Real-Time Anomaly Detection Using Distributed Tracing in Microservice Cloud Applications

Mahsa Raeiszadeh[†], Amin Ebrahimzadeh[†], Ahsan Saleem[†], Roch H. Glitho[†**], *Senior Member, IEEE*, Johan Eker[*], and Raquel A. F. Mini[*]

[†]CIISE, Concordia University, Montréal, QC, Canada

[*]Ericsson Research, Lund, Sweden

[**]Computer Science Programme, University of Western Cape, Capetown, South Africa

***Abstract*—Distributed tracing plays a vital role in microservice infrastructure, and learning-based trace analysis has been utilized to detect anomalies within such systems. However, existing approaches for learning-based trace-based anomaly detection face certain limitations. Some assume that trace patterns can be learned solely from normal executions, while others depend on anomaly injection to generate labeled traces categorized as normal or anomalous. However, in practical scenarios, anomalies may also happen during the normal execution. Moreover, a wide variety of anomalies may occur in practice, which cannot be captured solely through anomaly injection. To address these issues, we propose a Trace-Driven Anomaly Detection (TDAD) approach based on a Span Causal Graph (SCG) representation, which trains a model using a Graph Neural Network (GNN) and Positive and Unlabeled (PU) learning. This technique allows the model parameters to be optimized by estimating the underlying data distribution. As a result, TDAD can be effectively trained using a small number of labeled anomalous traces along with a relatively large number of unlabeled traces. Our evaluation reveals that TDAD outperforms not only the existing unsupervised trace-based anomaly detection methods by 11.9% in terms of F_1-score but also a supervised learning-based benchmark by 12x in terms of detection time.**

***Index Terms*—Anomaly Detection, Distributed Tracing, Microservice, Positive and Unlabeled Learning.**

I. Introduction

Industrial microservice systems typically encompass large-scale distributed architectures, housing hundreds to thousands of services within complex cloud infrastructures. These systems experience dynamic creation and destruction of service instances, necessitating prompt and efficient analytics capabilities within a bounded time. To attain the required observability in highly intricate and dynamic microservice environments, it is crucial to trace the flow of requests among services. To this end, distributed tracing has started to play a vital role in industrial microservice systems [1]. Typically, it is implemented as a pipeline that facilitates the collection, preprocessing, and storage of traces [2]. These traces serve as the foundation for a range of trace analysis techniques that aim to comprehend system behaviors [3], detect anomalies [4], and diagnose faults [5] within microservice architectures.

Detection of application anomalies in a microservice system is complicated due to the following challenges. First, microservice systems are highly dynamic with rapid updates and continuous integration and deployment of new features. These systems often run in a containerized environment, where container states frequently change from busy to idle, adding to the difficulty of performance diagnosis. Debugging in microservice systems is exceptionally challenging due to their intricate and dynamic nature. Developers face the task of analyzing concurrent behaviors across multiple microservices and comprehending the overall system interaction topology. Second, microservice architecture involves multiple intricately interconnected fine-grained services that are distributed loosely across the system, resulting in complex tracing paths. Additionally, each microservice may have multiple instances to handle requests, thus further increasing the complexity of trace paths. Therefore, tracing and visualizing system executions are fundamental and efficient methods for understanding and debugging distributed systems [6]. Traces generated from system executions provide valuable insights into runtime service dependencies and facilitate the analysis of request execution across various services [2]. These traces also capture important details on service invocations such as status codes and duration times. As a result, traces are widely used to identify potential anomalies by examining their structural aspects, such as the presence of missing service invocations and performance metrics (e.g., latency).

Recently, a few works have focused on learning-based trace analysis for anomaly detection in microservice systems [7]–[11]. Unsupervised learning approaches rely on the assumption that patterns within traces can be learned only from normal system executions, while supervised learning approaches depend on injecting anomalies into the system to generate labeled traces. We note, however, that unsupervised methods struggle to ensure that normal traces are truly anomaly-free, while supervised methods require a large number of labeled traces, and rely on time-consuming anomaly injection processes.

To tackle the aforementioned issues, we propose our Trace-Driven Anomaly Detection (TDAD), which uses a Graph Neural Network (GNN) to enable learning vector representations for traces and Positive and Unlabeled (PU) learning that allows for training a trace-based anomaly detection model with a partially labeled dataset. Our proposed TDAD represents a trace as a Span Causal Graph (SCG) that encompasses a complex hierarchy structure. In order to detect anomalies in microservice system traces, we employ a graph-based approach, where each node in the graph is associated with three distinct information types, including the semantics of the operation name, time-related attributes, and status code. To develop distinct representations for each graph, we use a combination of a graph attention network and a PU learning-based model for trace-based anomaly detection. Model parameters are optimized by estimating the empirical risk on the historical data, which serves as a measure of the expected loss on the training data using PU learning. Leveraging historical anomalous traces, TDAD trains the anomaly detection model using a small number of labeled anomalous traces along with a relatively large number of unlabeled traces, thus being able to detect anomalies in a timely manner. The main contributions of this paper are summarized as follows.

