Efficient and Robust Trace Anomaly Detection for Large-Scale Microservice Systems

Shenglin Zhang[†§], Zhongjie Pan[†], Heng Liu[†], Pengxiang Jin[†], Yongqian Sun[*†], Qianyu Ouyang[¶]
Jiaju Wang[†], Xueying Jia[†], Yuzhi Zhang[†], Hui Yang[‡], Yongqiang Zou[‡], Dan Pei[¶]
[†]Nankai University, {zhongjie, lheng, wangjiaju, jinpengxiang}@mail.nankai.edu.cn
{zhangsl, sunyongqian, zyz}@nankai.edu.cn, jiaxueying20@gmail.com
[‡]Accumulus Technologies (Tianjin) Co., Ltd, {hui.yang, yongqiang.zou}@yunzhanghu.com
[§]Haihe Laboratory of Information Technology Application Innovation
[¶]Tsinghua University, oyqy19@mails.tsinghua.edu.cn, peidan@tsinghua.edu.cn

***Abstract*—Microservice invocation anomalies can have a detrimental impact on user experience and service revenue. While existing trace anomaly detection approaches typically focus on anomalies in response time and invocation structure, they often overlook the importance of using fine-grained features to detect anomalies. Additionally, trace data obtained from real-world scenarios is typically accompanied by noise, which can hinder the effectiveness of anomaly detection approaches. Furthermore, large-scale trace data can significantly impact model training efficiency. To address these challenges, we propose *TraceSieve*, an unsupervised trace anomaly detection method that accurately detects trace anomalies. Our approach leverages an auto-encoder architecture within an adversarial training framework to filter out noise data. Additionally, we integrate VGAE-EWC, which combines Variational Graph Auto-Encoder (VGAE) with Elastic Weight Consolidation (EWC), to overcome the challenges of enormous time consumption during the training phase. Finally, we localize the root cause of trace anomalies. Our proposed method is evaluated using two different datasets, and our results demonstrate that *TraceSieve* achieves an F_1-score of 0.970 and 0.925, respectively, outperforming state-of-the-art trace anomaly detection approaches.**

***Index Terms*—microservice, trace, failure detection**

I. Introduction

As a company's business operations scale up, the corresponding processes become increasingly complex, necessitating the adoption of advanced approaches such as microservice architecture [1]. In contrast to traditional monolithic service architecture, microservice architecture employs a modular approach where service instances are partitioned into small, self-contained units that can be independently deployed and scaled [2]. The scalability, reusability, and independent deployment benefits of microservice architecture have made it the prevailing trend in software development. Nonetheless, the reliability issues that happen in microservice systems can greatly influence customers' satisfaction and business revenue. It is estimated that the average hourly cost of system's downtime is between $301,000 and $400,000 [3].

To ensure the reliability of microservice systems, it is important to timely detect anomalies and localize the root causes of anomalies. To this end, operators carefully record the *trace* data among microservices. *Trace* tracks the users' requests related invocations and executions among microservices in detail. For example, Figure 1 shows a trace of a microservice system. The trace recorded the *processing* time of each service as well as the *execution* time of requesting and responding. The left part of the figure shows the over structure of the request, while the right part shows the detailed information of execution 2 and 8. By analyzing such information, operators can identify the happening of an anomaly and localize the root cause of an anomaly.

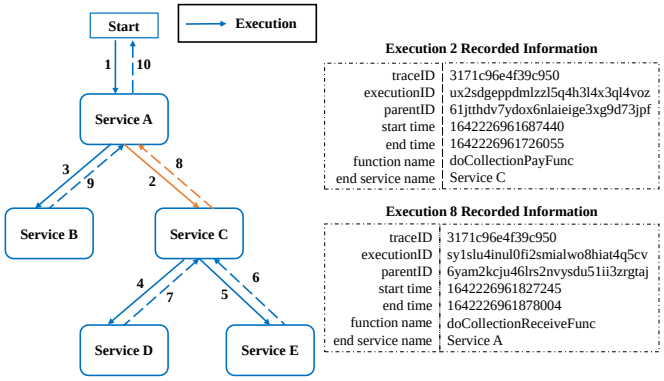


Fig. 1. An example of trace in a microservice system (Service X means a microservice).

Trace-based anomaly detection and root cause localization have been a topic of great interest over the years. A prevalent initial step in this process involves converting the traces into numerical vectors, which enables the application of machine learning techniques. Some approaches [4], [5] define features such as the frequencies of different service operations. Others [6], [7] enumerate the call path to consider structural feature, and then take the metrics on path into vectors.

However, most existing approaches ignore the detailed information carried by traces. We contend that leveraging this detailed information is essential for achieving more accurate anomaly detection and fine-grained root cause localization. The information carried by traces can be categorized into three aspects: structural, service processing, and transmission execution, which correspond to three types of system

* Yongqian Sun is the corresponding author.

anomalies (§ II-B). Structural information typically represents logical errors. Service processing information signifies internal service errors. Transmission execution information indicates networking issues. To perform a more comprehensive analysis, it is imperative to consider these three types of information. However, effectively utilizing this information faces two significant challenges.

1) **Mixed normal and anomalous data.** Traces inherently record both normal and abnormal invocations. However, manually labeling these anomalies can be time-consuming and error-prone. Consequently, identifying normal and abnormal traces in practice becomes challenging. Incorporating both normal and abnormal data in training can significantly compromise the performance of machine learning models due to the presence of noise in the data. For instance, in our collaborating company, abnormal traces constitute approximately 1% of the total traces. To conduct a preliminary study, we manually cleaned the traces from one week and assessed the impact of noise on the models' performance. As listed in Table I, this amount of noise can considerably degrade the performance of the models.
2) **Large data volume.** Traces are recorded in large volumes [4], [8]. For example, our cooperated company pipelines generate about *three million* traces in a single day. We take the data of one week to conduct pilot experiments. It takes more than 192 hours (8 days) to train existing trace analysis approaches, which is unacceptable for the company to detect abnormal operations in real-time.

TABLE I
THE IMPACT OF MIXED NORMAL AND ANOMALOUS DATA

Approach	F_1 trained on *cleaned* data	F_1 trained on *raw* data	Impact
MultimodalTrace [9]	0.809	0.337	0.472↓
AEVB [10]	0.831	0.328	0.503↓
TraceAnomaly [7]	0.828	0.385	0.443↓
TraceCRL [11]	0.860	0.427	0.433↓
Sage [12]	0.847	0.326	0.521↓

To address the above challenges, we propose *TraceSieve*, an unsupervised framework for trace-based anomaly detection and root cause localization. First, we filter the abnormal data from raw data by an auto-encoder architecture within an adversarial training framework. We further design VGAE-EWC, which combines variational graph auto-encoder (VGAE) with an incremental training strategy using the elastic weight consolidation (EWC) method, to reduce the time cost of model training. Our contributions can be summarized as follows:

1) We introduce a noise filtering method based on Generative Adversarial Networks (GANs) to eliminate interference from noise during the training data preprocessing stage, thus addressing the first challenge. We remove noise data through a custom threshold calculated from each trace's noise level (§ III-B2).
2) To tackle the challenge posed by large-scale training data, we propose VGAE-EWC, which combines VGAE with an incremental training strategy using the elastic weight consolidation (EWC) method. Our approach involves dividing the training data into multiple segments and training the model incrementally while accepting feedback from online detection through hyperparameters. This strategy enables us to improve training efficiency and the efficacy of the model simultaneously (§ III-C).
3) We conduct extensive experiments using two datasets, one of which is collected from a large-scale microservice system deployed in an e-commerce company. The results show that our framework detects trace anomalies in microservices with an average F_1-score of 0.970 and 0.925, outperforming baseline methods by 0.235 and 0.119 on average, respectively (§ IV).

