frac{t}{T}$. In our experiments, we had three variants based on different labels: the proportion of uncommon traces, the proportion of problem-related traces, and the proportion of traces that are both uncommon and problem-related.
- **Diversity** (*DIV*) reflects the ability of an approach to provide a representative sample. We denote the sampled traces obtained after sampling as $ST = \{st_1, ..., st_n\}$. Clustering them based on execution path yields trace patterns $TP = \{tp_1, ..., tp_m\}$. The diversity of the sampling results is simply the number of trace patterns obtained, which is denoted as $DIV = m$.

To assess the effectiveness of downstream root cause analysis, we employ three commonly used evaluation metrics: *A*@1, *A*@3, and *MRR*. These metrics serve as standard measures for evaluating the performance of RCA methods [25, 45] and are defined as follows.

- **Top-k accuracy (*A*@*k*)** represents the probability that the true root cause is included in the top-k positions of the results. Let rc_i be the root cause of the i-th issue, $Rank_i^k$ be the top-k result list for the ith issue. $A@k$ can be calculated as: $A@k = \frac{1}{|I|}\sum_{i=1}^{|I|}\left(rc_i \in \text{Rank}_i^k\right)$. Higher values of $A@k$ indicate better accuracy.
- **Mean reciprocal rank (*MRR*)** represents the inverse of the rank of the first identified root cause. If the actual root cause is not present in the result list, the rank is considered to be infinity and its reciprocal is zero. Let r_i be the rank of the root cause in the returned list for the ith issue. The calculation for *MRR* is: $MRR = \frac{1}{|I|}\sum_{i=1}^{|I|}\frac{1}{\text{rs}_i}$. Higher values of *MRR* indicate better accuracy.

5.4 RQ1: Sampling Quality of *TrasTrainer*.

To evaluate the sampling quality of *TraStrainer*, we compare its sampling results with four baseline methods. Our evaluation of sampling quality is divided into two aspects: bias sampling and representative sampling.

5.4.1 Bias Sampling. Our first set of experiments evaluates *TraStrainer*'s ability to bias towards uncommon traces and problem-related traces. We conducted experiments on both dataset $\mathcal{A}$ and $\mathcal{B}$. To investigate the impact of different budgets, we set five gradient budget sampling rates: 0.1%, 1%, 2.5%, 5%, and 10%. These rates encompass budgets that are both higher and lower than the label rate. As the online sampling approach does not allow for precise fixed sampling rates, we followed the approach used in previous study [16]. If the final sampling quantity exceeded the budget, we removed a portion of the sampled traces in reverse order until the sampling result met the budget. Conversely, if the sampling quantity fell short of the budget, we randomly selected additional traces from the unsampled ones to match the budget. We conducted repeated experiments with the same settings for each batch and recorded the average proportion of the results.

Bias towards uncommon traces. Fig. 8 (a1) and (b1) show the proportions of uncommon traces sampled by different approaches across various budget settings in two datasets. *TraStrainer* and Sieve exhibit similar performance, outperforming the other three approaches. Random sampling, with no specific preference, achieves proportions roughly equivalent to the budget sampling rate. HC and Sifter, which solely consider trace structure while disregarding time delays, lack the ability to identify uncommon traces that only differ in execution time. Therefore, their proportions on both datasets do not exceed 0.7.

We conducted a detailed comparison between *TraStrainer* and Sieve, both of which consider both time delays and structure during encoding, resulting in superior recognition of uncommon traces compared to other methods. When the budget is low, *TraStrainer* performs better, thanks to its metric-based encoding approach, which effectively distinguishes anomalies. However, when

