TRACEMESH: Scalable and Streaming Sampling for Distributed Traces

Zhuangbin Chen[§], Zhihan Jiang[‡], Yuxin Su[§], Michael R. Lyu[‡], Zibin Zheng[§¶]
[§]Sun Yat-sen University, Zhuhai, China, {chenzhb36, suyx35, zhzibin}@mail.sysu.edu.cn
[‡]The Chinese University of Hong Kong, Hong Kong, China, {zhjiang22, lyu}@cse.cuhk.edu.hk

***Abstract*—Distributed tracing serves as a fundamental element in the monitoring of cloud-based and datacenter systems. It provides visibility into the full lifecycle of a request or operation across multiple services, which is essential for understanding system dependencies and performance bottlenecks. To mitigate computational and storage overheads, most tracing frameworks adopt a uniform sampling strategy, which inevitably captures overlapping and redundant information. More advanced methods employ learning-based approaches to bias the sampling toward more informative traces. However, existing methods fall short of considering the high-dimensional and dynamic nature of trace data, which is essential for the production deployment of trace sampling. To address these practical challenges, in this paper we present TRACEMESH, a scalable and streaming sampler for distributed traces. TRACEMESH employs Locality-Sensitivity Hashing (LSH) to improve sampling efficiency by projecting traces into a low-dimensional space while preserving their similarity. In this process, TRACEMESH accommodates previously unseen trace features in a unified and streamlined way. Subsequently, TRACEMESH samples traces through evolving clustering, which dynamically adjusts the sampling decision to avoid over-sampling of recurring traces. The proposed method is evaluated with trace data collected from both open-source microservice benchmarks and production service systems. Experimental results demonstrate that TRACEMESH outperforms state-of-the-art methods by a significant margin in both sampling accuracy and efficiency.**

***Index Terms*—Distributed Tracing, Trace Sampling, Cloud Service Monitoring**

I. INTRODUCTION

Modern cloud systems have dramatically shifted the paradigm of software architecture by adopting loosely coupled designs for applications and services. For example, Uber's architecture is composed of several thousands of microservices [1], and WeChat system hosts more than 3,000 services to manage billions of daily requests [2]. While such modularity design brings the benefit of flexibility and scalability, it also necessitates sophisticated monitoring to navigate the inherent complexities of distributed systems. As such, distributed tracing has rapidly emerged as an essential management tool in cloud systems [3], [4]. This is primarily due to its ability to provide a detailed timeline of a request's journey through a system, enabling developers to identify bottlenecks, latency issues, and other performance anomalies.

In cloud service systems, unusual and edge-case system behaviors are rare by definition, such as tail latency. To maintain high coverage of outlier system events, substantial trace data may be generated in production systems, resulting in significant overhead and costs related to trace generation, collection, and ingestion. For example, Google is estimated to generate approximately 1,000 TB of raw traces on a daily basis [5]. State-of-the-art tracing frameworks, such as Jaeger [6] and Zipkin [7], mitigate this overhead by *head-based sampling*, which sets a small sampling rate (*e.g.*, 0.1% [5]) to collect traces. Since head-based sampling occurs prior to request execution, the sampling decision is made uniformly at random. Consequently, the sampled traces contain mostly common-case execution paths, with a lot of overlapping and redundant information. As an alternative, *tail-based sampling* captures traces for all requests, and decides whether to retain a trace after the trace has been generated. Tail-based sampling schemes allow *biased sampling* to collect more informative and uncommon traces by considering details such as latency and HTTP status code.

To pursue more effective biased sampling, some learning-based approaches have been proposed [1], [8]–[11]. They employ machine learning techniques to derive a unique feature representation for traces, which is then utilized to automatically distinguish useful traces from normal ones. However, while progress has been made, some practical challenges remain unaddressed in this field. In production environments, traces vary significantly in their characteristics due to the complex and dynamic interactions between services. Therefore, their feature space can be extremely large, particularly when considering both the structural and temporal features [10]. This could potentially lead to the issue known as the *curse of dimensionality*, compromising the performance and scalability of trace sampling analysis. Moreover, the emergence of new features in online scenarios poses a significant challenge to the model's *adaptability*. Existing methods [10], [12] propose heuristic rules to periodically eliminate features deemed irrelevant to the current window of trace data. In addition, they append new dimensions to feature vectors and modify the model structure accordingly to accommodate the unseen features. These strategies, however, can incur substantial computational overhead and the resulting model may be sub-optimal, especially with the frequent emergence of new features.

To address these practical challenges, in this paper we propose TRACEMESH. It aims to sample uncommon traces for distributed systems in a scalable and streaming way, while trying to maintain a low storage budget. TRACEMESH leverages the technique of *streaming Locality-Sensitive Hashing*

[¶]Corresponding author.

(LSH) [13]–[15] to enable efficient similarity computation between traces. This is done by projecting high-dimensional trace data into a low-dimensional space while preserving their similarity. In this process, new trace features can be seamlessly incorporated without affecting the dimensionality of the input vectors. To sample uncommon traces, TRACEMESH groups evolving trace data into meaningful clusters [16], where traces exhibiting significant deviations or variances will be identified and selected. In this process, TRACEMESH dynamically adjusts the classification of traces (*e.g.*, from uncommon to common) to prevent accumulating redundant information. Experimental results on trace data collected from two open-source microservice benchmarks and one production cloud system demonstrate that TRACEMESH can sample uncommon traces more effectively and efficiently than existing methods.

The major contributions of this work are as follows:

- We propose TRACEMESH, a tail-based trace sampler for cloud service systems, which addresses some practical challenges in this field. Specifically, TRACEMESH performs dimension reduction on trace data to mitigate the efficiency issue raised by the high-dimensional nature of traces. It can also adapt seamlessly to new trace features that emerge in streaming scenarios, without changing the input dimensionality or model structure. The implementation of TRACEMESH is publicly available[*].
- We conduct experiments with trace data collected from open-source benchmark microservices as well as production cloud systems. The experimental results demonstrate the effectiveness and efficiency of TRACEMESH over existing baseline methods.

The remainder of the paper is organized as follows. Section II introduces the background of distributed tracing and the problem statement of this work. Section III describes the proposed methodology. Section IV presents the experiments and experimental results. Section V discusses the related work. Finally, Section VI concludes this work.

II. BACKGROUND

A. Distributed Traces and Their Sampling

Distributed tracing provides a detailed end-to-end view of requests as they traverse complex, multi-tier cloud service systems. Representative open-source distributed tracing tools include Jaeger [6], Zipkin [7], SkyWalking [17], and Lightstep [18]. According to the specifications of OpenTracing [19], a trace is a directed acyclic graph including multiple spans which represent the individual units of work done in a distributed system. Each trace is assigned a unique trace ID upon the initiation of a request, referred to as the *root span*. This trace ID is then propagated to subsequent child spans, which serves as a crucial identifier for the construction of a complete, cohesive trace. Each span encapsulates various attributes, including trace ID, span ID, parent span ID, latency, and additional metadata (*e.g.*, IP address or service version). This enables us to understand the performance characteristics, locate problems, and optimize the system. Distributed tracing tools operate within live production environments, involving trace transmission, processing, and storage, which inevitably causes significant computational and storage overheads. For example, WeChat could produce dozens of terabytes of trace data daily [10]. Therefore, sampling has emerged as a prevailing approach to reduce these tracing overheads. To ensure the utility of the captured data, sampling decisions are coherent per request, *i.e.*, a trace is either sampled in its entirety, recording the complete end-to-end execution, or not at all [9].

The feasibility of trace sampling is based on the fact that cloud service systems operate under normal conditions most of the time, *e.g.*, many services guarantee an SLA of over 99.9% [20]. Thus, useful traces only manifest in a small fraction of requests, which trace sampling aims to capture and persist. We would like to emphasize that a useful trace is not exclusively one associated with performance issues. It can also record a normal request execution that triggers a previously unseen service call graph. The objective of trace sampling is to identify and discard traces that contain repetitive or redundant system execution information. As mentioned in Section I, there are two generic strategies for trace sampling, namely head sampling and tail sampling. Head sampling makes decision at the beginning of a request, which, while useful for curbing overhead, cannot know a priori whether a request will carry interesting information and should be traced. Such a random decision-making process tends to miss important and minor traces such as tail-latency traces. In contrast, tail sampling executes after traces have been generated. It pays the runtime costs of generating and caching trace data, but in return allows more flexible and biased sampling since the execution results (*e.g.*, latency, execution graph) become available.