II. BACKGROUND

A. Microservices and Traces

Microservice systems are large-scale distributed online systems that consist of thousands of isolated microservice instances constantly calling each other [2]. The process of calling microservice instances in a specific order is referred to as a trace [9], [13]. Due to the invocation patterns commonly found in microservice systems, a trace is constructed in a chain of microservice instance invocations. Executions in a trace record various features such as start time, end time, function name, and end service name when invocations occur. These features can be used to rearrange and analyze traces.

A trace in a microservice architecture can form a directed graph that consists of a sequence of executions, each of which corresponds to an invocation between two microservices [2]. Traces and executions are both assigned a unique identifier to differentiate them from others. Each execution contains information about the microservice instance being invoked, such as the function name and the end service name to which the invocation is directed. Moreover, each execution in a trace has a parent execution that generates the invocation to the current execution, except for the first execution in the trace. Distributed traces play an important role in microservice systems. A variety of open-source distributed trace recording infrastructures, including Jaeger [14], Zipkin [15], SkyWalking [16], OpenTracing [17], and ES-APM have been developed in recent years to support the development of distributed traces. In the studied company, ES-APM was deployed by operators to collect trace data.

Previous methods for detecting anomalies in traces have primarily focused on response time and invocation structure features [7], [18]. While these methods have demonstrated relatively impressive detection results, they do not make use of all available features, which can limit the effectiveness of anomaly detection. For instance, each execution in a trace contains time features, as depicted in Figure 1. The start time denotes the moment when the service sends a request, and the end time denotes the moment when the service receives a response. Through these time features, we can calculate the processing time at the services and the waiting time at the executions, as shown in Equation 1:

$$PT(E) = ST(6) - ET(5)$$
$$WT(5) = ET(5) - ST(5) \quad (1)$$

Here, $ST(5)$ and $ST(6)$ are the start time of Execution 5 and Execution 6, and $ET(5)$ is the end time of Execution 5. $PT(E)$ represents the invocation gaps, which we refer to as *processing time* at Service E. $WT(5)$ represents the time consumed during request transmission and waiting in the queue, which we refer to as *waiting time* for Execution 5. These features enable us to gain a closer insight into the invocations between executions in a trace. By leveraging these features, we can detect many different anomalies in a trace with high accuracy.

B. Trace Anomaly

We identify three types of common anomalies: processing time (PT), waiting time (WT), and structural anomaly. Waiting time and processing time anomalies can be determined by deviations in their values from the distribution of normal values. Structural anomalies can be determined by unusual calling structures.

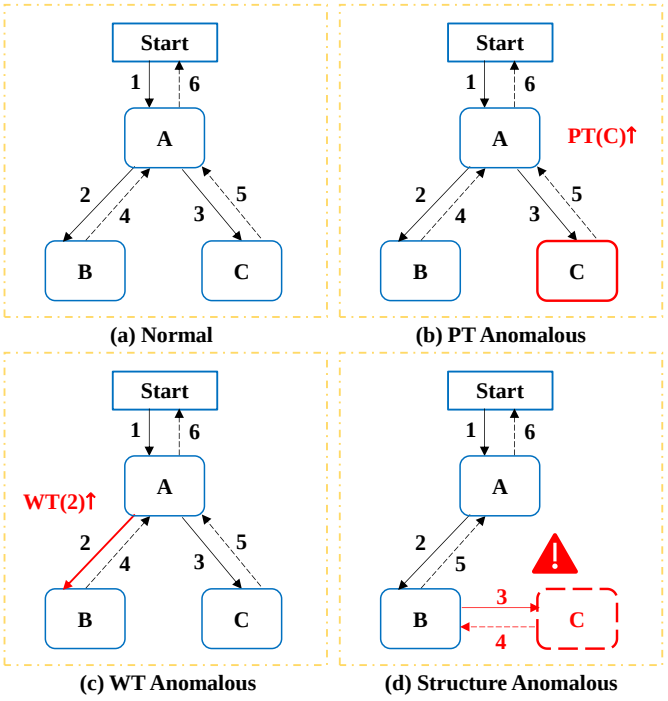


Fig. 2. Examples of normal trace and three types of anomalous traces.

- **Processing time anomaly.** This type of anomaly is often associated with internal service errors, which may be due to issues like resource contention, inefficient algorithms, software bugs, *etc*. Figure 2(b) shows an example of processing time anomaly. The processing time of Service C in the anomalous trace increases rapidly in a short time, which is beyond the normal range.
- **Waiting time anomaly.** This type of anomaly is often associated with networking issues, which can be caused by problems like network congestion, packet loss, high latency, *etc*. Figure 2(c) shows an example of waiting time anomaly. The waiting time of Execution 2 in the anomalous trace increases rapidly in a short time, which is beyond the normal range.
- **Structural anomaly.** This type of anomaly is often associated with logical errors within the system, which can lead to unexpected behavior. Figure 2(d) shows an example of structural anomaly. Instead of the normal executions between Service A and Service C, incorrect executions are made between Service B and Service C.

III. TraceSieve Approach

A. Design Overview

This paper proposes an unsupervised trace anomaly detection method, named *TraceSieve*, which is based on the variational graph auto-encoder (VGAE [19]) to effectively and efficiently detect trace anomalies. Figure 3 illustrates the overall framework of *TraceSieve*, which comprises three stages: data preprocessing, offline training, and online detection.

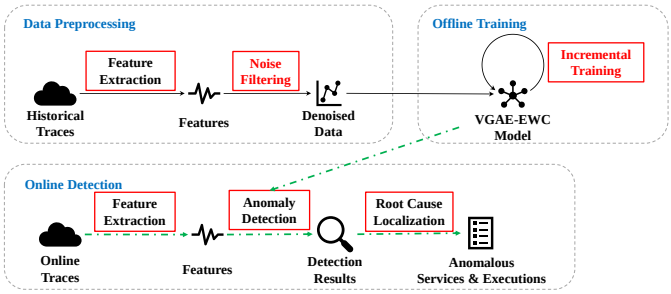


Fig. 3. The overview framework of *TraceSieve*.

In the data preprocessing stage, historical trace data is subjected to feature extraction and noise filtering processes. Specifically, we extract the numerical features of a trace and represent them in a trace feature matrix (TFM), while the structural features are transformed into an adjacency matrix. To mitigate the influence of three types of noise on anomaly detection, we propose an auto-encoder architecture within an adversarial training framework inspired by GANs [20]. Then, we adopt the VGAE-EWC model for offline training.

During online detection, each new trace is first converted into the TFM and adjacency matrix representations. The trained model then computes the negative log-likelihood (NLL) as the anomaly score for each new trace. Finally, instead of manually determining a threshold for anomalous traces, we utilize the p-value to automatically decide the threshold based on the anomaly score. The p-value is set at a commonly used level of 0.001 for statistical hypothesis testing missions.