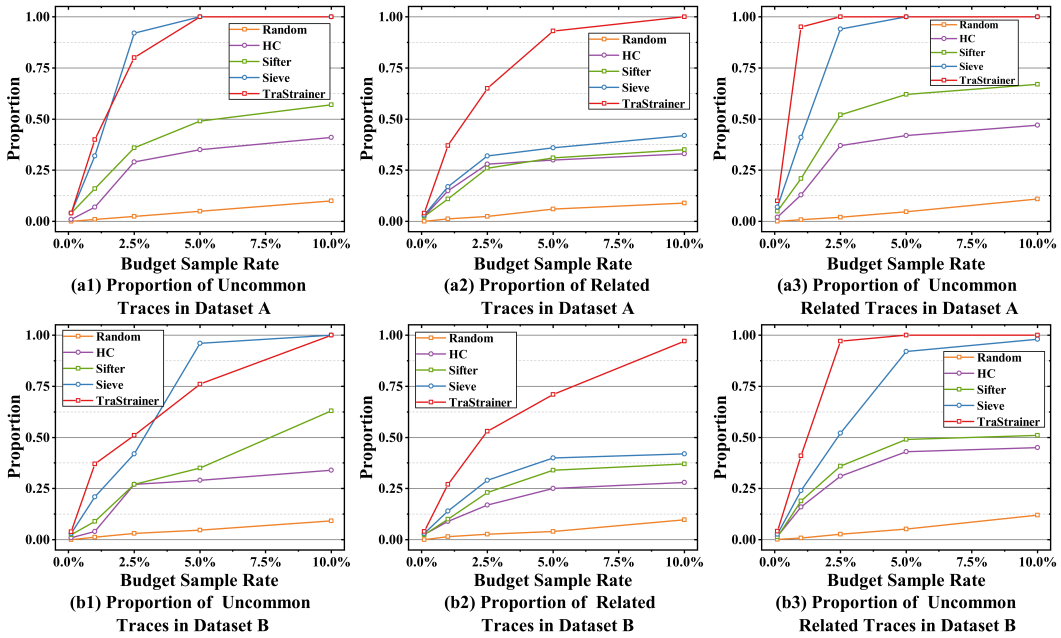


Fig. 8. Proportions of uncommon, related, and uncommon-related traces sampled by different approaches across different budget settings in two datasets.

the budget is around the label rate (2.5% in dataset $\mathcal{A}$ and 5% in $\mathcal{B}$), *TraStrainer*'s proportion is lower than Sieve. This is because, in addition to bias towards uncommon traces, *TraStrainer* also exhibits a bias towards problem-related traces, which include some common traces. When the budget exceeds twice the label rate, both *TraStrainer* and Sieve achieve a proportion of 1, meaning all uncommon traces are captured.

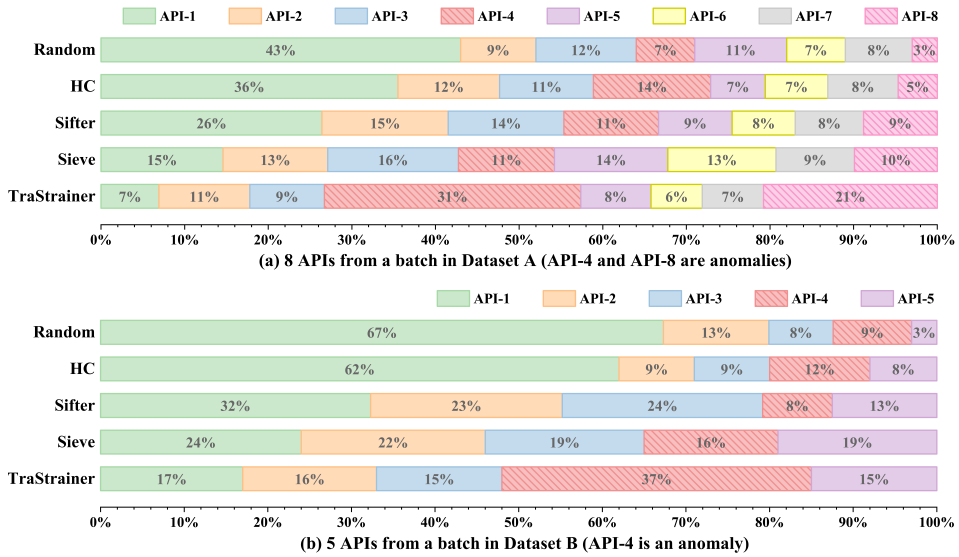
Bias towards problem-related traces. Fig. 8 (a2) and (b2) show the proportions of problem-related traces. *TraStrainer* outperforms the four baselines in both datasets significantly. Sieve, Sifter, and HC exhibit similar proportion growth to random when the budget is high, with proportions below 0.5 in both datasets. *TraStrainer* takes system runtime state into account when setting sampling preferences. When the budget equals the label rate, the proportion for problem-related traces is around 0.6. When the budget is twice the label rate, the proportion approaches 1.0, meaning that all problem-related traces are captured.

Bias towards uncommon problem-related traces. Fig. 8 (a3) and (b3) show the proportions of traces that are both uncommon and problem-related. *TraStrainer* outperforms the baseline methods with better results even at lower budgets. When the budget rate equals the label rate (1% in dataset $\mathcal{A}$ and 2.5% in $\mathcal{B}$), *TraStrainer* achieves a proportion above 0.9 in both datasets, while the other baselines remain below 0.55. This indicates that *TraStrainer* not only prefers uncommon traces at lower budgets but also shows a stronger preference for problem-related traces within the uncommon traces.

5.4.2 Representative Sampling. Our next set of experiments assess the diversity of *TraStrainer*'s sampling results and its capability to prioritize underrepresented and abnormal request types. Similar to the previous experiments, we conducted comparisons with baselines on two datasets at various budget sampling rates. Additionally, we evaluated the sampling effectiveness of two variants of *TraStrainer*.

Table 2. Comparison of the result diversity of different sampling approaches across different budget settings.