- We define a trace representation using an SCG, incorporating the hierarchical structure and contextual information of spans which can be obtained via a parallel processing approach.
- We present a trace-based anomaly detection method that utilizes a combination of GNN and PU learning. This approach leverages historical anomalous traces and only relies on a small number of labeled anomalous traces.
- We have carried out a comprehensive experimental evaluation on a real-world microservice benchmark to evaluate the performance of our proposed TDAD in terms of detection accuracy and detection time.

The remainder of this paper is structured as follows. Section II reviews related work. Section III describes the system model, problem statement, and the proposed method. Evaluation results are presented in Section IV. Section V concludes the paper.

II. Related Work

Distributed trace-based anomaly detection methods can be classified into two categories (a) Machine Learning-Based and (b) Trace Comparison methods. There are several factors that determine the classifications, including the availability of data, system requirements, and desired capabilities. In the following, we review each category in detail.

A. Machine Learning-Based Methods

Machine learning-based anomaly detection methods are classified into two groups of supervised and unsupervised learning methods.

1) Supervised Learning: The authors of [8] proposed the so-called Seer, which is a deep neural network that uses convolutional and LSTM layers to detect performance anomalies in application services. Seer receives key performance indicators (KPIs) such as latency, outstanding requests, and resource consumption from distributed traces and node interactions. The output neurons identify the affected services. Seer continuously processes traces to detect anomalies, communicates with the node runtime to identify saturated resources, and notifies the system manager to mitigate performance degradation by allocating additional computing resources. In [9], dual neural networks for service anomaly detection were proposed. The first network was a variational autoencoder trained on normal traces to identify anomalies through reconstruction errors. The second network was a convolutional neural network trained on failure-injected traces to recognize the specific failure-causing anomalies. False positives were filtered out during the post-processing of the autoencoder's output. The convolutional network determines the type of anomaly when a service is considered anomalous. Bogatinovski et al. [10] proposed an anomaly detection approach based on event dependencies. A self-supervised encoder-decoder network was trained to identify events in hidden positions by considering nearby events. During anomaly detection, the network generated expected event lists for each position in a new trace. The post-processing stage flagged genuinely logged events as anomalies if they were absent from their expected positions. Anomaly scores were calculated based on the ratio of anomalous events to trace length, and if the score exceeded a user-defined threshold, a functional anomaly in the application was indicated.

2) Unsupervised Learning: In TraceAnomaly [7], a deep Bayesian neural network with the posterior flow was presented for anomaly detection. The network determines the likelihood of a trace being normal. It stores observed service call paths and sequences of service interactions in traces. During online anomaly detection, TraceAnomaly checks for previously unseen call paths. If found, it evaluates them for functional anomalies using a whitelist. If no functional anomalies are detected, the trace is forwarded to a Bayesian neural network, which determines the probability that the trace is normal. If the probability is below a threshold, the trace is considered a performance anomaly. Microscope [11] is an application-level performance anomaly detection model. It monitors KPIs at the front-end of a microservice application and compares them against specified Service Level Objectives (SLOs). Deviations from the specified SLOs are detected by Microscope as performance anomalies affecting the application. Jin et al. [4] introduced RPCA, which is an offline anomaly detection solution for microservice applications. They analyzed distributed tracing traces to detect performance anomalies, considering various metrics such as CPU and memory consumption. Principal component analysis was employed to identify services involved in anomalous interactions. Performance metrics were collected using unsupervised learning algorithms, and anomaly values were detected by applying a linear function to the principal components of anomalous traces. Anomaly scores were assigned to services connected to anomalous traces, and a list of impacted services was generated based on a predefined threshold. Services were ranked according to their anomaly scores.