B. Data Preprocessing

1) Feature Extracting: We extract three types of features, namely processing time, invocation structure, and waiting time, to represent a trace. Following the approach proposed

in [7], we reconstruct the invocation path of a trace to extract the execution time and start time of each span. Compared to the STV approach, we introduce a new data structure called trace feature matrix (TFM) to store the features extracted from traces. The execution time of each span is stored in the first dimension of the TFM in the order of the invocation path. Next, we calculate the waiting time of each service using the method described in Eq.1 and store them in the second dimension of the TFM in the same order as the execution time. Regarding the invocation structure, we use a sparse matrix, referred to as the adjacency matrix, to store each end service in a trace in the order of the invocation path. When a trace is transformed into an invocation graph, each service serves as a node in the graph, and the invocations between two services correspond to the non-zero elements in the adjacency matrix.

Figure 4 illustrates the TFM and adjacency matrix of the sample trace depicted in Figure 1.

Trace Feature Matrix (TFM)

Waiting Time	Processing Time
WT(1)	0
WT(2)	PT_1(A)
WT(3)	PT_2(A)
WT(4)	PT_1(C)
WT(5)	PT_2(C)
WT(6)	PT(E)
WT(7)	PT(D)
WT(8)	PT_3(C)
WT(9)	PT(B)
WT(10)	PT_3(A)

Adjacency Matrix

0	1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	0	0
0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0	1	1
0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	0	0

Fig. 4. The TFM and adjacency matrix of the example trace.

2) Noise Filtering: A modern microservice system at a large scale always involves a considerable amount of invocation data (traces) and operation instances. As a result, it is challenging to ensure the quality of all the collected trace data, as there may be issues with the recording process due to machine malfunction or manual errors. Our analysis reveals that the proportion of noise in the trace data is typically minimal (around 1%), which can be classified into three types that are similar to trace anomalies: waiting time noise, processing time noise, and invocation structure noise. Among these types of noise, invocation structure noise constitutes the largest proportion (around 70%). The percentage of waiting time noise and processing time noise is 25% and 5%, respectively. Nevertheless, noise remains a significant obstacle to the model's ability to learn standard patterns during the offline training process.

To address the issue of noisy trace data, we propose an auto-encoder architecture within an adversarial training framework, which is inspired by the Generative Adversarial Networks (GANs) [20]. The proposed approach is an unsupervised neural network comprising a generator (G) and two discriminators (D_1 and D_2). The generator aims to generate trace data to maximize the probability of the discriminators making mistakes. The discriminators, on the other hand, reconstruct features from traces and classify them as real or generated.

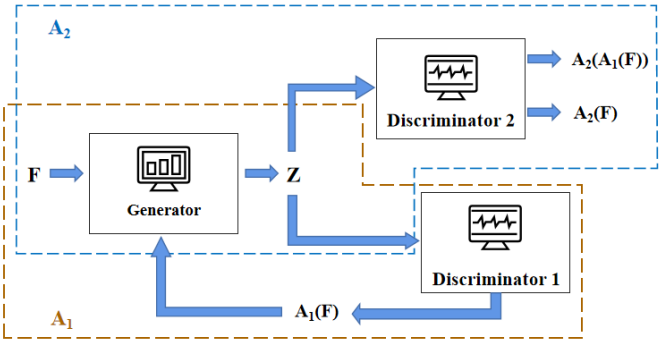


Fig. 5. The structure of noise filtering module.

The GAN architecture includes two auto-encoders (A_1 and A_2) that use the same generator G. We extract features from traces through data preprocessing, which we denote as F, and f_i represents features from one trace.

$$\begin{aligned} F &= \{f_1, f_2, \ldots, f_n\} \\ A_1(F) &= D_1(G(F)) \\ F' &= F \cup A_1(F) \\ A_2\left(F'\right) &= D_2\left(G\left(F'\right)\right) \end{aligned} \tag{2}$$

Figure 5 shows the execution process of the proposed GAN-based approach. Initially, the two auto-encoders (A_1 and A_2) reconstruct the normal input features F. During the training process, A_1 interferes with A_2's judgment, and A_2 determines whether the data is real (F) or generated ($A_1(F)$). The proposed approach effectively reduces the impact of noise in trace data and improves the quality of the data used for offline training.

More details extend in the following:

- **Stage 1: Reconstruct Traces.** Input features F are encoded by generator G into the latent variable Z and then reconstructed by discriminators D_1 and D_2 in turn. The specific implementation principles are given by Eq. 2:

$$\begin{aligned} \mathscr{L}_{A_1} &= ||F - A_1(F)||_2 \\ \mathscr{L}_{A_2} &= ||A_2(A_1(F)) - A_2(F)||_2 \end{aligned} \tag{3}$$

  Here, $\mathscr{L}_{A_1}$ and $\mathscr{L}_{A_2}$ represent the loss function of auto-encoders A_1 and A_2, respectively.
- **Stage 2: Classification.** After processing input features, the objective of A_2 is to discriminate between raw trace features and reconstructed trace features from A_1. Features from A_1 are encoded repeatedly into Z. According to the adversarial training framework, the objective of A_1 is to minimize the distinction between F and the output of A_2, while the objective of A_2 is to maximize it. The objective function can be written as:

$$\min_{A_1} \max_{A_2} ||F - A_2(A_1(F))||_2 \tag{4}$$

This accounts for the following loss functions:

$$\begin{aligned} \mathscr{L}_{A_1} &= \frac{1}{n}||F - A_1(F)||_2 + (1 - \frac{1}{n})||F - A_2(A_1(F))||_2 \\ \mathscr{L}_{A_2} &= \frac{1}{n}||F - A_2(F)||_2 + (1 - \frac{1}{n})||F - A_2(A_1(F))||_2 \end{aligned} \tag{5}$$

After features from traces have undergone two rounds of calculation, the anomaly score can be defined as follows:

$$\mathscr{S}(F) = \alpha||F - A_1(F)||2 + (1-\alpha)||F - A_2(A_1(F))||_2 \tag{6}$$

Here, α is a parameter that determines the proportion of the two loss functions from A_1 and A_2. The size of α determines the sensitivity of the noise filtering process, as discussed in § IV-E.

C. VGAE-EWC

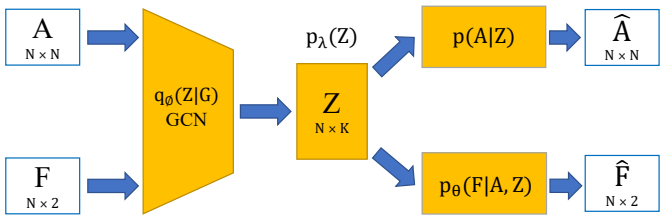


Fig. 6. A general architecture of VGAE.

Our VGAE design is presented in Figure 6, where **F** and **A** represent the trace feature matrix and adjacency matrix, respectively, of the same trace. N denotes the number of invocations in the trace. Our VGAE leverages a Graph Convolutional Network (GCN) as an encoder to encode **F** and **A** into hidden features. Subsequently, the hidden features are passed through the encoder to obtain the variational posterior distribution $q_\phi(Z|\mathbf{A}, \mathbf{F})$. Further, a fully-connected network is employed to decode graph-level Z back into node-level features, which are used to generate the reconstructed $\widehat{A}$ and $\widehat{F}$. Here, $q_\phi(z|\mathbf{A}, \mathbf{F})$ and $p_\theta(N, \mathbf{A}, \mathbf{F}|z)$ denote the encoder and decoder of the main Variational Autoencoder (VAE), respectively. Specifically, A_{ij} and F_i follow Bernoulli and Gaussian distributions, respectively. Additionally, $p_\theta(z)$ represents a learned RealNVP prior [21]. The loss function of our proposed method *TraceSieve* is formulated as follows:

$$Loss = \gamma \cdot KLD[q(Z|F) \;||\; p(Z)] - E_{q(Z|A,\ F)}[\log p(A|Z)] \tag{7}$$

where γ is a hyperparameter that balances the proportion of KL-divergence.