Approach	DataSet $\mathcal{A}$					DataSet $\mathcal{B}$				
	0.1%	**1.0%**	**2.5%**	**5.0%**	**10.0%**	**0.1%**	**1.0%**	**2.5%**	**5.0%**	**10.0%**
Random	5.95	28.03	36.77	48.42	61.68	6.61	21.16	27.77	32.79	37.56
HC	7.27	28.11	40.68	52.77	67.60	9.04	27.56	33.86	39.02	41.84
Sifter	9.24	44.31	58.07	74.23	85.62	12.05	35.50	43.85	44.02	45.06
Sieve	13.70	98.24	151.27	155.10	161.24	13.70	51.27	54.44	54.44	54.44
TraStrainer w/o $\mathcal{M}$	**15.7**	**101.54**	**158.27**	**162.40**	**166.35**	**14.92**	**53.50**	**54.44**	**54.44**	**54.44**
TraStrainer w/o $\mathcal{D}$	8.06	28.53	42.66	55.26	73.23	6.05	17.73	23.03	27.34	32.45
TraStrainer	**14.65**	**95.87**	**139.84**	**153.61**	**159.15**	**14.34**	**47.34**	**53.42**	**54.44**	**54.44**



(a) 8 APIs from a batch in Dataset A (API-4 and API-8 are anomalies)

(b) 5 APIs from a batch in Dataset B (API-4 is an anomaly)

Fig. 9. Distribution of different APIs in the sampling results.

Diversity of the sampling result. Table 2 shows the diversity of sampling results. It can be observed that *TraStrainer* w/o $\mathcal{M}$ achieves the best performance in both datasets, demonstrating the effectiveness of *TraStrainer*'s metric-based trace encoding in distinguishing different trace patterns and ensuring fair sampling. Sieve follows closely behind as it considers both structure and latency during encoding. The diversity of *TraStrainer*'s sampling results is slightly lower than the previous two approaches because *TraStrainer* also takes preferences for related traces into account. On average, *TraStrainer*'s diversity is only slightly lower than *TraStrainer* w/o $\mathcal{M}$ by less than 5%. HC and Sifter, lacking consideration for latency, exhibit noticeably lower diversity in their sampling results compared to the previous three approaches. *TraStrainer* w/o $\mathcal{D}$, which does not consider diversity preferences, shows no clear pattern in the diversity of its sampling results.

Representative ability. We selected a batch from dataset $\mathcal{A}$ and $\mathcal{B}$, respectively. In dataset A, the traces consist of 8 different high-level API calls, with the following distribution ratios: 41%, 12%, 11%, 10%, 9%, 7%, 6%, and 4%. Among them, API-4 and API-8 are considered as exceptional interfaces. In dataset $\mathcal{B}$, the traces consist of 5 different high-level API calls, with the following distribution ratios: 68%, 11%, 9%, 8%, and 4%. API-4 is considered as an exceptional interface in this case. We conducted 50 repeated experiments using the same setup and recorded the average distribution ratios of the sampling results for each approach.

Fig. 9 shows the distribution ratios of different APIs in the sampling results. It can be observed that *TraStrainer* enhances the representation of less frequently used APIs, resulting in a more balanced sampling. Additionally, considering the system's state, *TraStrainer* also improves the

Table 3. Comparison of the effects of different sampling approaches in downstream root cause analysis.

RCA Approach	Sampling Approach	A@1			A@3			MRR		
		0.1%	**1.0%**	**2.5%**	**0.1%**	**1.0%**	**2.5%**	**0.1%**	**1.0%**	**2.5%**
TraceAnomaly	Random	10.71	9.26	16.67	44.73	57.41	57.41	0.2820	0.3503	0.3997
	HC	9.26	12.96	18.52	37.04	42.59	55.56	0.2590	0.3664	0.3747
	Sifter	11.67	24.07	16.67	37.04	57.41	62.96	0.2753	0.4025	0.4145
	Sieve	8.81	18.52	29.63	44.44	53.70	57.41	0.2620	0.3762	0.4383
	TraStrainer w/o $\mathcal{M}$	11.67	20.37	22.22	45.15	51.85	55.56	0.2903	0.3722	0.4068
	TraStrainer w/o $\mathcal{D}$	12.96	38.89	44.44	49.81	77.78	75.93	0.3485	0.5948	0.6247
	TraStrainer	**46.30**	**51.61**	**54.84**	**66.67**	**79.19**	**87.10**	**0.5707**	**0.6438**	**0.7151**
TraceRCA	Random	7.41	20.37	29.63	40.74	61.11	68.52	0.2525	0.4123	0.4991
	HC	9.26	24.07	24.07	46.30	62.96	62.96	0.2546	0.4324	0.4627
	Sifter	8.67	19.63	25.19	37.04	55.56	61.11	0.2449	0.4272	0.4836
	Sieve	18.52	31.48	38.89	42.59	51.85	57.41	0.3008	0.4157	0.4873
	TraStrainer w/o $\mathcal{M}$	18.52	33.33	35.19	44.44	55.56	55.56	0.3191	0.4432	0.4642
	TraStrainer w/o $\mathcal{D}$	24.07	55.56	55.56	38.89	81.48	77.78	0.3650	0.6880	0.6843
	TraStrainer	**55.56**	**55.56**	**58.06**	**70.37**	**85.19**	**89.63**	**0.6265**	**0.7019**	**0.7510**
MicroRank	Random	5.56	16.67	27.78	20.37	50.00	61.11	0.1571	0.3423	0.4352
	HC	7.41	18.52	22.22	27.78	46.30	51.85	0.1954	0.3398	0.3731
	Sifter	5.56	18.52	27.78	23.42	46.30	61.11	0.1605	0.3414	0.4358
	Sieve	9.26	25.83	35.19	20.37	58.15	62.96	0.1657	0.4246	0.4963
	TraStrainer w/o $\mathcal{M}$	12.96	16.67	24.07	42.59	42.59	55.56	0.2994	0.3241	0.4012
	TraStrainer w/o $\mathcal{D}$	29.63	42.59	46.30	74.04	68.52	72.22	0.5228	0.5463	0.5509
	TraStrainer	**42.59**	**45.16**	**50.00**	**77.74**	**78.52**	**82.26**	**0.5509**	**0.5889**	**0.6556**