Based on the idea of tail sampling, some learning-based approaches have been proposed. They exploit machine learning techniques to automatically analyze and predict the significance of a trace, without explicit feature engineering [9], [21]. In this process, two types of features are often employed as key indicators of traces' commonness, namely *structural information* (*i.e.*, calling path, span depth) and *temporal information* (*i.e.*, span duration). For example, a special input might trigger an unusual service execution path, or early interruptions could result in incomplete traces. These scenarios can all be characterized by the structural information of a trace. On the other hand, even if a trace maintains a usual structure, it may not necessarily indicate normal operation, since the request could experience significant latency. These two types of features collectively provide a comprehensive picture of traces, which facilitates a more accurate trace analysis.

B. Problem Statement

The goal of this work is to sample useful traces for cloud service systems to mitigate the computational and storage overheads related to trace processing. Given a sampling budget (*e.g.*, 1%), we try to identify and capture *uncommon traces* in a streaming scenario, where traces are continuously being generated. We are interested in two types of traces. The first

[*]https://github.com/OpsPAI/TraceMesh

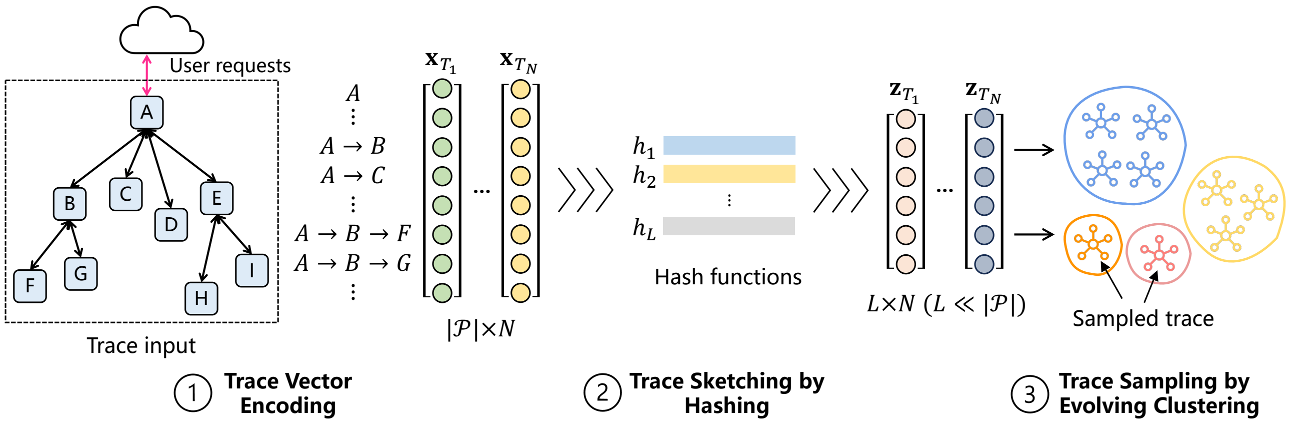

Fig. 1. The overall framework of TRACEMESH

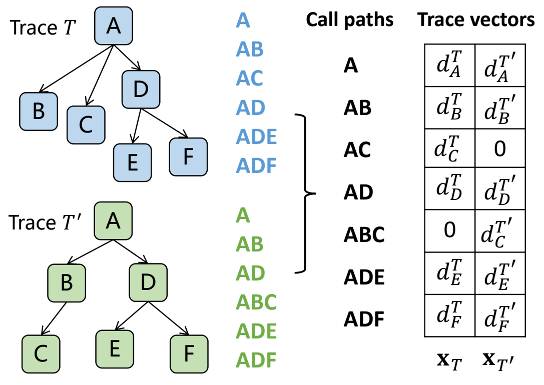

Fig. 2. Trace vector encoding

type is the unusual traces that record service performance outliers or system failures. The second type of trace is normal but corresponds to new service operations that haven't been observed in recent periods. In this process, we try to improve the efficiency of trace sampling by leveraging the idea of dimension reduction. We also accommodate new features that could emerge in stream data. These are two essential practical challenges for the deployment of trace sampling techniques in production cloud systems.

III. METHODOLOGY

In this section, we present the design of TRACEMESH. The overall framework is illustrated in Figure 1, which consists of three phases, namely, *Trace Vector Encoding*, *Trace Sketching by Hashing*, and *Trace Sampling by Evolving Clustering*. In the first phase, TRACEMESH takes graph traces as input, and encodes them as feature vectors by considering both the structural and temporal information. In the next phase, TRACEMESH employs streaming LSH to transform the raw trace vectors into sketch vectors, which has a much lower dimensionality and can adapt to unseen features. In the last phase, TRACEMESH performs streaming trace sampling through evolving clustering, which adjusts the sampling decision on-the-fly to achieve more accurate results.

A. Trace Vector Encoding

Traces are provided in a graph format. Encoding traces into numerical vectors serves as a prerequisite step for many learning-based methods in trace analysis. According to Section II, a trace's structural information and temporal information are two essential indicators of its uncommonness. Therefore, the encoding process should be able to capture and represent these two aspects in an effective and interpretable way. To this end, we perform trace vector encoding [10], [22], as demonstrated in Figure 2. A trace records the execution trajectory of a service request. The initial span of the trace corresponds to the entry point of the request, *i.e.*, the root span. For trace T, we start from the first span A, and traverse through it in Breadth First Search (BFS) order to visit each of its child spans. Once a new span (including the first span) is reached, we record the path from the first span to it, and refer to it as a *call path*. For example, after navigating all spans in trace T, we get call paths $A, A \rightarrow B, \ldots, A \rightarrow D \rightarrow F$. Similarly, for trace T', we can extract a new call path $A \rightarrow B \rightarrow C$. The set of call paths produced after processing all traces is denoted as $\mathcal{P}$, which will be used to construct the entries of the trace vector. In $\mathcal{P}$, call paths with the same length will be sorted lexicographically for conciseness. We use duration d associated with the tail span of each call path as the value of the corresponding vector entry. For example, the entries AB and ADE of trace T (we omit the symbol $\rightarrow$) have value d_B^T and d_E^T, respectively. In particular, we apply a log transformation, $\lfloor log_{10} d \rfloor$, to normalize the duration. In reality, minor variations in duration is not practically significant to affect the overall trace pattern. This allows us to concentrate on the most significant part of the duration, thereby enhancing the stability of trace similarity mining (Section III-B). Different traces may encompass different sets of call paths. For those call paths that are not included within a particular trace (*e.g.*, path ABC in trace T), their corresponding entries will be assigned a value of 0. The final trace vector of trace T is represented as $\mathbf{x}_T = (d_1, d_2, \ldots, d_{|\mathcal{P}|})$, where $|\mathcal{P}|$ is the size of the call path set $\mathcal{P}$.

The trace vectors encoded in this way are capable of distinguishing trace samples with infrequent structural and/or temporal features. Specifically, if a trace has unique call paths, the resulting vector values will be non-zero, indicating the presence of its uncommon structure. For two traces that share an identical structure but exhibit significantly different duration statuses, their trace vectors will also differ in the temporal aspect. The comparison of these aspects can be accomplished by measuring the similarity between trace vectors. Beyond the path and duration features, TRACEMESH can also incorporate more meaningful features into the trace vectors to further differentiate uncommon traces. For example, the request status code or service events can be embedded into the call paths to form more informative trace vectors.