However, the application of the general VGAE for training the detection model reveals a significant increase in both training time and memory cost. Although the issue of high memory cost can be resolved using simple techniques such as parallel computing and distributed input, the training time cannot be mitigated by these methods. To address this problem, we propose a novel approach, VGAE-EWC, which combines VGAE with an incremental training strategy called Elastic Weight Consolidation (EWC) to significantly reduce training time. EWC is a method that enables continual model learning by restricting the plasticity of synapses that are essential to previous tasks [22]. This method constrains important parameters to stay close to their old values. Given a dataset $\mathscr{D}$ and the biases θ of the linear projection, we can use Bayes' rule to calculate the conditional probability $p(\mathscr{D}|\theta)$ as follows:

$$\log p(\theta|\mathscr{D}) = \log p(\mathscr{D}|\theta) + \log p(\theta) - \log p(\mathscr{D}) \tag{8}$$

It can be observed that $\log p(\mathscr{D}|\theta)$ is the negative of the loss function for $-\mathscr{L}(\theta)$. Assuming that $\mathscr{D}_A$ and $\mathscr{D}_B$ constitute the dataset $\mathscr{D}$, the Eq.8 can be rearranged as follows for task $A(\mathscr{D}_A)$ that has been used for training and task B ($\mathscr{D}_B$) to be trained:

$$\log p(\theta|\mathscr{D}) = \log p(\mathscr{D}_B|\theta) + \log p(\theta|\mathscr{D}_A) - \log p(\mathscr{D}_B) \tag{9}$$

As a result, all the parameters defined in the entire dataset $\mathscr{D}$ will depend only on the loss function of task B, which is $\log p(\mathscr{D}_B|\theta)$. The trained information of task A is stored in the posterior distribution $p(\theta|\mathscr{D}_A)$. During training of the two tasks, the focus is on minimizing the loss function of task B as follows:

$$\mathscr{L}(\theta) = \mathscr{L}_B(\theta) + \sum_i \frac{\lambda}{2} F_i(\theta_i - \theta^*_{A,i})^2 \tag{10}$$

Here, λ represents the weight of the last task compared to the next, θ^*_A is the optimal parameter of task A, and F is the Fisher information matrix [23] used to measure the importance of the parameters θ^*_A.

To avoid the exponential increase in training time due to large-scale data, we iteratively train the VGAE-EWC model using multiple datasets obtained by dividing the trace data of a specific number of days by day.

D. Anomaly Score

During the testing process, it is necessary to assign an anomaly score to each trace to discern whether it is anomalous. The negative log-likelihood (NLL) is a commonly utilized anomaly score in other trace anomaly detection methodologies, and it reflects the extent to which a graph G deviates from the normal pattern that is learned by the model. Monte Carlo integration is a popular method for calculating the NLL [24]. Specifically, the NLL of a graph G can be expressed as:

$$\begin{aligned} NLL_G &= -\log p_{model}(G) \\ &= -\log \mathbb{E}q\phi(z|G, N) \left[\frac{p_\theta(G, N, z)}{q_\phi(z|G, N)}\right] \\ &\approx -\log \left[\frac{1}{L}\sum_{l=1}^{L} \frac{p_\theta(N, \mathbf{A}, \mathbf{X}, z^{(l)})}{q_\phi(z^{(l)}|G, N)}\right] \end{aligned} \tag{11}$$

where $z^{(l)}$ is a sample from $q_\theta(z|G, N)$, and L is the number of samples. The NLL predominantly calculates the dissimilarity between the reconstructed graphs and the original traces, which can be derived from Eq. 7.

Directly using negative log-likelihoods (NLLs) to identify anomalous traces can result in a problem. Nalisnick et

al. [25] observed that deep generative models, including Variational Autoencoders (VAEs), exhibit low NLLs for out-of-distribution data when compared to regular training data. They did not propose practical solutions to address this issue. We propose a new strategy for computing the anomaly score. Let p^* represent the distribution of testing data. The expectation of NLLs ($\mathbb{E}_{p^*}[NLL_x]$) can be expressed as the sum of the Kullback-Leibler (KL) divergence of p^* on the model p_{model} and the data entropy ($\mathbb{H}[p^*]$). The KL divergence measures the difference between testing data and the model, which is always ≥ 0. However, the data entropy is not controlled by the model, unlike the KL divergence. If the data entropy of the testing data ($\mathbb{H}[p^*]$) is no greater than the training data ($\mathbb{H}[p_{training}]$), the NLLs on the testing data could be lower than the training data. This problem is referred to as the entropy gap. To address this issue, we propose a new strategy. After analyzing thousands of traces, we observed that the standard deviations (STDs) of x_i for anomalous traces are generally more significant than the expected data. Therefore, we can decrease the entropy gap by clipping the STDs of $p(x_i|\mathbf{A}, z)$. Specifically, we clip the STDs by computing $\widetilde{\sigma_i} = \min \sigma_i, \sigma_{std99.9}$, where σ_i and $\widetilde{\sigma_i}$ represent the STDs of x_i before and after clipping, respectively, and $\sigma_{std99.9}$ is the boundary value of the 99.9% of the STDs. As $\widetilde{\sigma_i} < \sigma_i$ usually holds for anomalous traces, their NLLs can increase, thereby deviating from the normal distribution.

E. Online Detection

TraceSieve trains a fine-tuned VGAE-EWC model for online detection of anomalies in new trace data. In this stage, operators extract features and construct the TFM and the adjacent matrix for each new trace, which are then fed into the pre-trained model to calculate the anomaly score using Eq.11 and the distribution learned from the training set. As the traces in the studied company are dynamic, it is not feasible to manually set a fixed threshold to classify anomalous scores. Following previous works in trace anomaly detection [7], [9], [10], we propose to use the p-value approach to distinguish anomalous scores. We set the p-value threshold at 0.001, following the standard criterion commonly used in statistical hypothesis testing.

F. Localizing Root Cause

When an anomalous trace is detected in the online detection period, the mission of root cause localization is to identify the root microservice that caused the system failure, such as a service request timeout, waiting queue overrun, or incorrect interface dependency. The abnormal features of the microservice and its corresponding invocation path are used to clearly interpret the root cause.

Our root cause localization algorithm leverages the physical significance of the trace feature matrix, where each dimension represents an execution and includes the waiting and processing times. When an anomalous trace is detected, *TraceSieve* searches the training set to identify which trace feature matrices are homogeneous (i.e., have the same valid dimensions) with the trace feature matrix of the anomalous trace. If no homogeneous trace feature matrix is found, *TraceSieve* determines that the anomalous trace has an invocation structure anomaly. To further investigate the anomalous trace, *TraceSieve* identifies the trace feature matrix with the longest common invocation path with the homogeneous trace feature matrix of the anomalous trace.