representation of APIs that are related to system issues and experience deterioration. In comparison, HC and Sifter only slightly increase the representation of low-frequency APIs, while Sieve achieves balanced sampling results but fails to enhance the representation of deteriorating APIs.

5.5 RQ2: Effectiveness of *TrasTrainer* for Downstream Root Cause Analysis.

Root cause analysis (RCA) is a typical downstream analysis task. We evaluate the impact of sampling strategies on the effectiveness of RCA approaches. We use the sampled traces obtained through the sampling method as input for the RCA method to combine them, conducting experiments on two datasets. To explore the performance of sampling at different budgets, we conducted experiments at budget sampling rates of 0.1%, 1.0%, and 2.5%. The average results on the two datasets are shown in Table 3. It is evident that the variant employing *TraStrainer* as the sampling strategy achieves the best results for each RCA methods, in terms of *A*@1, *A*@3, and *MRR*.

In various variants of TraceAnomaly, *TraStrainer*-based approach significantly improved *A*@1, *A*@3, and *MRR* by 33.78%, 26.88%, and 31.45%, respectively. TraceAnomaly employs a VAE-based model to detect anomalies and locate root causes, and the quality of the traces used to train this VAE-based model has a significant impact on root cause localization. We observed that when the budget is low, variants of the baselines HC, Sifter, and Sieve performed poorly, even slightly worse than the results based on random sampling. Further analysis of intermediate results revealed that in this phase, the baselines primarily obtained abnormal traces, making it difficult for the VAE-based model to learn normal trace patterns effectively, thus hindering accurate root cause localization. *TraStrainer*, on the other hand, focuses not only on abnormal traces but also on problem-related traces, enabling better results with lower sampling budgets. As the sampling budget increases, the performance of all variants improves. However, the baselines' *A*@1 remains consistently below 30%, while *TraStrainer* achieves an *A*@1 of 54.84% and an *MRR* of 0.7151.

Both TraceRCA and MicroRank employ spectrum analysis, which means that a microservice with more abnormal traces and fewer normal traces is more likely to be the root cause. In various

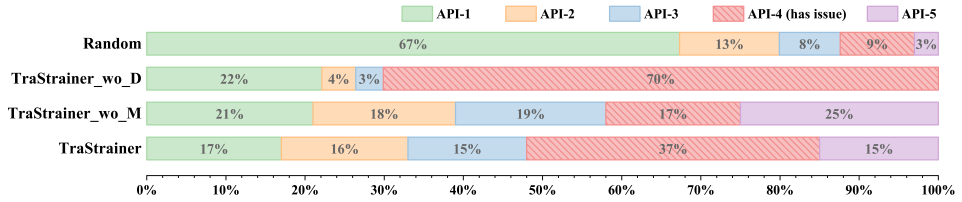


Fig. 10. Distribution of different APIs in the sampling results of different variants.

variants of these two methods, *TraStrainer*-based approach improved *A*@1, *A*@3, and *MRR* by 31.48%, 38.34%, and 28.96%, respectively. We can observe that variants of HC and Sifter show analysis results similar to random sampling methods at various budgets. This is because TraceRCA and MicroRank determine abnormal invocations based solely on latency, and these two sampling approaches consider only trace structure during encoding, thus providing no assistance to root cause analysis. As for the Sieve-based approach, it performs poorly at lower budgets because spectrum analysis weaves clues from both normal and anomalous traces for root cause localization, and the Sieve-based approach cannot provide a sufficient number of normal traces. *TraStrainer* consistently achieves good analysis results across different budgets, maintaining an *A*@3 above 70%. This is due to *TraStrainer*'s more comprehensive sampling preferences. Even with lower budgets, it discards traces unrelated to the problem rather than valuable traces for analysis, resulting in acceptable analysis performance.