B. Trace Sketching by Hashing

While the vector encoding technique in Section III-A can capture abundant information from a trace, we encounter challenges related to *adaptability* and *dimensionality*. In production cloud services, the generated traces could encompass a wide variety of span types and exhibit significant length. Thus, it requires knowing the size of the complete call path set $\mathcal{P}$ to specify the dimension of each trace vector. With new call paths continuously being formed from the new trace types arriving in stream, the full call path set (and hence its size) always remains undetermined. Even if the universal call path set is fixed, its size, *i.e.*, $|\mathcal{P}|$, can get prohibitively large. This could potentially lead to the issue known as *the curse of dimensionality*, which could severely impair the performance of downstream trace analysis methods.

To enhance a model's adaptability to previously unseen features, one straightforward way is to adjust the dimensions and then retrain the model. This is however not scalable, especially when the feature space is dynamically evolving. Existing methods resort to tree-based models to incorporate new dimensions. For example, when there is a trace containing paths that never appear before, Sieve [10] extends the dimension of other traces by appending -1 to their vectors. The tree structure is subsequently modified by creating a new root. While this design does accommodate the introduction of new paths, the resulting tree structure may not be optimal. Therefore, we need a more streamlined approach to handling new paths. To mitigate the dimensionality problem, prior work [10], [12] has touched upon the idea of dimension reduction. However, this is achieved by discarding features irrelevant to the current window of trace data, rather than eliminating those that are unimportant. The discarded features could be those that are introduced in earlier windows to accommodate unseen call paths. In such a design, certain features can be added and removed for multiple times. This is not only inefficient, but could damage the model's structure if the number of new dimensions is large or if they manifest frequently.

To address these challenges, we propose to leverage streaming Locality-Sensitive Hashing (LSH) [13], [14] for efficient and online trace sampling. An LSH scheme enables efficient similarity computation by projecting high-dimensional vectors into a low-dimensional space while preserving their similarity. For instance, by doing so, SIMHASH [14] can quickly measure the cosine similarity between real-valued vectors. In order to use the SIMHASH in the streaming setting, Manzoor et al. [15] further proposed STREAMHASH, which is employed in this paper to accommodate unseen call paths. Details are introduced below.

Given input trace vectors in $\mathbb{R}^{|\mathcal{P}|}$, STREAMHASH is first instantiated with L projection vectors $\mathbf{r}_1, \dots, \mathbf{r}_L \in \{+1, -1\}^{|\mathcal{P}|}$. Each element of $\mathbf{r}_l$, $l = 1, \dots, L$ is drawn uniformly from $\{+1, -1\}$. The LSH $h_{\mathbf{r}_l}(\mathbf{x})$ of an input trace vector $\mathbf{x}$ for a given random projection vector $\mathbf{r}_l$ is defined as follows:

$$h_{\mathbf{r}_l}(\mathbf{x}) = \begin{cases} +1 & \text{if } \mathbf{x} \cdot \mathbf{r}_l \geq 0 \\ -1 & \text{if } \mathbf{x} \cdot \mathbf{r}_l < 0 \end{cases} \tag{1}$$

That is, $h_{\mathbf{r}_l}(\mathbf{x}) = sign(\mathbf{x} \cdot \mathbf{r}_l)$. In particular, $h_{\mathbf{r}_l}(\mathbf{x})$ possesses the following nice property: the probability (over vectors $\mathbf{r}_1, \dots, \mathbf{r}_L$) of any pair of input vectors $\mathbf{x}_T$ and $\mathbf{x}_{T'}$ hashing to the same value is proportional to their cosine similarity:

$$\mathbf{Pr}_{l=1,\dots,L}[h_{\mathbf{r}_l}(\mathbf{x}_T) = h_{\mathbf{r}_l}(\mathbf{x}_{T'})] = 1 - \frac{cos^{-1}(\frac{\mathbf{x}_T \cdot \mathbf{x}_{T'}}{\|\mathbf{x}_T\| \|\mathbf{x}_{T'}\|})}{\pi} \tag{2}$$

Since the computation of similarity between two traces now requires only these hash values, it is feasible to substitute each $|\mathcal{P}|$-dimensional input vector $\mathbf{x}$ with a more concise, L-dimensional *trace sketch vector* $\mathbf{z}$, which encapsulates its LSH values, *i.e.*, $\mathbf{z} = [h_{\mathbf{r}_1}(\mathbf{x}), \dots, h_{\mathbf{r}_L}(\mathbf{x})]$. In this case, each sketch vector can be succinctly represented using just L bits, where each bit corresponds to a value in $\{+1, -1\}$. This allows for highly memory-efficient storage algorithms, reducing the complexity of the data structure without compromising its integrity.

The similarity between two input vectors can be subsequently estimated by empirically evaluating the probability presented in Equation (2). This is done by determining the proportion of aligned hash values when the input vectors are hashed with L random vectors. As such, the similarity computation between two trace vectors is transformed into a process of quantifying the level of agreement between the hash values.

$$sim(T, T') \propto \frac{|\{l : \mathbf{z}_T(l) = \mathbf{z}_{T'}(l)\}|}{L} \tag{3}$$

In summary, given a target dimensionality $L \ll |\mathcal{P}|$, each trace $\mathbf{x}$ can be represented by a sketch vector $\mathbf{z}$ of dimension L, allowing us to discard the original $|\mathcal{P}|$-dimensional trace path vectors and compute similarities within this newly-defined vector space.

While the above operation can effectively mitigate the dimensionality challenge, the adaptability issue remains. As shown in Figure 3, suppose a new trace T'' comes with a new call path ADG having a duration $d_G^{T''}$. In this case, many existing studies append a new dimension (in our example, a "0") to all the other trace vectors and update the model accordingly. This could incur significant computational overhead,

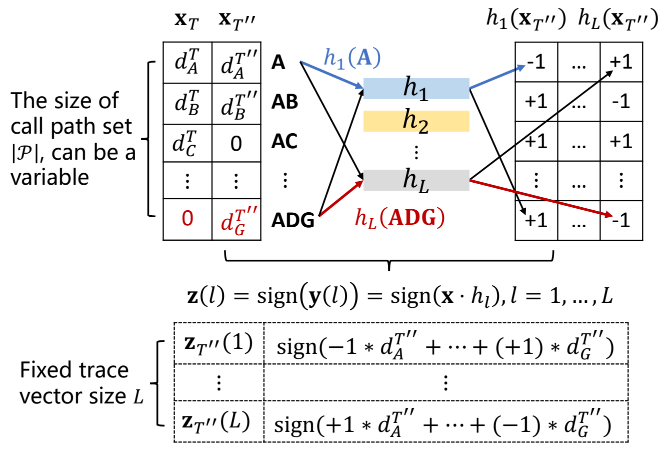
$\mathbf{x}_T$
$\mathbf{x}_{T''}$
$h_1(\mathbf{x}_{T''})$
$h_L(\mathbf{x}_{T''})$
The size of call path set $|\mathcal{P}|$, can be a variable
A
AB
AC
ADG
$h_1(\mathbf{A})$
$h_L(\mathbf{ADG})$
h_1
h_2
h_L
$\mathbf{z}(l) = \mathrm{sign}(\mathbf{y}(l)) = \mathrm{sign}(\mathbf{x}\cdot h_l), l = 1, \ldots, L$
Fixed trace vector size L
$\mathbf{z}_{T''}(1)$
$\mathrm{sign}(-1 * d_A^{T''} + \cdots + (+1) * d_G^{T''})$
$\mathbf{z}_{T''}(L)$
$\mathrm{sign}(+1 * d_A^{T''} + \cdots + (-1) * d_G^{T''})$

Fig. 3. Streaming trace vector encoding

and the resulting model may be sub-optimal. To address this problem, instead of using the L projection vectors $\mathbf{r}_1, \ldots, \mathbf{r}_L \in \{+1, -1\}^{|\mathcal{P}|}$ (whose dimensionality is fixed), StreamHash instantiates L hash functions $h_1, \ldots, h_L$ picked uniformly at random from a family $\mathcal{H}$ of hash functions, mapping call paths to {+1,-1}. As shown in the top part of Figure 3, a $h_l \in \mathcal{H}$, $l = 1, \ldots, L$, is a deterministic function that maps a given call path to either +1 or -1, which replaces the original elements in the projection vectors $\mathbf{r}_i$. To obtain the trace sketch, we first construct the projection vector $\mathbf{y}$ of the trace as:

$$\mathbf{y}(l) = \mathbf{x} \cdot h_l = \sum_{i=1,\ldots,|P|} \mathbf{x}(i) h_l(p_i) \tag{4}$$

Then, the L-bit trace sketch for each input vector $\mathbf{x}$ under the hash functions can be calculated by $\mathbf{z} = sign(\mathbf{y})$, which can be used to measure the similarity between traces. In this way, the adaptability issue is also addressed as the trace sketch vectors can be constructed and maintained incrementally, accommodating call paths that are not previously observed. As a result, we eliminate the need to know the complete call path set $\mathcal{P}$, or to maintain $|\mathcal{P}|$-dimensional random vectors $\mathbf{r}_i$ in memory.