For each valid dimension in the anomalous trace, *TraceSieve* calculates and stores the mean μ and standard deviation σ of the waiting and processing times from the training set. Because the waiting and processing times have different distributions, *TraceSieve* applies the z-score normalization strategy to measure the abnormality of the values in the anomalous trace's trace feature matrix. Specifically, for a value x in the trace feature matrix, *TraceSieve* calculates the anomaly severity as follows:

$$\begin{aligned} Anomaly\ Score(x_i) &= \frac{x_i - \mu_x}{\sigma_x} \\ \mu_x &= \frac{\sum_{i \in N} x_i}{N} \\ \sigma_x &= \frac{\sum_{i \in N} (x_i - \mu_x)^2}{N} \end{aligned} \tag{12}$$

where x is either the waiting time or the processing time, and has a set of μ_x and σ_x respectively. Among these abnormalities of values in the trace feature matrix, *TraceSieve* selects the top k abnormal features by their severity values. Finally, *TraceSieve* maps the abnormal features to their corresponding microservices in the trace feature matrix to identify the top k abnormal microservices.

IV. Evaluation

In this study, we aim to answer the following research questions (RQs) through extensive experimentation:

RQ1. How does the performance of *TraceSieve* compare to baseline methods in terms of trace anomaly detection, including both effectiveness and training time?

RQ2. How does the precision of *TraceSieve* compare to baseline methods in root cause localization?

RQ3. What are the significant technical contributions of *TraceSieve*, including feature extraction, noise filtering, and incremental training, and how do these modules contribute to its overall performance?

RQ4. How do the hyperparameters of *TraceSieve* impact its performance, and what are the optimal settings for these parameters?

TABLE II
THE DETAILS OF DATASETS

Dataset	# Microservice Instances	# Failures	# Trace Records
$\mathscr{D}1$	10	1191820	23520998
$\mathscr{D}2$	9	8442	36705835

A. Experiment Setup

1) Datasets: To evaluate the performance of *TraceSieve*, we conducted extensive experiments on two datasets, one

TABLE III
THE EFFECTS OF *TraceSieve* IN COMPARISON WITH DIFFERENT APPROACHES ON TWO DATASETS

Dataset	Approach	WT			IS			PT			Total			Time (h)
		P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	
$\mathscr{D}1$	CFG [26]	0.851	0.873	0.862	-	-	-	0.832	0.865	0.848	0.652	0.749	0.697	90
	CPD [27]	-	-	-	0.923	0.947	0.935	-	-	-	0.478	0.682	0.562	96
	MultimodalTrace [9]	0.812	0.857	0.834	0.631	0.764	0.691	0.795	0.823	0.809	0.747	0.807	0.776	126.7
	AEVB [10]	0.827	0.784	0.805	-	-	-	0.803	0.772	0.787	0.634	0.687	0.659	683.2
	TraceAnomaly [7]	0.866	0.820	0.842	0.886	0.791	0.836	0.873	0.848	0.860	0.867	0.819	0.842	315
	TraceCRL [11]	0.883	0.796	0.864	0.867	0.849	0.852	0.906	0.887	0.891	0.895	0.824	0.874	159.6
	TraceSieve	**0.984**	**0.972**	**0.978**	**0.965**	**0.973**	**0.969**	**0.971**	**0.962**	**0.966**	**0.973**	**0.968**	**0.970**	**4.3**
$\mathscr{D}2$	CFG [26]	0.802	0.839	0.820	-	-	-	0.817	0.831	0.826	0.610	0.722	0.661	46
	CPD [27]	-	-	-	**0.905**	0.933	0.919	-	-	-	0.443	0.634	0.522	48
	MultimodalTrace [9]	0.537	0.675	0.598	0.651	0.760	0.701	0.553	0.664	0.603	0.580	0.700	0.634	62.8
	AEVB [10]	0.823	0.804	0.813	-	-	-	0.796	0.751	0.773	0.610	0.684	0.645	314.2
	TraceAnomaly [7]	0.849	0.716	0.777	0.704	0.658	0.680	0.863	0.797	0.829	0.805	0.722	0.761	139.1
	TraceCRL [11]	0.861	0.795	0.837	0.752	0.718	0.739	0.885	0.827	0.860	0.829	0.769	0.808	165.2
	TraceSieve	**0.889**	**0.938**	**0.913**	0.892	**0.981**	**0.934**	**0.943**	**0.866**	**0.903**	**0.915**	**0.936**	**0.925**	**7.6**

open-source and one real-world. The details of the datasets are provided in Table II, which includes the number of microservice instances, the number of failures that occurred in the microservice instances, and the total number of trace records in each dataset.

- **Dataset 1** ($\mathscr{D}1$) is the Generic AIOps Atlas (GAIA) dataset provided by CloudWise[1]. GAIA comprises multi-modal data, including metrics, logs, and traces, recorded from the MicroSS benchmark system[2], which simulates user logins using QR codes in a microservice environment. The dataset includes over 6,500 metrics, 7,000,000 log items, and detailed trace data continuously collected over two weeks. The dataset also simulates system failures by controlling user behavior and mimicking erroneous manipulations to the system. We use a testing set of 3,232 normal traces, 1,564 waiting time anomaly traces, 1,238 invocation structure anomaly traces, and 976 processing time anomaly traces.
- **Dataset 2** ($\mathscr{D}2$) is collected from a large-scale microservice system operated by an e-commerce company. We use two weeks of data as the training set. Due to the difficulty of obtaining labeled data, we inject failures into the system using chaos engineering, such as microservice publish dependency order exceptions, Elastic Load Balancing (ELB) network issues, and downstream service throttling. Professional operators labeled the traces and classified anomaly types. We use a testing set of 3,962 normal traces, 1,132 waiting time anomaly traces, 1,098 invocation structure anomaly traces, and 925 processing time anomaly traces. Unfortunately, we cannot make this dataset publicly available due to a non-disclosure agreement.

2) Environment and Parameters: The implementation of *TraceSieve* is in Python 3.7.12, with PyTorch 1.10.0 serving as the primary deep learning framework. The experiments are conducted on a server with two 16C32T Intel(R) Xeon(R) Gold 5218 CPU @ 2.30 GHz, one NVIDIA(R) Tesla(R) V100S, and 188 GB of RAM. A batch size of 128, learning rate of 0.001, and 50 epochs are used for all deep learning models.

3) Baselines: To evaluate the performance of *TraceSieve* on anomaly detection, we employ six recently proposed trace anomaly detection methods as baseline methods, including CFG [26], CPD [27], AVEB [10], MultimodalTrace [9], TraceAnomaly [7], and TraceCRL [11], which have been introduced in related work. Since CFG and CPD do not employ deep learning methods, we provide the important parameter settings for the remaining three methods. For AEVB, we set the window size to 32. For MultimodalTrace, we use two LSTMs to process the input data and set the weight of the two loss functions to 0.5 each. For TraceAnomaly, we set the dimension of the latent variable z to 10.

To evaluate the precision of *TraceSieve* in root cause localization, we use three recently proposed root cause localization methods, including TraceAnomaly [7], MicroRank [28], and TraceRCA [29]. We mainly provide the parameter settings for the following two methods: MicroRank and TraceRCA. For MicroRank, we set the damping factor d to 0.85 and the preference vector weight φ to the default value of 0.5. And for TraceRCA, we use the default values of 1, 0.1, and 0.1, respectively, for these three parameters δ_{ad}, δ_{fs}, and δ_{spt}.