5.6 RQ3: Contribution of the Combination of System Status and Trace Diversity.

We performed ablation experiments to examine the impact of combining system state and trace diversity in the sampling preference setting. We created two variants: *TraStrainer* w/o $\mathcal{M}$, which solely focuses on bias sampling using trace diversity, and *TraStrainer* w/o $\mathcal{D}$, which solely considers system state. We replicated partial experiments from RQ1 and RQ2 on these two variants and analyzed the experimental results.

Fig. 10 shows the distribution of different APIs in the sampling results sampled by various variants in dataset B. It is evident that the sampling results of *TraStrainer* w/o $\mathcal{D}$ exhibit a significant imbalance in the distribution ratios of different APIs. Specifically, the distribution ratio of deteriorating interface API-4 exceeds 70%, while the underrepresented API-5 is completely absent in the samples. Although *TraStrainer* w/o $\mathcal{M}$ produces relatively balanced sampling results on average, it fails to bias towards the more crucial deteriorating API-4. On the other hand, *TraStrainer* not only enhances the diversity of the sampling results but also exhibits a stronger inclination towards the problematic API.

Table 3 shows the effectiveness comparison of different variants of *TraStrainer* in downstream analysis. It can be seen that *TraStrainer* w/o $\mathcal{D}$ performs better overall than *TraStrainer* w/o $\mathcal{M}$, suggesting that prioritizing problem-related traces yield better results for analysis tasks compared to solely focusing on uncommon traces. When analyzing the effects under different budgets, we notice that at a low budget (0.1%), *TraStrainer* w/o $\mathcal{D}$ has a significantly lower *A*@1 compared to *TraStrainer*. However, at a high budget (2.5%), *TraStrainer* w/o $\mathcal{D}$ approaches the performance of *TraStrainer*. This is because *TraStrainer* w/o $\mathcal{D}$'s sampling preference is solely based on problem-relatedness without bias towards uncommon abnormal traces, potentially missing root cause traces at low budgets. As the budget increases, *TraStrainer* w/o $\mathcal{D}$ captures most of the problem-related traces including the culprits, resulting in a noticeable improvement in performance. On the other hand, *TraStrainer*, which considers both system state and trace diversity, achieves better analysis performance across all budgets. This finding further confirms the motivation, as described in § 2.2, of combining system state and trace diversity to enhance the effectiveness of trace sampling.

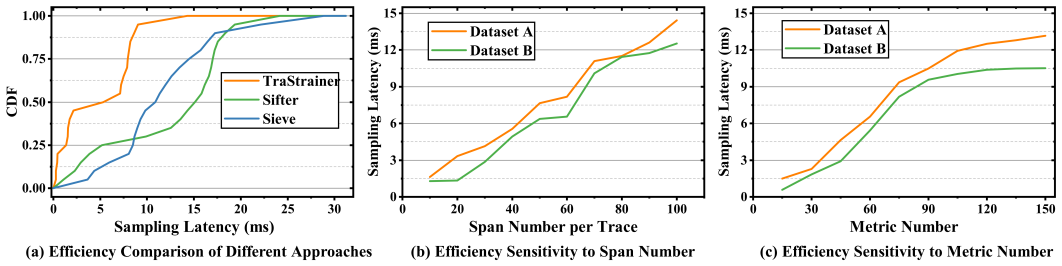


Fig. 11. Efficiency Validation of *TraStrainer*.

5.7 RQ4: Efficiency of *TraStrainer*.

We evaluated the efficiency of *TraStrainer* through a series of experiments. It is worth noting that in *TraStrainer*, the *System-Bias Extractor* and *Trace Sampler* run in parallel, and *Trace Sampler* only needs to read the current preference vector. During the efficiency validation, we deployed the sampler part of *TraStrainer* in a single-process form and conducted it on an Intel Xeon Gold 5318Y 2.10GHz CPU. We measured the average time it took for each trace to go through *TraStrainer* and make a sampling decision.

Fig. 11(a) shows the distribution of sampling latencies for traces from the two datasets. It is evident that *TraStrainer* generally exhibits smaller sampling latencies compared to the baselines, ranging from 0.28ms to 14.29ms. The maximum latency is 40.82% to 50.16% lower than the baselines. Two factors will affect the sampling latency: trace size and the number of metrics. Fig. 11(b) and (c) demonstrate the impact of these factors on *TraStrainer*'s sampling latency. Following the principle of controlling variables, we fixed the metric number at 75 when investigating the effect of trace size, and fixed the trace size at 50 when examining the impact of metric number. It can be observed that sampling latency exhibits a linear increase with trace size, but the maximum value remains below 15ms. When the metric number is low, the sampling latency also increases linearly. However, when the metric number exceeds 100, the growth rate of sampling latency becomes much slower. This is because *TraStrainer* employs dimension reduction strategies to prevent dimension explosion.

From the above analysis, it can be seen that *TraStrainer* is efficient enough for online sampling. Additionally, the dimension reduction strategy of *TraStrainer* can effectively improve the efficiency when there are a large number of metrics.