In [15], the strongly universal multilinear family [23] is adopted as the hash function family $\mathcal{H}$ for string data. As our trace vector encoding also produces string call paths, we employ the same configuration. In this family, the input call path p is divided into $|p|$ components (*e.g.*, spans) as $p = s_1 s_2 \ldots s_{|p|}$. A hash function h_l is constructed by first choosing $|p|$ random numbers $m_1^{(l)}, \ldots, m_{|p|}^{(l)}$, and p is then hashed as follows:

$$h_l(p) = 2 \times ((m_1^{(l)} + \sum_{i=2}^{|p|} m_i^{(l)} \times int(s_i)) \ mod \ 2) - 1 \tag{5}$$

where $int(s_i)$ is a function that maps span s_i to a unique integer and $h_l(p) \in \{+1, -1\}$. We start from one with an incremental growth of one for new span types. The hash value for a call path of length $|p|$ can be computed in $\Theta(|p|)$ time.

Each hash function is represented by $|p|_{max}$ random numbers, where $|p|_{max}$ denotes the maximum possible length of a call path. These numbers remain constant per hash function h_l, which serve as the parameters. Thus, the hash functions can deterministically hash a given call path to the same value each time. In practice, these L hash functions can be generated uniformly at random from this family by creating a matrix of $L \times |p|_{max}$ uniformly random 64-bit integers using a pseudorandom number generator. While we still need some global information, *i.e.*, $|p|_{max}$, this is a much lighter restriction. Existing work needs to know a priori the concrete type of all call paths. In cases where a new call path has a length exceeding $|p|_{max}$, we can break it into larger chunks (e.g., two spans constitute one component) [15].

C. Trace Sampling by Evolving Clustering

In real-world systems, most traces share similar and common characteristics. Trace sampling aims to identify the uncommon traces that exhibit rare patterns and ensure that they are sampled with a higher probability. In Section III-B, based on the streaming LSH technique, we are able to encode new traces with arbitrary call paths for similarity computation. Next, TraceMesh conducts trace sampling by grouping them into meaningful clusters, where each cluster contains similar traces in both structural and temporal perspectives. The uncommon traces can then be identified based on the deviations from these established clusters. In production systems, traces are continuously being generated, *i.e.*, streaming data. During this process, uncommon traces exhibiting unprecedented patterns can emerge. Thus, our clustering approach should not only recognize the new unusual patters, but also allow the false positives (*i.e.*, the new usual patterns) to eventually evolve into normal clusters and stop sampling them. This strategy ensures a more accurate and efficient trace sampling.

To this end, TraceMesh adapts DenStream [16] to better fit our task of trace stream clustering. DenStream is a popular density-based clustering approach for evolving stream data. Without the assumption on the number of clusters, it can discover clusters with arbitrary shape and handle outliers. In DenStream, each data point is associated with a weight, which decreases exponentially with time t via a fading function $f(t) = 2^{-\lambda \cdot t}$, where $\lambda > 0$ is a decay factor. Since there is no global information about data streams, DenStream resorts to the idea of micro-clusters [24] (*i.e.*, local stream information) to approximate the precise result in a streaming environment. Each micro-cluster possesses three attributes: a weight w indicating its commonness, which is determined based on the number and weight of points in it; a center c, which is the weighted center of the points in the micro-cluster; and a radius r, which is the weighted average of the distance from the points in the micro-cluster to the center.

Given the dynamic nature of evolving data streams, the roles of outliers and clusters are often exchanged. Consequently, new clusters may emerge, and old ones gradually fade out. To accommodate these continuous shifts, DenStream introduces two additional types of micro-clusters by setting different constraints on the weight, *i.e.*, Potential Micro Cluster (PMC) and Outlier Micro Cluster (OMC). The PMC contains frequent and usual data points, while the OMC holds the points that could potentially be outliers or the seed of a new PMC (*i.e.*, previously unseen but new normal data points). Thus, PMCs will have a larger weight than OMCs. As time progresses, the number of data points within a cluster, along with their respective weights, will change. This will in turn influence the overall weight of the cluster. Based on such evolving weights, DenStream dynamically modifies the role of clusters.

Following the idea of DenStream, TRACEMESH performs a continuous process to discover micro-clusters in streaming trace data and alter their role for trace sampling. Before entering the online clustering process, TRACEMESH first applies the DBSCAN algorithm [25] on the training data to generate the initial trace clusters. In reality, the training data can be collected during the system's fault-free phases, which are easily obtainable since production services are mostly running in normal status. The initial trace clusters represent the prevalent trace types within the system and act as a baseline to identify traces that deviate from these typical patterns. When a new trace T arrives, TRACEMESH tries to merge it into existing micro-clusters as follows.

1) At first, TRACEMESH attempts to merge T into its nearest PMC C_p, and calculates the new radius of C_p. If C_p's new radius is below or equal to a predefined threshold ϵ, which means the new trace T is indeed similar to the members in C_p (and thus a usual trace), the merge is successful. Next, the attributes of C_p will be updated as follows. The weight w is updated based on the formula $w_p^* = w_p \times 2^{-\lambda} + 1$, where w_p^* is the updated weight, w_p is the previous weight, and λ is the decay factor. The center c is updated based on the formula $c_p^* = (c_p \times w_p \times 2^{-\lambda} + v_T)/w_p^*$, where c_p^* is the updated center, c_p is the previous center, and v_T is the vector representation of trace T. Recall that the weight of a PMC is determined not only by the quantity of points it contains, but also by the individual weight of these points. The weight of C_p increases with a larger number of points and with the recency of these points (as newer points have a larger weight). However, these two factors inversely affect the sampling probability of the traces in C_p. This is because we are more interested in the rare traces and those that haven't appeared recently. Therefore, we calculate the sampling probability p_T for T, which is inversely proportional to the weight of C_p. We multiply the probability by the sampling budget $\mathcal{B}$ to meet the storage requirement:
$$p_T = \mathcal{B} \times (1 - \frac{w_p^*}{\sum_{i=0}^{N_{pmc}} w^{(i)}}) \tag{6}$$
where N_{pmc} is the total number of existing PMCs and $w^{(i)}$ is the weight of the i-th PMC.
2) Else, if merging into the nearest PMC is unsuccessful, TRACEMESH will try to merge T into the nearest OMC C_o. Similarly, if the new radius of C_o is below the predefined threshold ϵ, the merge will be kept and the attributes of C_o will be updated as in the first case. Then, TRACEMESH checks whether the new weight of C_o is above a noise threshold α, which is the weight constraint for defining a PMC. If this is the case, it means C_o has evolved into a PMC, *i.e.*, its trace pattern is deemed common. Thus, TRACEMESH will switch its role accordingly. In this step, T will be sampled no matter whether C_o's role will be switched. This is because T is dissimilar to the usual traces recorded in existing PMCs, but more similar to the traces in C_o that are relative rare.
3) Otherwise, if the previous two merging attempts fail, TRACEMESH creates a new OMC containing only T. The weight and center of this new OMC are 1 and T, respectively. In this case, T will be sampled, because it carries an unprecedented trace pattern that is not similar to any of the existing clusters. Note that the sampling of uncommon traces in this step (and step two) is not bound by the sampling budget. This is due to their rarity and inherent value, providing crucial insights into a system's edge-case behaviors.