4) Performance Metrics: In the context of anomaly detection, we compute an anomaly score for each trace in the test set, and subsequently classify the traces as anomalous or not based on a predefined threshold for the anomaly score. The problem of trace anomaly detection can be formulated as a binary classification problem for each trace, with the F_1-score being a widely used performance metric. The F_1-score is computed using true positive samples (TP), false positive samples (FP), and false negatives samples (FN) obtained from the failure detection model.

- **Precision**: the proportion of predicted positive samples that are true positive samples, which can be calculated as: $\text{Precision} = \frac{\text{TP}}{\text{TP+FP}}$.
- **Recall**: the proportion of true positive samples that are correctly predicted as positive, which can be calculated as:

[1]https://github.com/CloudWise-OpenSource

[2]https://github.com/CloudWise-OpenSource/GAIA-DataSet/tree/main/MicroSS

$\text{Recall} = \frac{\text{TP}}{\text{TP+FN}}$.

- F_1**-score**: the harmonic mean of precision and recall that provides a balance between the two metrics, which can be calculated as: $F_1 - score = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$.

To detect and evaluate anomalies for different types of traces, we conduct experiments separately for each type of trace anomaly. We also compute the best F_1-score for failure-incorporative data, which includes all types of failures in the traces. The best F_1-score is defined as the optimal threshold value that maximizes the F_1-score when the threshold value is unknown.

In the context of root cause localization, we use precision at top-k (P@k) as a commonly used metric for ranking, which indicates the probability that the top-k microservices identified by the approach contain the root cause.

B. TraceSieve vs. Baseline Algorithms (RQ1)

Table III lists the average precision, recall and F_1-score of *TraceSieve* and baseline approaches on the two datasets, where WT, IS, PT represent waiting time, invocation structure and processing time. *TraceSieve* outperforms all the baseline approaches on both datasets, with the average best F_1-score of 0.97 and 0.925, respectively. To evaluate the effect on different types of trace anomaly, we classify the anomalies and detect them respectively. In order to ensure the accuracy of our detection of each type of anomaly, we ensure that each anomalous trace in the testing set contains only one corresponding anomaly type. Moreover, we also combine the different anomalies together and calculate the average precision, recall and F_1-score.

Due to the baseline approaches only extract response time from traces, they can not detect anomalous traces which have a processing time anomaly. AEVB achieves the lowest performance on $\mathscr{D}1$. Meanwhile, MultimodalTrace achieves lowest on $\mathscr{D}2$. They both use LSTM based deep learning model to detect trace anomalies. However, both of them regard traces as sequences, which mainly reconstruct using timestamps recorded. According to the results, AEVB and MultimodalTrace are relatively good at detecting execution time anomalies that happened in a trace, but failed to detect invocation structure anomalies. CFG focuses on execution time anomaly detection, while CPD focuses on invocation structure anomaly detection. Both of them can only detect one type of trace anomaly. TraceAnomaly do better at detecting execution time anomaly than invocation structure anomaly. The overall detection performance of TraceCRL is better than TraceAnomaly on both datasets, although its effectiveness in detecting invocation structure anomalies is slightly worse.

TraceSieve is effective in detecting different types of trace anomalies, with the average best F_1-score obviously higher than existing methods. Compared with the best-performing baseline approach, the best F_1-score of *TraceSieve* outperform it by 0.096 and 0.117 on the two datasets, respectively.

As for efficiency, we also evaluate the time cost for model training of all the baseline approaches and *TraceSieve*. Table III lists the time that approaches use to train an anomaly detection model on training set. CFG takes the least amount of time on training, about 90h and 46h, due to its simple calculation of similarity among traces. The same result goes for CPD, which only cost 96h and 48h. The remaining four methods that take the least time are over 5 days and 2 days, respectively. Especially, AEVB has the longest training time, about 683.2h and 314.2h, due to the using of multi-modal LSTM. Although our approach spends about 4.3h and 7.6h on training, we propose a more complex data structure for feature information extraction and construct invocation graph for traces, which compensates for the over-time on training model.

C. Root Cause Localization of TraceSieve (RQ2)

Besides anomaly detection, we also compare the precision of *TraceSieve* in root cause localization with baseline approaches on $\mathscr{D}1$ and $\mathscr{D}2$, listed in Table IV. We can find that the precision of *TraceSieve* in root cause localization is better than baseline approaches, which reaches 0.98 at P@3 on both datasets. *TraceSieve* especially has a good result on P@1, which can demonstrate that *TraceSieve* has strong discrimination for abnormal microservices. Since our results are root-caused features in a trace, we evaluate our model locating the three types of anomalous traces, which is shown in Table V.

TABLE IV
THE PRECISION OF ROOT CAUSE LOCALIZATION IN COMPARISON WITH DIFFERENT APPROACHES

Dataset	Approach	P@1	P@2	P@3
$\mathscr{D}1$	MEPFL [5]	0.41	0.47	0.53
	TraceAnomaly [30]	0.65	-	-
	TraceRCA [29]	0.69	0.72	0.79
	MicroRank [28]	0.76	0.83	0.88
	Sage [31]	0.82	0.86	0.92
	TraceSieve	**0.92**	**0.95**	**0.98**
$\mathscr{D}2$	MEPFL [5]	0.32	0.41	0.49
	TraceAnomaly [30]	0.60	-	-
	TraceRCA [29]	0.67	0.68	0.73
	MicroRank [28]	0.72	0.83	0.85
	TraceSieve	**0.90**	**0.94**	**0.98**

TABLE V
THE F_1-SCORE ($F1$) OF ROOT CAUSE LOCALIZATION IN THREE ANOMALY TYPES ON TWO DATASETS

Dataset	F_1(WT)	F_1(IS)	F_1(PT)
$\mathscr{D}1$	0.98	0.97	0.97
$\mathscr{D}2$	0.91	0.93	0.90

D. Contributions of TraceSieve (RQ3)

We conduct a series of experiments to evaluate the contributions of key components of *TraceSieve*. During these experiments, we construct three alternatives of *TraceSieve*, which are:

(1) ***TraceSieve* using STV.** To show the importance of trace feature matrix (TFM) in feature extraction, we change the TFM into STV used in existing methods.

(2) ***TraceSieve* w/o GAN.** To study the effect of noise filtering in data preprocessing, we remove GAN from *TraceSieve*.
(3) ***TraceSieve* w/o EWC.** To evaluate the performance of incremental training , we remove the elastic EWC from *TraceSieve*.

Table VI lists the average precision, recall and best F_1-score of the three alternative mentioned above on two datasets. When the TFM of *TraceSieve* is replaced by the original STV, both precision and recall decrease. It shows that TFM is more efficient than STV for extracting feature information from traces. Both precision and recall decrease when the denoising module is removed, demonstrating that noise filtering can reduce false positives by preventing *TraceSieve* from learning patterns of anomalous traces. We also injected three types of abnormal trace in different proportions shown in Table VII.