6 LIMITATIONS

In this work, we only implement biased sampling in a tail-based way. However, the most recent research [52] on trace sampling introduces the concept of retroactive sampling [52], which lazily retrieves trace data to perform biased sampling at earlier stages of the trace lifecycle, aiming to improve the efficiency of biased sampling. Nonetheless, current retroactive sampling approach also solely biases towards edge-cases. Our work proposes a more effective method for setting sampling preferences. We believe that this can be beneficial for various types of biased sampling, not only tail-based sampling but also retroactive sampling.

7 THREATS TO VALIDITY

The main threats to internal validity are associated with the implementation of baselines. For Sieve, we utilize its open-source implementation directly. However, since HC and Sifter lack open-source implementations, we carefully develop our own implementations and use exactly the same libraries based on relevant papers. Additionally, previous work has mentioned that online samplers make essentially random sampling decisions in the initial sampling phase due to the random initialization

of the internal model [16, 21]. To facilitate result comparison, we allow each online sampler to undergo a warm-up period to ensure that the model is initialized before testing the sampling effectiveness. During the experiments, we use 10% of the traces to warm up the sampler.

The threats to external validity mainly lie in the representativeness of the datasets. Although we used data from 13 production microservice systems and two open-source microservice benchmarks for validation, the number of fault types we tested was still limited (7 in total). There could be more complex fault types in other microservice systems, so further testing is needed to assess the effectiveness of *TraStrainer* in more complex scenarios.

8 RELATED WORK

Distributed traces have become the primary source of observability and analysis for microservice systems [39], widely utilized for profiling, diagnosing, and debugging. Additionally, due to the large amount of trace data involved, sampling becomes essential in distributed tracing [16]. Below, we provide a brief summary of current methods for trace-based analysis and trace sampling techniques.

Trace-based analysis approaches. In microservice systems, trace analysis serves various purposes like anomaly detection (AD) and root cause analysis (RCA). Nedelkoski et al.[28] introduced a method for anomaly detection that employs a multimodal LSTM network, allowing the model to learn from trace data. TraceAnomaly [26] proposes an anomaly detection and root cause localization approach using Variational Autoencoders (VAE) with trace data. TraceRCA [23] and MicroRank [45] identifies root cause services by examining the ratio of normal to abnormal invocations and applying spectrum analysis. Sage [10] utilizes causal Bayesian networks and employs graphical variational autoencoders to pinpoint root cause microservices. FSF [35] utilizes knowledge of failure propagation and the client-server communication model to deduce root causes.

Trace sampling approaches. Uniform random sampling approach has been utilized by tracing systems like Dapper [37], Jaeger [17] and Zipkin [2]. However, this random sampling method fails to guarantee the quality of the sampled traces. Consequently, subsequent methods have introduced several biased sampling approaches. Las-Casas et al. [20] propose a hierarchical clustering method PERCH to bias the sampling to maximize the diversity of traces based on label counting. However, this approach was primarily designed for offline sampling. Sifter [21] performs an online sampling method, which approximates the common-case model of the traces and biases towards traces that are poorly represented by this model. Sieve [16] provides an online sampler that aims to bias sampling towards uncommon traces by using robust random cut forest. TraceCRL [51] utilizes contrastive learning and graph neural network techniques to encode trace data into vectors, enabling the sampling of traces based on the diversity of these vector representations. SampleHST [11] biases towards the edge-cases based on the distribution of mass scores obtained from a forest of Half Space Trees (HST). Hindsight [52] describes and implements retroactive sampling for capturing traces of symptomatic edge-cases. We can see that previous biased trace sampling approaches were merely different variations based on the fundamental principle of being biased towards edge-case traces, without introducing new perspectives for consideration.

Perspectives of *TraStrainer*. Compared to existing trace sampling methods, *TraStrainer* takes a more comprehensive approach to sampling preferences. It not only considers trace diversity but also takes into account the impact of system runtime state on sampling preferences. Previous trace sampling methods were limited to using trace information alone to set sampling preferences. In contrast, *TraStrainer* considers the information of both the traces and the system state metrics, utilizing multiple modalities of information for better sampling results. Additionally, *TraStrainer* is designed with the requirements of downstream trace-based analysis methods [10, 23, 26, 28, 45], providing higher-quality sampled data for these methods.

9 CONCLUSION

In this study, we introduce *TraStrainer*, an online biased sampler that takes into account sampling preferences more comprehensively. The core idea behind *TraStrainer* is to consider not only trace diversity but also system runtime state, allowing for dynamic adjustment of sampling preferences. *TraStrainer* employs a more interpretable and comprehensive encoding method for traces and utilizes a dynamic voting mechanism to generate integrated sampling decisions. We construct two datasets and conduct a comprehensive evaluation of *TraStrainer*, including the quality of sampling results, performance in downstream tasks, and sampling efficiency. The experimental results demonstrate that compared to four baseline methods, *TraStrainer* can identify more valuable traces within the same budget and improve the performance of downstream analysis methods. Furthermore, *TraStrainer* exhibits higher efficiency than two other online biased.