For each existing PMC, its weight will gradually decay if it fails to incorporate any new points. If the weight is below the threshold α, it means that the PMC has degraded back to an OMC, and it will be removed. This is because the trace pattern represented by this PMC is considered expired, given no traces of this type have emerged for a certain period of time. This could be attributed to system updates or the change of user behaviors, both of which could lead to the disappearance of certain types of traces. In this way, we can maintain the latest trace patterns for online sampling. To identify and remove the old PMCs, TRACEMESH checks their weight periodically. The time interval is calculated by the following formula:
$$T_p = \lceil \frac{1}{\lambda} \log(\frac{\alpha}{\alpha - 1}) \rceil \tag{7}$$
In DenStream, the weight of existing OMCs also needs to be checked periodically. OMCs falling below a certain weight limit are classified as outliers and subsequently removed. However, this operation is not necessary in TRACEMESH. Our objective is to sample uncommon traces, which should include not just the outliers, but also the ones that represent normal system executions that have never appeared. This is achieved by sampling traces associated with OMCs in both step two and three. As the data stream proceeds, if an OMC indeed grows into a PMC, we will stop sampling from it. Thus, there is no need for OMCs to undergo periodic checks or premature disposal, which saves computational overheads. In this way, TRACEMESH can identify diverse trace types while simultaneously preventing the over-sampling of recurring traces.

D. Complexity Analysis

TRACEMESH performs two main operations to determine whether a trace should be sampled, *i.e.*, computing its sketch vector and merging it to micro-clusters. In the worst case, the vector size of the trace is $|\mathcal{P}|$ (*i.e.*, it comprises all unique call paths) and the length of all its call paths is equal to $|p|_{max}$. Then, computing its L-dimensional sketch vector requires a time complexity of $\mathcal{O}(|\mathcal{P}| \times L \times |p|_{max})$, which can be accelerated via matrix multiplication. In reality, the complexity of a trace is likely to be much less than this worst-case scenario. As for the merging part, the worst case is that the trace fails to join any of the existing micro-clusters. The time complexity in this case is $\mathcal{O}((N_{pmc} + N_{omc}) \times L)$, where N_{omc} is the number of OMCs. Since TRACEMESH periodically fades out expired clusters, the total number of PMCs and OMCs remains small. Therefore, we can conclude that the time complexity of TRACEMESH is relatively low, rendering it a scalable trace sampler.

IV. EVALUATION

In this section, we present the evaluation of TRACEMESH. We first introduce the experimental settings, including the datasets, the metrics for evaluation, and the baseline methods. Next, we demonstrate the experimental results, which include the effectiveness of trace sampling, the efficiency, and the sensitivity study of some important parameters.

The default parameter configurations in our experiments are as follows: the size of random numbers in each hash function $|p|_{max} = 64$, the sketch length $L = 100$, the threshold for declaring that a trace is close enough to a micro-cluster $\epsilon = 0.01$. All experiments are conducted on a Linux server with an Intel(R) Xeon(R) Gold 6226R CPU and 256GB RAM. We repeat all experiments five times, which are averaged to yield the final results.

A. Experimental Settings

1) Datasets: We collect traces from two open-source benchmark microservice applications, *i.e.*, Train Ticket [3] and Online Boutique [26], which have been widely used in previous studies [27]–[29]. Train Ticket is a railway ticketing system with 41 microservices, where users can search, book, pay, and cancel train tickets. Online Boutique is a web-based e-commerce app with 11 microservices, where users can browse items, add them to the cart, and make a payment. These benchmarks are implemented in different programming languages such as Java, Go, Node.js, Python, etc. We follow [27] to deploy these applications and inject faults to generate abnormal traces that correspond to edge-case system behaviors. For each application, we synthesize workloads via the load testing tool Locust [30] to simulate user requests. We label two types of uncommon traces that are deemed necessary for sampling. The first is the abnormal traces generated during fault injection, and the second is those corresponding to new service operations that are not previously observed in the training data.

Besides the benchmark applications, we also evaluate TRACEMESH on a production dataset, which comprises hundreds of thousands of traces from large-scale cloud service systems. These services run within containers that are directly

TABLE I
DATASET STATISTICS

Dataset	#Span	#Train	#Test	#Label
Train Ticket	201	653	31,814	624
Online Boutique	43	1,024	78,931	225
Industry	1,308	11,847	497,439	2,453

managed by Kubernetes [31]. The traces to be sampled are labeled by on-site engineers based on their domain knowledge. Table I summarizes the statistics of the three trace datasets. *#Span* denotes the number of different span types in the dataset. We split each dataset into training and testing sets, whose size is given by *#Train* and *#Test*. In particular, as our goal is to design a streaming trace sampler, we keep a low ratio for the training set in order to effectively evaluate TRACEMESH in real-world conditions. The traces in the testing set are gradually fed into the model based on their timestamps to simulate an online scenario. Finally, *#Label* gives the number of labeled traces, which we aim to sample.

2) Evaluation Metrics: Intuitively, there are two essential aspects to gauge the effectiveness of a trace sampling algorithm. The first is whether the sampler can successfully capture the useful traces. We refer to this metric as *Coverage*, which can be calculated as the ratio of labeled traces sampled by the algorithm to the total number of labeled traces. The second is at what *Sampling Rate* the coverage is accomplished. This metric can be calculated as the ratio of the number of traces sampled to the total number of traces available in the testing set. An ideal trace sampler should be able to capture a broad spectrum of useful traces (*i.e.*, a high Coverage), while maintaining resource efficiency (*i.e.*, a low Sampling Rate). In reality, missing important traces could lead to their permanent loss, particularly in online scenarios. As a result, engineers may lose valuable insights into the functioning and performance of the system. Thus, the Coverage should generally be prioritized over the Sampling Rate.

3) Baseline Methods: The following methods are selected for a comparative evaluation of TRACEMESH. The default sampling budget for all methods is set as 1%, except for Sieve which does not require this parameter.

- *Uniform*: This strategy uniformly samples a trace subset from the testing set, without considering any specific characteristics of the traces. It is the default sampling mechanism used in many distributed tracing tools, *i.e.*, the head sampling.
- *Sieve*: Sieve [10] is an online trace sampler that leverages an attention mechanism to bias sampling toward uncommon traces. It uses the technique of Robust Random Cut Forest (RRCF), which is a variant of the Isolation Forest, to calculate an attention score for traces. Uncommon traces tend to have a shallower depth from the root to leaf, and thus will receive more attention (*i.e.*, a higher probability) for sampling. New dimensions are integrated

into the model by appending them to the feature vectors, and the tree structure is subsequently modified to align with these additions.

- *Sifter*: Sifter [9] captures edge-case traces by learning an unbiased, low-dimensional model based on fixed-length trace sub-paths. Such a model is able to approximate the system's common-case behaviors. Thus, by measuring the reconstruction loss of an incoming trace, the sampling decisions can be made toward traces that lead to a large loss. Particularly, Sifter only considers the structural feature of traces, and is thus insensitive to temporal deviations.
- *SampleHST*: Based on the Bag-of-Words (BoW) representation of traces, SampleHST [11] calculates a distribution of the mass values obtained from a forest of tree-based classifier, *i.e.*, Half Space Trees (HSTs). The mass distribution derived from the HSTs is then utilized to cluster the traces online, leveraging a variant of the mean-shift algorithm. A trace is more likely to be sampled if it is associated with a low-mass-value cluster. Similar to Sifter, the BoW representation makes SampleHST unable to consider the temporal feature of traces.
- *Perch*: Perch [8] represents traces based on various graph-based features, *e.g.*, occurrence count embeddings based on user-specified events. Then, a hierarchical clustering algorithm named PERCH is utilized to groups traces. Representative traces are then evenly selected from each trace group.