TABLE VI
COMPARISON BETWEEN *TraceSieve* AND DIFFERENT MODEL ALTERNATIVE

Dataset	Approach	P	R	F_1	Time (h)
$\mathscr{D}1$	using STV	0.923	0.917	0.920	3.7
	w/o GAN	0.927	0.941	0.932	4.5
	w/o EWC	0.975	0.986	0.980	60.1
	TraceSieve	0.973	0.968	0.970	4.3
$\mathscr{D}2$	using STV	0.864	0.842	0.853	6.9
	w/o GAN	0.894	0.903	0.898	7.3
	w/o EWC	0.929	0.948	0.938	100.3
	TraceSieve	0.915	0.936	0.925	7.6

TABLE VII
COMPARISON OF DENOISING PERFORMANCE IN DIFFERENT TYPES OF NOISE AT DIFFERENT INJECTION RATIOS

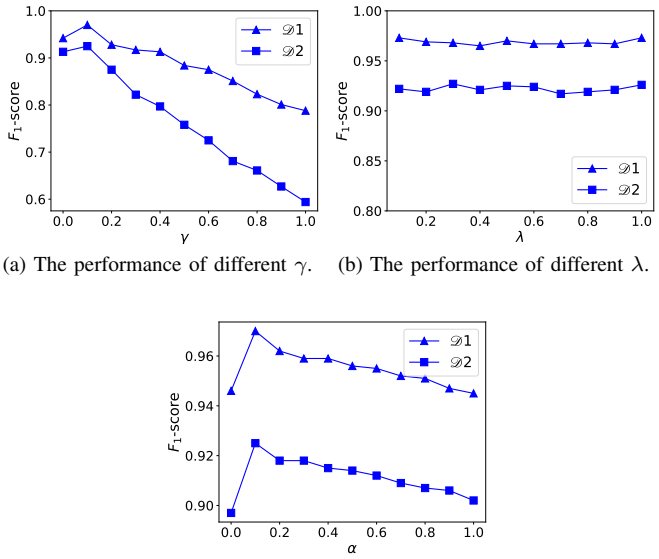
Failure Type	Noise Rate	F_1(ND)	F_1(D)
WT failure	0.1%	0.891	0.947
	0.4%	0.887	0.943
	0.7%	0.862	0.935
	1%	0.858	0.932
PT failure	0.1%	0.883	0.919
	0.4%	0.881	0.907
	0.7%	0.875	0.902
	1%	0.868	0.900
IS failure	0.1%	0.520	0.834
	0.4%	0.517	0.826
	0.7%	0.509	0.818
	1%	0.496	0.803

As for removing EWC, Table VI shows that when *TraceSieve* adds EWC, the precision and recall slightly reduce. But when we consider the training time cost between *TraceSieve* and *TraceSieve* without EWC, we find that time cost in model training decreases remarkably through the incremental training method.

E. TraceSieve Hyperparameters Sensitivity(RQ4)

We mainly discuss the effect of three hyperparameters in data preprocessing and online detection on *TraceSieve*.

Figure 7a shows the average best F_1-score of *TraceSieve* changes with different values of γ, hyperparameter in loss function. Specifically, we increase γ from 0.0 to 1.0. From the results, we can find that when γ is at 0.1, *TraceSieve* has the best average F_1-score. If γ continuously increase, it will affect



(a) The performance of different γ. (b) The performance of different λ.

(c) The performance of different α.

Fig. 7. The effect of different hyperparameters.

the choosing on the threshold of anomaly score, degrading *TraceSieve*'s performance. Thus, we pick 0.1 as the γ.

Then we focus on λ, which is a hyperparameter in EWC during incremental training. Figure 7b shows how the average best F_1-score of *TraceSieve* changes as λ varies. Through the experiment, we change λ from 0.1 to 0.9 with a step size of 0.2. According to the result, we observe that the variation of λ does not impact the accuracy of *TraceSieve* significantly. In this paper, we set λ = 0.5.

We also change the size of α in noise filtering. We gradually increase the value of α. Figure 7c shows the impact of changing α on average F_1-score. We observe that when α is 0.1, the F_1-score reaches the top. Then it keeps decreasing as the α increases. According to the analysis, we set α = 0.1.

V. DEPLOYMENT AND DETECTION

We install and deploy *TraceSieve* in the production environment of a large-scale e-commerce company. First, operators use the developed automated data collection tool to collect RPC log data in the production environment. The tool then pushes the collected log data input into Elastic APM[3], a widely used application performance monitoring system built on the Elastic Stack. It provides data storage and indexing and analysis and visualization of data. Elastic APM processes log data into call chain data and stores it. During offline training, operators use the trace data for the last week to train the model firstly and learn the existing transaction invocation patterns of the company. During online detection, the trace data in the production environment is obtained in real-time from Elastic APM, and the latest trained model is used to detect anomalies. Due to the important and large amount of data generated on weekdays, we adopt the strategy of updating the detection

[3]https://www.elastic.co/

model by incremental training on weekends periodically and conducting online detection on weekdays, which can make better use of new data to improve the effectiveness of the model. Since the scale of abnormal transaction invocation from data is tiny, our *TraceSieve* can achieve full coverage of failures that appeared in data. Since the production environment of this financial trading company is relatively stable, we rarely find abnormal traces. But, we still find the three types of trace anomalies we propose during the online detection stage, shown in Figure 8.

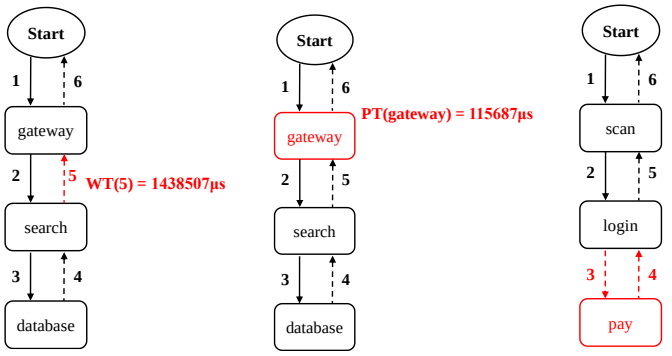


Fig. 8. Different types of anomalous trace in the studied company.

Figure 8a shows an instance of waiting time anomaly that appeared in trace. We can find that the waiting time of Execution 5 reaches over 1,000,000μs, which deviates from the distribution of normal traces.

Figure 8b shows an instance of processing time anomaly that appeared in trace. The waiting time of Service *gateway* reaches 115,687μs, which is significantly above the range of normal waiting times.

Figure 8c shows an instance of structure anomaly that appeared in trace. It is obvious that Execution 3 and Execution 4 disappear in the trace compared to the normal pattern, which represents the invocation between Service *login* and Service *pay* crashes down.

VI. Related Work

Many researchers have contributed to applying traces in the natural production environment. In the microservice system, traces are often used to help maintain the safe operation of the system. Li et al. [32] conduct an empirical survey on the industrial environment and demonstrate that distributed traces are widely used for failure detection in microservice systems. Similarly, Luo et al. [33] makes an excellent effort to analyze distributed traces in the microservice system of Alibaba, which significantly promotes the development of trace anomaly detection.

Trace Anomaly Detection. More and more researchers focused on machine learning methods. AVEB [10] establishes a multi-modal LSTM (Long-Short-Term-Memory) architecture to learn the sequence execution patterns of normal traces. MultimodalTrace [9] is a trace anomaly detection method that presents a deep learning model for sequence learning to model the causal relationship between the service instances in a trace, using the single-modality, sequential text data. It detects dependent and parallel tasks using the model to reconstruct the execution path. TraceAnomaly [7] is a trace anomaly detection method that only considers service-level traces (operations not considered). It adopts posterior flow based variational autoencoders (VAE) to detect anomalous traces. It extracts the execution time of each span from traces and reconstructs invocation path through parent node to create service trace vector (STV). TraceCRL [11] uses representation learning approach based on contrastive learning and graph neural network to incorporate graph-structured information in the downstream trace analysis tasks.