Artifact Availability. The basic implementation of *TraStrainer* are publicly available at [40].

ACKNOWLEDGMENTS

The research is supported by the Guangdong Basic and Applied Basic Research Foundation (No.2023B1515020054) and the National Natural Science Foundation of China (No.62272495). This research is also sponsored by Huawei. The corresponding author is Pengfei Chen.

REFERENCES

[1] 2023. Kubernetes Homepage. http://kubernetes.io/. [Online].
[2] 2023. Zipkin Homepage. https://zipkin.io. [Online].
[3] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D Ernst. 2016. Debugging distributed systems. *Commun. ACM* 59, 8 (2016), 32–37.
[4] Chaosblade. 2023. Chaosblade. https://github.com/chaosblade-io/chaosblade. Accessed Jan. 6, 2023.
[5] Chris Chatfield. 2000. *Time-series forecasting*. CRC press.
[6] Yufu Chen, Meng Yan, Dan Yang, Xiaohong Zhang, and Ziliang Wang. 2022. Deep Attentive Anomaly Detection for Microservice Systems with Multimodal Time-Series Data. In *ICWS 2022*. IEEE, 373–378.
[7] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. 2007. {X-Trace}: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*.
[8] Michael Frigge, David C Hoaglin, and Boris Iglewicz. 1989. Some implementations of the boxplot. *The American Statistician* 43, 1 (1989), 50–54.
[9] FudanSELab. 2023. TrainTicket. https://github.com/FudanSELab/train-ticket. Accessed Jan. 6, 2023.
[10] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 135–151. https://doi.org/10.1145/3445814.3446700
[11] Alim Ul Gias, Yicheng Gao, Matthew Sheldon, José A. Perusquía, Owen O'Brien, and Giuliano Casale. 2022. SampleHST: Efficient On-the-Fly Selection of Distributed Traces. arXiv:2210.04595 [cs.DC]
[12] GoogleCloudPlatform. 2023. OnlineBoutique. https://github.com/GoogleCloudPlatform/microservices-demo. Accessed Jan. 6, 2023.
[13] Grafana. 2023. Grafana Tempo. https://github.com/grafana/tempo. [Online].
[14] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. 2020. Graph-Based Trace Analysis for Microservice Architecture Understanding and Problem Diagnosis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1387–1397. https://doi.org/10.1145/3368089.3417066
[15] Lexiang Huang and Timothy Zhu. 2021. Tprof: Performance Profiling via Structural Aggregation and Automated Analysis of Distributed Systems Traces. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 76–91. https://doi.org/10.1145/3472883.3486994
[16] Zicheng Huang, Pengfei Chen, Guangba Yu, Hongyang Chen, and Zibin Zheng. 2021. Sieve: Attention-based Sampling of End-to-End Trace Data in Distributed Microservice Systems. In *2021 IEEE International Conference on Web Services (ICWS)*. 436–446. https://doi.org/10.1109/ICWS53863.2021.00063
[17] jaeger. 2023. Jaeger. https://www.jaegertracing.io/. Accessed: 2023/7/14.

[18] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 34–50. https://doi.org/10.1145/3132747.3132749

[19] Kmaork. 2023. Hypno. https://docs.aws.amazon.com/prescriptive-guidance/latest/implementing-logging-monitoring-cloudwatch/configure-cloudwatch-ec2-on-premises.html. Accessed Jan. 6, 2023.

[20] Pedro Las-Casas, Jonathan Mace, Dorgival Guedes, and Rodrigo Fonseca. 2018. Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 326–332. https://doi.org/10.1145/3267809.3267841

[21] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. 2019. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 312–324. https://doi.org/10.1145/3357223.3362736

[22] Xing Li, Yan Chen, and Zhiqiang Lin. 2019. Towards automated inter-service authorization for microservice applications. In *SIGCOMM 2019*. ACM, 3–5.

[23] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, Zhekang Chen, Wenchi Zhang, Xiaohui Nie, Kaixin Sui, and Dan Pei. 2021. Practical Root Cause Localization for Microservice Systems via Trace Analysis. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. 1–10. https://doi.org/10.1109/IWQOS52092.2021.9521340

[24] Jinjin Lin, Pengfei Chen, and Zibin Zheng. 2018. Microscope: Pinpoint Performance Issues with Causal Graphs in Micro-service Environments. In *ICSOC 2018*. Springer, 3–20. https://doi.org/10.1007/978-3-030-03596-9_1

[25] Dewei Liu, Chuan He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshun Wu. 2021. MicroHECL: High-Efficient Root Cause Localization in Large-Scale Microservice Systems. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice* (Virtual Event, Spain) *(ICSE-SEIP '21)*. IEEE Press, 338–347. https://doi.org/10.1109/ICSE-SEIP52600.2021.00043

[26] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, and Dan Pei. 2020. Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. 48–58. https://doi.org/10.1109/ISSRE5003.2020.00014

[27] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. 2019. Anomaly Detection from System Tracing Data Using Multimodal Deep Learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 179–186. https://doi.org/10.1109/CLOUD.2019.00038

[28] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. 2019. Anomaly Detection from System Tracing Data Using Multimodal Deep Learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 179–186. https://doi.org/10.1109/CLOUD.2019.00038

[29] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. 2013. Using of Jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, Vol. 1. 380–384.