B. Experimental Results

1) Trace Sampling: Table II shows the performance of trace sampling of different methods on three datasets, including both the coverage and sampling rate. We can see that TRACEMESH achieves the best coverage score on all datasets. The score on the Industry dataset is comparatively lower due to its scale and complexity. Sieve ranks second in terms of the coverage performance. However, as it lacks explicit consideration of the sampling budget, it has the highest sampling rate in all cases. On the Train Ticket and Online Boutique datasets, TRACEMESH also has a relatively high sampling rate. This is because in our design (Section III-C), rare traces are deemed necessary for sampling regardless of the sampling budget, as they provide essential insights into a system's edge-case behaviors. Nevertheless, TRACEMESH demonstrates the best sampling rate on the Industry dataset. We can impose a harder budget constraint to further lower the sampling rate, *i.e.*, an even smaller $\mathcal{B}$ in Equation (6). It will only decreases the sampling of traces that are relatively common. Uniform sampling achieves the best sampling rate (*i.e.*, 1%), but with the worst coverage, which is also around 1%. The remaining three methods also present a good sampling rate. This can be attributed to their relatively stringent budget constraints, particularly in the case of Perch. However, they fall short in terms of coverage scores. An important reason is that they only consider the structural features of traces. This observation underscores the significance of temporal features.

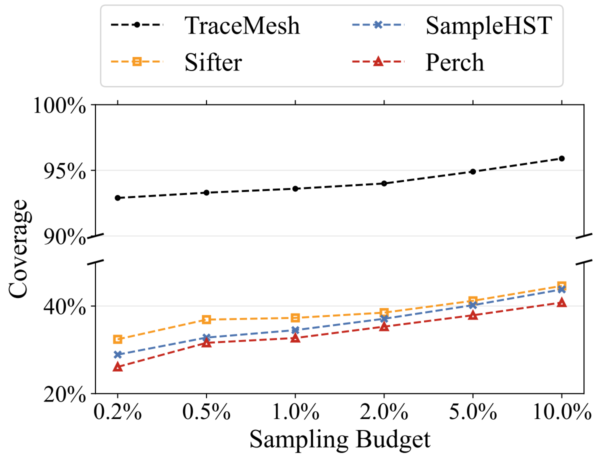

Fig. 4. Coverage with different sampling budgets

In particular, certain approaches *e.g.*, [8], perform representative sampling, where only a few traces are selected from each cluster of similar traces. However, in real-world situations, when performance issues arise, it becomes essential to capture the entire spectrum of edge-case traces regardless of their similarity. This not only provides a more comprehensive understanding of the issue at hand (*e.g.*, the blast radius [32]) but also ensures that the solutions are robust and effective. TRACEMESH mitigates this problem by continuously collecting traces that belong to OMCs, while also allows the dynamic transformation of a trace cluster's role to avoid over-sampling.

We further evaluate the coverage of different methods with various sampling budgets, as shown in Figure 4. We exclude Sieve due to its lack of a budget-control design. As Industry is the most challenging dataset, we only show the results derived from it. Other datasets demonstrate similar results. Clearly, as the sampling budget grows, all methods achieve better coverage scores. In all situations, TRACEMESH outperforms the baselines by a large margin, demonstrating both superior performance and stability. This is because TRACEMESH can automatically identify the most important traces for sampling, even when the sampling budget is exceeded. It is worth noting that, edge cases are rare in production systems, and the storage budget is typically sufficient. The key issue lies in how to accurately identify such valuable traces.

2) Efficiency: In production cloud service systems, the sheer volume and complexity of traces pose significant challenges on the efficiency of trace sampling. Thus, we evaluate different methods in this aspect, as shown in Figure 5. We only present results for the Industrial dataset, which has a significantly higher feature dimension compared to other datasets. The efficiency is quantified by the time taken by each method to complete the sampling process for the Industry dataset. The performance of different methods can be divided into three tiers. The first tier includes SampleHST and our method, taking the minimum time, *i.e.*, 1,218.2 and 1,306.7 seconds respectively. SampleHST employs HSTs to compute

TABLE II
EXPERIMENTAL RESULTS OF TRACE SAMPLING

Method	Train Ticket		Online Boutique		Industry	
	Coverage	**Sampling Rate**	**Coverage**	**Sampling Rate**	**Coverage**	**Sampling Rate**
Uniform	1.1%	**1.00%**	1.1%	1.00%	1.0%	1.00%
Sieve	71.8%	6.24%	82.2%	3.91%	61.1%	3.29%
Sifter	50.6%	1.28%	42.2%	1.13%	37.3%	1.16%
SampleHST	46.2%	1.19%	38.2%	1.07%	34.5%	1.10%
Perch	42.9%	**1.00%**	35.1%	**0.98%**	32.7%	0.95%
TRACEMESH	**98.7%**	2.34%	**100%**	1.35%	**93.6%**	**0.83%**

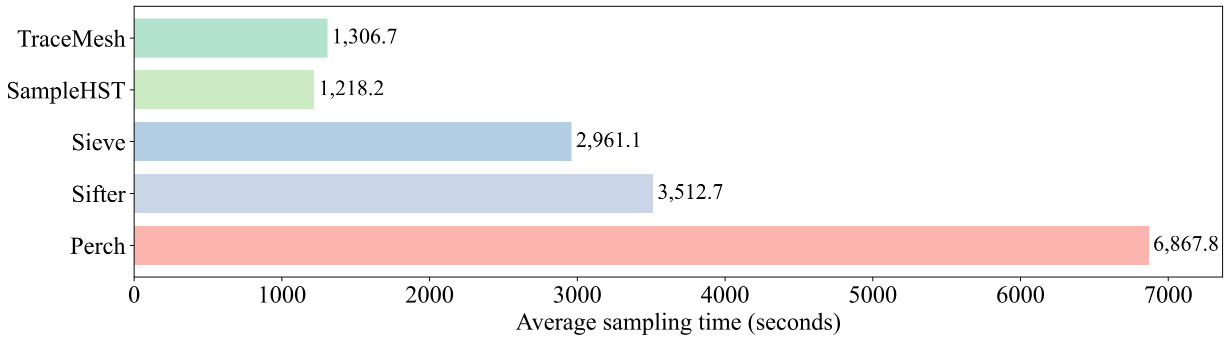

Fig. 5. Efficiency on the Industry dataset

the mass distribution of traces, making it a lightweight model with a constant amortized time complexity. The performance of TRACEMESH can be attributed to its efficient sketching mechanism and evolving clustering design. Particularly, in Figure 3, for the zero entries of the trace vector (*i.e.*, the trace does not have the corresponding call paths), the sketching process can be omitted, thereby further enhancing its efficiency. Sieve and Sifter belong to the next tier, which require a considerably longer duration of 2,961.1 and 3,512.7 seconds respectively. This indicates that the second tier operates nearly three times slower than the first for trace sampling. In online scenario, Sieve needs to adjust its tree structures to accommodate new feature dimensions. This process is expensive, especially when dimension expansion happens frequently. Sifter involves embedding calculation, which can be time-consuming. Perch alone constitutes the third tier, which demands the maximum time, *i.e.*, 6,867.8 seconds. Similar to Sieve, Perch requires continuous tree structure modifications, a process that becomes increasingly slower with the addition of more traces.