Trace-based Root Cause Localization. At first, MonitorRank [34] put forward a method to automatically find root causes of failures in the service architecture based on the generated call graph. Then Microhecl [30] localize root cause with a dynamic call graph. Homoplastically, MicroRank [28] propose a new strategy, in root cause localization based on extended-spectrum techniques. AID [35] propose an approach to predict the intensity of dependencies between cloud services by aggregating all invocation pairs which the current service candidates. These methods utilize the trace structural diagram to proceed with root cause localization. Nonetheless, they do not fully use the feature information contained in traces.

VII. Conclusion

This paper proposes *TraceSieve*, a trace anomaly detection approach through constructing a trace data graph. The approach performs online and continuous analysis of trace data produced by a running microservice system, and returns anomalous traces and services. *TraceSieve* leverages data graphing to discover potential anomalous invocations in evolving trace data and uses graph similarity analysis to localize anomalous services based on the trace graph. Moreover, *TraceSieve* can effectively incorporate online feedback from operators based on trace data in a natural production environment to improve the accuracy of detecting anomalous traces. Our evaluation confirms that *TraceSieve* can effectively detect anomalies and localize anomalous services in a microservice system. Furthermore, *TraceSieve* is efficient and can be further improved by denoising trace data.

In future work, we plan to explore more advanced learning-based trace representation methods to improve the effectiveness of trace anomaly detection and uncommon service localization. We also plan to evaluate *TraceSieve*'s effectiveness and efficiency on more extensive and complex real-world microservice systems.

VIII. Acknowledge

The work was supported in part by the Advanced Research Project of China (No.31511010501), National Natural Science Foundation of China (Grant No.62272249, 62072264), and Natural Science Foundation of Tianjin (Grant No.21JCQNJC00180).

REFERENCES

[1] F. Li, "Cloud native database systems at alibaba: Opportunities and challenges," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2263–2272, 2019.

[2] J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," 2014, last checked 12 August 2022. [Online]. Available: https://www.martinfowler.com/articles/microservices.html

[3] T. Alsop, "Average cost per hour of enterprise server downtime worldwide in 2019," https://www.statista.com/statistics/753938/worldwide-enterprise-server-hourly-downtime-cost/.

[4] P. Las-Casas, J. Mace, D. Guedes, and R. Fonseca, "Weighted sampling of execution traces: Capturing more needles and less hay," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 326–332.

[5] 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*, 2019, pp. 683–694.

[6] X. Guo, X. Peng, H. Wang, W. Li, H. Jiang, D. Ding, T. Xie, and L. Su, "Graph-based trace analysis for microservice architecture understanding and problem diagnosis," 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*, 2020, pp. 1387–1397.

[7] 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*, 2020, pp. 48–58.

[8] 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*, 2021, pp. 436–446.

[9] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection from system tracing data using multimodal deep learning," in *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*, 2019, pp. 179–186.

[10] Sasho Nedelkoski and Jorge Cardoso and Odej Kao, "Anomaly detection and classification using distributed tracing and deep learning," in *19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2019, Larnaca, Cyprus, May 14-17, 2019*, 2019, pp. 241–250.

[11] 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*, 2022, pp. 1221–1232.

[12] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, 2019, pp. 19–33.

[13] D. Josephsen, "ivoyeur: Opentracing," *login Usenix Mag.*, vol. 43, no. 1, 2018.

[14] Jaegertracing.io, "Jaeger," 2022, last accessed 1 August 2022. [Online]. Available: https://www.jaegertracing.io/

[15] Twitter, "Zipkin," 2022, last accessed 1 August 2022. [Online]. Available: https://zipkin.io/

[16] A. SkyWalking, "Skywalking," 2022, last accessed 30 July 2022. [Online]. Available: https://skywalking.apache.org/

[17] Opentracing.io, "Opentracing," 2022, last accessed 31 July 2022. [Online]. Available: https://opentracing.io/

[18] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[19] T. N. Kipf and M. Welling, "Variational graph auto-encoders," *CoRR*, vol. abs/1611.07308, 2016.

[20] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, 2014, pp. 2672–2680.

[21] H. Xu, W. Chen, J. Lai, Z. Li, Y. Zhao, and D. Pei, "Shallow vaes with realnvp prior can perform as well as deep hierarchical vaes," in *Neural Information Processing - 27th International Conference, ICONIP 2020, Bangkok, Thailand, November 18-22, 2020, Proceedings, Part V*, ser. Communications in Computer and Information Science, vol. 1333, 2020, pp. 650–659.

[22] J. Kirkpatrick, R. Pascanu, N. C. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell, "Overcoming catastrophic forgetting in neural networks," *CoRR*, vol. abs/1612.00796, 2016.

[23] R. K. Srivastava, J. Masci, S. Kazerounian, F. J. Gomez, and J. Schmidhuber, "Compete to compute," in *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, 2013, pp. 2310–2318.

[24] Scratchapixel, "Monte carlo integration," last checked 12 October 2022. [Online]. Available: https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/monte-carlo-integration

[25] E. T. Nalisnick, A. Matsukawa, Y. W. Teh, D. Görür, and B. Lakshminarayanan, "Do deep generative models know what they don't know?" in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[26] A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, and S. Bhattacharya, "Anomaly detection using program control flow graph mining from execution logs," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, 2016, pp. 215–224.

[27] L. Bao, Q. Li, P. Lu, J. Lu, T. Ruan, and K. Zhang, "Execution anomaly detection in large-scale systems through console log analysis," *J. Syst. Softw.*, vol. 143, pp. 172–186, 2018.

[28] G. Yu, P. Chen, H. Chen, Z. Guan, Z. Huang, L. Jing, T. Weng, X. Sun, and X. Li, "Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments," in *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, 2021, pp. 3087–3098.

[29] Z. Li, J. Chen, R. Jiao, N. Zhao, Z. Wang, S. Zhang, Y. Wu, L. Jiang, L. Yan, Z. Wang, Z. Chen, W. Zhang, X. Nie, K. Sui, and D. Pei, "Practical root cause localization for microservice systems via trace analysis," in *29th IEEE/ACM International Symposium on Quality of Service, IWQOS 2021, Tokyo, Japan, June 25-28, 2021*, 2021, pp. 1–10.

[30] D. Liu, C. He, X. Peng, F. Lin, C. Zhang, S. Gong, Z. Li, J. Ou, and Z. Wu, "Microhecl: High-efficient root cause localization in large-scale microservice systems," in *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*, 2021, pp. 338–347.

[31] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: practical and scalable ml-driven performance debugging in microservices," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, 2021, pp. 135–151.

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

[33] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, 2021, pp. 412–426.

[34] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture," in *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '13, Pittsburgh, PA, USA, June 17-21, 2013*, 2013, pp. 93–104.

[35] T. Yang, J. Shen, Y. Su, X. Ling, Y. Yang, and M. R. Lyu, "AID: efficient prediction of aggregated intensity of dependency in large-scale cloud systems," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, 2021, pp. 653–665.