[30] Opentelemetry. 2023. Opentelemetry. https://opentelemetry.io. Accessed: 2023/7/14.

[31] Opentelemetry. 2023. OpenTelemetry Collector. https://github.com/open-telemetry/opentelemetry-collector. [Online].

[32] Opentelemetry. 2023. Opentelemetry span-events concept. https://opentelemetry.io/docs/concepts/signals/traces/#span-events. Accessed: 2023/7/14.

[33] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. 2020. *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media.

[34] Thomas W. Reps, Thomas Ball, Manuvir Das, and James R. Larus. 1997. The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem. In *6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*. 432–449.

[35] Jesus Rios, Saurabh Jha, and Laura Shwartz. 2022. Localizing and Explaining Faults in Microservices Using Distributed Tracing. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. 489–499. https://doi.org/10.1109/CLOUD55607.2022.00072

[36] Alex Sherstinsky. 2020. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *Physica D: Nonlinear Phenomena* 404 (2020), 132306.

[37] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[38] Apache SkyWalking. 2023. Apache SkyWalking. https://skywalking.apache.org. Accessed July. 6, 2023.

[39] Cindy Sridharan. 2018. *Distributed systems observability: a guide to building robust systems*. O'Reilly Media.
[40] TraStrainer. 2024. TraStrainer implementation. https://github.com/IntelligentDDS/TraStrainer. Accessed Feb. 20, 2024.
[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[42] Haixu Wu, Jiehui Xu, Jianmin Wang, and Mingsheng Long. 2021. Autoformer: Decomposition transformers with auto-correlation for long-term series forecasting. *Advances in Neural Information Processing Systems* 34 (2021), 22419–22430.
[43] Zihao Ye, Pengfei Chen, and Guangba Yu. 2021. T-Rank: A Lightweight Spectrum based Fault Localization Approach for Microservice Systems. In *CCGrid 2021*. IEEE/ACM, 416–425. https://doi.org/10.1109/CCGrid51090.2021.00051
[44] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 159–172.
[45] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. MicroRank: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments. In *WWW 2021*. ACM, 3087–3098.
[46] Guangba Yu, Pengfei Chen, Pairui Li, Tianjun Weng, Haibing Zheng, Yuetang Deng, and Zibin Zheng. 2023. LogReducer: Identify and Reduce Log Hotspots in Kernel on the Fly. In *ICSE 2023*. 1763–1775.
[47] Guangba Yu, Pengfei Chen, Yufeng Li, Hongyang Chen, Xiaoyun Li, and Zibin Zheng. 2023. Nezha: Interpretable Fine-Grained Root Causes Analysis for Microservices on Multi-Modal Observability Dat. In *ESEC/FSE 2023*. ACM, 1–1. https://doi.org/10.1145/3611643.3616249
[48] Guangba Yu, Pengfei Chen, and Zibin Zheng. 2019. Microscaler: Automatic Scaling for Microservices with an Online Learning Approach. In *ICWS 2019*. IEEE, 68–75.
[49] Ailing Zeng, Muxi Chen, Lei Zhang, and Qiang Xu. 2023. Are transformers effective for time series forecasting?. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 37. 11121–11128.
[50] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. DeepTraLog: Trace-Log Combined Microservice Anomaly Detection through Graph-based Deep Learning. In *ICSE 2022*. IEEE, 623–634.
[51] Chenxi Zhang, Xin Peng, Tong Zhou, Chaofeng Sha, Zhenghui Yan, Yiru Chen, and Hong Yang. 2022. TraceCRL: Contrastive Representation Learning for Microservice Trace Analysis. In *ESEC/FSE 2022*. ACM, 1221–1232.
[52] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. 2023. The Benefit of Hindsight: Tracing {Edge-Cases} in Distributed Systems. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 321–339.
[53] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. In *SoCC 2018*. ACM, 149–161.
[54] Tian Zhou, Ziqing Ma, Qingsong Wen, Xue Wang, Liang Sun, and Rong Jin. 2022. Fedformer: Frequency enhanced decomposed transformer for long-term series forecasting. In *International Conference on Machine Learning*. PMLR, 27268–27286.
[55] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering* 47, 2 (2018), 243–260.
[56] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE TSE* 47, 2 (2021), 243–260.
[57] Zhenyi Zhu. 2022. Anomaly detection over time series data. (2022).

Received 2023-09-29; accepted 2024-01-23