3) Sensitivity Analysis: The sensitivity to parameter configurations is paramount for a method to consistently deliver stable performance. We conduct a sensitivity analysis of TRACEMESH with regard to two critical parameters, *i.e.*, the sketch length (L) and epsilon (ϵ), as shown in Figure 6. TRACEMESH employs LSH to efficiently measure the cosine similarity between two trace vectors in an L-dimensional space. A large L can reduce the error of cosine distance approximation but at the cost of increased computational

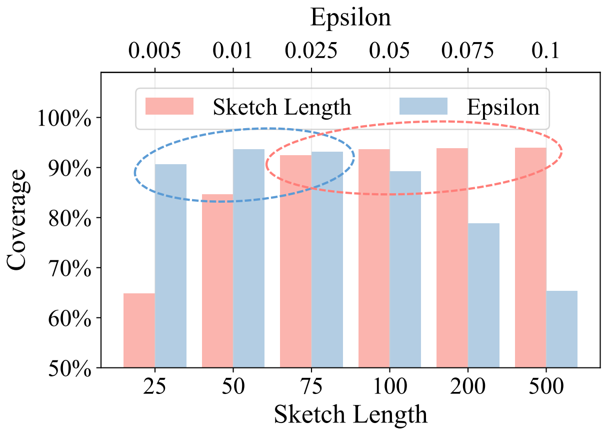

Fig. 6. Parameter sensitivity analysis

overhead. The default setting of L in our experiments is 100. After trying a larger spectrum of settings, *i.e.*, from 25 to 500, we can see that TRACEMESH's performance converges when L is larger than 75. Another parameter, ϵ, is used to determine whether a new trace belongs to a micro-cluster (Section III-C). This is significant because it impacts the evolution process of micro-clusters, such as when an OMC evolves into a PMC. It can be observed that TRACEMESH yields the best performance when ϵ lies within the range of $[0.005, 0.025]$. In light of our findings, TRACEMESH demonstrates a good ability to maintain consistent performance across various parameter settings.

V. Related Work

Distributed tracing has emerged as a crucial tool for understanding and optimizing complex service-oriented architectures. It provides an invaluable mechanism for tracking requests as they navigate through the intertwined services of a distributed system, enabling developers to identify bottlenecks, spot inefficiencies, and troubleshoot performance issues. This area of study has seen significant contributions from various efforts. X-Trace [33] aims to provide a comprehensive view of service behavior across multiple system layers. Dapper [5] is a distributed system tracing infrastructure developed by Google. Its design goals include low overhead, application-level transparency, and ubiquitous deployment across extensive, large-scale systems. Pivot Tracing [34] considers the *happened-before* relations between events during dynamic instrumentation. This provides users with the ability to define arbitrary metrics for mining useful information about root causes at runtime. Canopy [35], developed by Facebook, is an end-to-end performance tracing infrastructure. It captures performance data with causal relationships across various platforms including browsers, mobile applications, and backend services. This aids engineers in querying and analyzing performance data in real-time. DeepFlow [36] is a non-intrusive distributed tracing framework for troubleshooting microservices. It provides out-of-the-box tracing via a network-centric tracing plane and implicit context propagation. In recent years, open-source tracing frameworks like Jaeger [6] and Zipkin [7] have seen widespread adoption in practical applications.

Besides the infrastructure for generating distributed traces, trace data are widely used in various system reliability assurance tasks. The empirical study in [3] shows that the current industrial practices of microservice debugging can be improved through the application of appropriate tracing and visualization techniques. MEPFL [37] leverages system trace logs in the production environment to predict latent errors, faulty microservices, and fault types at runtime. Groot [38] builds a real-time causality graph to facilitate root cause analysis in microservices, which allows adaptive customization of link construction rules to incorporate domain knowledge. TraceStream [12] identifies potential clusters of anomalous traces within evolving trace data and employs spectral analysis to pinpoint the specific anomalous services. Some work [27], [39], [40] utilizes a multi-modal approach, which integrates traces with logs and metrics to provide more comprehensive information about system status for microservice troubleshooting. Traces also serve a crucial role in in analyzing system dependencies [4], critical paths [41], resource characterization [42], [43], and microservice architecture [44], [45]. This enables developers to identify bottlenecks, spot inefficiencies, and subsequently optimize the overall system performance.

With the dramatic increase in trace volume in production systems, trace sampling has become an essential technique to manage data overload and maintain system efficiency. Representative approaches are introduced in Section IV-A3, which leverage a variety of methods, including tree-based models [10], [11], clustering algorithms [8], and neural language techniques [9]. STEAM [1] preserves system observability by sampling mutually dissimilar traces. It employs Graph Neural Networks (GNN) for trace representation [21], and requires human labeling to incorporate domain knowledge. Hindsight [46] introduces the idea of *retroactive sampling*. Instead of eagerly ingesting and processing traces, it lazily retrieves trace data only after symptoms of a problem are detected. Hindsight is based on the observation that the generation of traces at nodes is not expensive, but the ingestion of trace data is resource-intensive. Thus, tail sampling strategy can be applied to achieve accurate trace sampling without occurring too much runtime overhead (*i.e.*, before the processing of trace data).

While progress has been made, existing learning-based approaches for trace sampling lack consideration of practical challenges associated with their deployment in production environments. Specifically, the large volume of trace data and feature space demand a highly efficient solution. The diversity of traces also requires fast ability to accommodate new dimensions. Unlike these works, our design emphasizes operational efficiency and flexibility. It not only handles large volumes of trace data with a vast feature space, but also quickly adapts to the emergence of unprecedented trace features.

VI. Conclusion

In this paper, we propose TraceMesh, a scalable and streaming trace sampler. It addresses the practical challenges of deploying trace sampling techniques in production systems, which have not been adequately addressed in existing work. The scale and complexity of modern service systems render trace data highly diverse and dynamic. The unpredictable system updates and user behaviors further compound the situation. Thus, a practical sampler should be able to handle the vast feature space of traces and quickly generalize to previously unknown features within the trace stream. To this end, we first perform trace vector encoding by considering both the structural and temporal features. Next, we employ streaming Locality-Sensitivity Hashing (LSH) technique to enable efficient trace similarity mining by projecting them into a low-dimensional space. In this process, new features can be seamlessly incorporated without changing the input dimensionality. Finally, given a sampling budget, we identify and capture uncommon traces through evolving clustering. We dynamically adjust the sampling decision to prevent accumulating redundant information. We have evaluated TraceMesh using trace data collected from both open-source microservice benchmarks and production service systems. Experimental results highlight its potential to pave the way for efficient system monitoring.

Acknowledgement

This work is supported by the National Natural Science Foundation of China (No. 62202511) and the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14206921 of the General Research Fund).

References

[1] S. He, B. Feng, L. Li, X. Zhang, Y. Kang, Q. Lin, S. Rajmohan, and D. Zhang, "STEAM: observability-preserving trace sampling," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 1750–1761. [Online]. Available: https://doi.org/10.1145/3611643.3613881

[2] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, "Overload control for scaling wechat microservices," in *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*. ACM, 2018, pp. 149–161. [Online]. Available: https://doi.org/10.1145/3267809.3267823

[3] 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 Trans. Software Eng.*, vol. 47, no. 2, pp. 243–260, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2018.2887384

[4] 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 *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, C. Curino, G. Koutrika, and R. Netravali, Eds. ACM, 2021, pp. 412–426. [Online]. Available: https://doi.org/10.1145/3472883.3487003

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

[6] Jaeger. (2024) An open source, distributed tracing platform. [Online]. Available: https://www.jaegertracing.io/

[7] Zipkin. (2024) A distributed tracing system. [Online]. Available: https://zipkin.io/

[8] P. H. B. Las-Casas, J. Mace, D. O. Guedes, and R. Fonseca, "Weighted sampling of execution traces: Capturing more needles and less hay," in *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*. ACM, 2018, pp. 326–332. [Online]. Available: https://doi.org/10.1145/3267809.3267841

[9] P. H. B. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace, "Sifter: Scalable sampling for distributed traces, without feature engineering," in *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 2019, pp. 312–324. [Online]. Available: https://doi.org/10.1145/3357223.3362736

[10] Z. Huang, P. Chen, G. Yu, H. Chen, and Z. Zheng, "Sieve: Attention-based sampling of end-to-end trace data in distributed microservice systems," in *2021 IEEE International Conference on Web Services, ICWS 2021, Chicago, IL, USA, September 5-10, 2021*, C. K. Chang, E. Daminai, J. Fan, P. Ghodous, M. Maximilien, Z. Wang, R. Ward, and J. Zhang, Eds. IEEE, 2021, pp. 436–446. [Online]. Available: https://doi.org/10.1109/ICWS53863.2021.00063

[11] A. U. Gias, Y. Gao, M. Sheldon, J. A. Perusquía, O. O'Brien, and G. Casale, "Samplehst: Efficient on-the-fly selection of distributed traces," in *NOMS 2023, IEEE/IFIP Network Operations and Management Symposium, Miami, FL, USA, May 8-12, 2023*. IEEE, 2023, pp. 1–9. [Online]. Available: https://doi.org/10.1109/NOMS56928.2023.10154383

[12] T. Zhou, C. Zhang, X. Peng, Z. Yan, P. Li, J. Liang, H. Zheng, W. Zheng, and Y. Deng, "Tracestream: Anomalous service localization based on trace stream clustering with online feedback," in *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023, Florence, Italy, October 9-12, 2023*. IEEE, 2023, pp. 601–611. [Online]. Available: https://doi.org/10.1109/ISSRE59848.2023.00033

[13] S. Har-Peled, P. Indyk, and R. Motwani, "Approximate nearest neighbor: Towards removing the curse of dimensionality," *Theory Comput.*, vol. 8, no. 1, pp. 321–350, 2012. [Online]. Available: https://doi.org/10.4086/toc.2012.v008a014

[14] M. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, J. H. Reif, Ed. ACM, 2002, pp. 380–388. [Online]. Available: https://doi.org/10.1145/509907.509965

[15] E. A. Manzoor, S. M. Milajerdi, and L. Akoglu, "Fast memory-efficient anomaly detection in streaming heterogeneous graphs," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, and R. Rastogi, Eds. ACM, 2016, pp. 1035–1044. [Online]. Available: https://doi.org/10.1145/2939672.2939783

[16] F. Cao, M. Ester, W. Qian, and A. Zhou, "Density-based clustering over an evolving data stream with noise," in *Proceedings of the Sixth SIAM International Conference on Data Mining, April 20-22, 2006, Bethesda, MD, USA*, J. Ghosh, D. Lambert, D. B. Skillicorn, and J. Srivastava, Eds. SIAM, 2006, pp. 328–339. [Online]. Available: https://doi.org/10.1137/1.9781611972764.29

[17] SkyWalking. (2024) An apm (application performance monitoring) system. [Online]. Available: https://skywalking.apache.org/

[18] Lightstep. (2024) A distributed tracing library. [Online]. Available: https://www.servicenow.com/products/observability.html

[19] OpenTracing. (2024) Vendor-neutral apis and instrumentation for distributed tracing. [Online]. Available: https://opentracing.io/

[20] A. W. Services. (2024) Aws service level agreements (slas). [Online]. Available: https://aws.amazon.com/legal/service-level-agreements/

[21] C. Zhang, X. Peng, T. Zhou, C. Sha, Z. Yan, Y. Chen, and H. Yang, "Tracecrl: contrastive representation learning for microservice trace analysis," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 1221–1232. [Online]. Available: https://doi.org/10.1145/3540250.3549146

[22] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue, and D. Pei, "Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks," in *31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020*, M. Vieira, H. Madeira, N. Antunes, and Z. Zheng, Eds. IEEE, 2020, pp. 48–58. [Online]. Available: https://doi.org/10.1109/ISSRE5003.2020.00014

[23] D. Lemire and O. Kaser, "Strongly universal string hashing is fast," *Comput. J.*, vol. 57, no. 11, pp. 1624–1638, 2014. [Online]. Available: https://doi.org/10.1093/comjnl/bxt070

[24] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB '03. VLDB Endowment, 2003, p. 81–92.

[25] M. Ester, H. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, E. Simoudis, J. Han, and U. M. Fayyad, Eds. AAAI Press, 1996, pp. 226–231. [Online]. Available: http://www.aaai.org/Library/KDD/1996/kdd96-037.php

[26] O. Boutique. (2024) A cloud-first microservices demo application. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[27] 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, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 553–565. [Online]. Available: https://doi.org/10.1145/3611643.3616249

[28] X. Li, Y. Chen, and Z. Lin, "Towards automated inter-service authorization for microservice applications," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos, SIGCOMM 2019, Beijing, China, August 19-23, 2019*. ACM, 2019, pp. 3–5. [Online]. Available: https://doi.org/10.1145/3342280.3342288

[29] J. F. Loff, D. Porto, J. Garcia, J. Mace, and R. Rodrigues, "Antipode: Enforcing cross-service causal consistency in distributed applications," in *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, Eds. ACM, 2023, pp. 298–313. [Online]. Available: https://doi.org/10.1145/3600006.3613176

[30] Locust. (2024) An open source load testing tool. [Online]. Available: https://locust.io/

[31] Kubernetes. (2024) An open-source system for automating deployment, scaling, and management of containerized applications. [Online]. Available: https://kubernetes.io/

[32] Z. Chen, Y. Kang, L. Li, X. Zhang, H. Zhang, H. Xu, Y. Zhou, L. Yang, J. Sun, Z. Xu, Y. Dang, F. Gao, P. Zhao, B. Qiao, Q. Lin, D. Zhang, and M. R. Lyu, "Towards intelligent incident management: why we need it and how we make it," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 1487–1497. [Online]. Available: https://doi.org/10.1145/3368089.3417055

[33] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings*, H. Balakrishnan and P. Druschel, Eds. USENIX, 2007. [Online]. Available: http://www.usenix.org/events/nsdi07/tech/fonseca.html

[34] 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, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, E. L. Miller and S. Hand, Eds. ACM, 2015, pp. 378–393. [Online]. Available: https://doi.org/10.1145/2815400.2815415

[35] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, and Y. J. Song, "Canopy: An end-to-end performance tracing and analysis system," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 34–50. [Online]. Available: https://doi.org/10.1145/3132747.3132749

[36] J. Shen, H. Zhang, Y. Xiang, X. Shi, X. Li, Y. Shen, Z. Zhang, Y. Wu, X. Yin, J. Wang, M. Xu, Y. Li, J. Yin, J. Song, Z. Li, and R. Nie, "Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code," in *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, H. Schulzrinne, V. Misra, E. Kohler, and D. A. Maltz, Eds. ACM, 2023, pp. 420–437. [Online]. Available: https://doi.org/10.1145/3603269.3604823

[37] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds. ACM, 2019, pp. 683–694. [Online]. Available: https://doi.org/10.1145/3338906.3338961

[38] H. Wang, Z. Wu, H. Jiang, Y. Huang, J. Wang, S. Köprü, and T. Xie, "Groot: An event-graph-based approach for root cause analysis in industrial settings," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 419–429. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678708

[39] Z. Chen, J. Liu, Y. Su, H. Zhang, X. Wen, X. Ling, Y. Yang, and M. R. Lyu, "Graph-based incident aggregation for large-scale online service systems," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 430–442. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678746

[40] 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 *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1750–1762. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00150

[41] Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi, "CRISP: critical path analysis of large-scale microservice architectures," in *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, J. Schindler and N. Zilberman, Eds. USENIX Association, 2022, pp. 655–672. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou

[42] 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, ICPP 2022, Bordeaux, France, 29 August 2022 - 1 September 2022*. ACM, 2022, pp. 47:1–47:11. [Online]. Available: https://doi.org/10.1145/3545008.3545026

[43] J. Liu, Z. Jiang, J. Gu, J. Huang, Z. Chen, C. Feng, Z. Yang, Y. Yang, and M. R. Lyu, "Prism: Revealing hidden functional clusters from massive instances in cloud systems," in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 268–280. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00077

[44] D. Huye, Y. Shkuro, and R. R. Sambasivan, "Lifting the veil on meta's microservice architecture: Analyses of topology and request workflows," in *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, J. Lawall and D. Williams, Eds. USENIX Association, 2023, pp. 419–432. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/huye

[45] X. Peng, C. Zhang, Z. Zhao, A. Isami, X. Guo, and Y. Cui, "Trace analysis based microservice architecture measurement," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 1589–1599. [Online]. Available: https://doi.org/10.1145/3540250.3558951

[46] L. Zhang, Z. Xie, V. Anand, Y. Vigfusson, and J. Mace, "The benefit of hindsight: Tracing edge-cases in distributed systems," in *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, M. Balakrishnan and M. Ghobadi, Eds. USENIX Association, 2023, pp. 321–339. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/zhang-lei