B. Trace Comparison Methods

Wang et al. [5], Meng et al. [12], and Chen et al. [13] utilized trace comparison as a technique for online anomaly detection in microservice applications with distributed tracing instrumentation. The techniques involved collecting possible traces in a microservice application and then comparing newly collected traces with the existing ones. This work is based

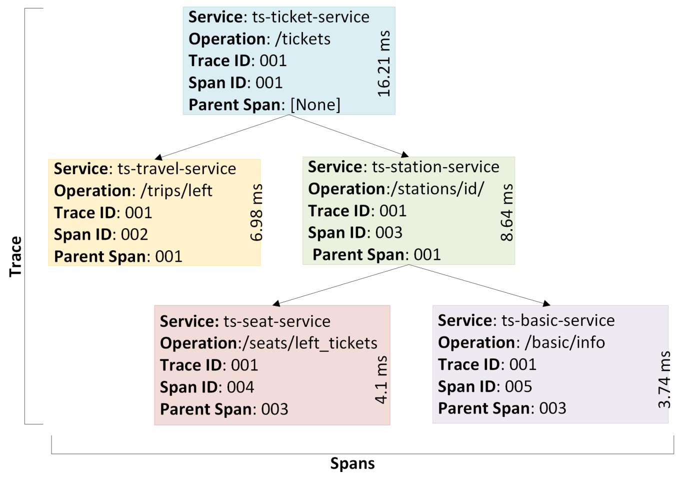


Fig. 1. Example of trace and spans in TrianTicket microservice application.

on the assumption that the application maintains a consistent behavior compared to the past runs. Thus, previously collected traces serve as a reference for comparison. However, anomaly detection may be less accurate if the runtime conditions of the application differ from those under which the reference traces were acquired. Wang et al. [5] and Meng et al. [12] proposed trace-based anomaly detection methods for microservice applications. They collect traces in a pre-production environment, build representative call trees, and detect anomalies by computing tree-edit distances and analyzing response times. These approaches are computationally expensive and thus suitable only for offline detection scenarios. In contrast, Chen et al. [13] introduced a trace comparison-based method using a fast matrix sketching algorithm. By comparing response times with sketch vectors, anomalies are detected efficiently with reduced false positives/negatives. The sketch vectors were updated to adapt to changing runtime conditions, enabling online anomaly detection.

The literature on anomaly detection in microservice applications commonly employs machine learning algorithms, which are either supervised or unsupervised [7]–[11]. Both unsupervised and supervised learning-based approaches in anomaly detection have limitations. Unsupervised methods assume that most training traces are normal, but this assumption may not be valid in practice, and incorporating historical anomalous traces is challenging. Supervised methods rely on a large number of labeled traces, which necessitates the need for a laborious and time-consuming anomaly injection process and may struggle to cover diverse types of normal and anomalous traces. Some works are based on trace comparison methods, which compare newly generated traces with stored traces to identify similarities. However, these methods can be time-consuming, limiting their suitability to promptly detect and respond to anomalies in real-time, e.g., [5] [12]. The research gap lies in the need to train the anomaly detection model with a small number of labeled anomalous traces. Also, there is a need for efficient real-time anomaly detection methods that can effectively detect anomalies in microservice applications while the system is running, thus allowing for a timely response to failures.

III. System Model, Problem Statement, and Proposed Solution

A. System Model and Problem Statement

Distributed tracing systems commonly adhere to the OpenTracing specification, which establishes a language-independent data structure and a collection of principles for distributed tracing[1]. OpenTracing, a project under the Cloud Native Computing Foundation (CNCF), provides an API specification and a range of frameworks and libraries that have implemented this specification [2]. Fig. 1 presents an illustrative example, depicting an instance of a trace that conforms to the OpenTracing specification. Based on the OpenTracing specification, a trace refers to a sequential representation of the steps involved in processing a request across multiple service instances. Each step in this workflow is called a span and captures the context of a service operation. Each trace in the system is assigned a unique trace ID, and spans within a trace are identified by their own unique span ID, along with the ID of their parent span indicating the preceding span in the sequence. Moreover, a span contains information about both the caller operation and the callee operation involved in the current invocation. As shown in Fig. 1, the caller operation of Span 004 is Span 003, its callee operation is Span 004, and its parent span is Span 003. The span captures the initiation and completion times of the server-side invocation, while the parent span records the corresponding initiation and completion times on the client-side for the same invocation. For instance, Span 002 captures the server-side initiation and completion times between Span 001 and Span 002, whereas its parent span (Span 001) captures the client-side initiation and completion times for the identical invocation. Trace i is denoted by $T_i = \left(s_1^i, \ldots, s_L^i\right)$, $i \in \{1, \ldots, Z\}$, from Z time instant, where s_j^i is span j of trace i and L is the length of T_i, i.e., the number of spans in trace i . With these considerations in mind, we aim to solve the problem of determining whether trace T_i is an anomaly or not in real time with the main objective of maximizing detection accuracy.

B. Proposed Solution

Fig. 2 illustrates an overview of our proposed TDAD method, which comprises four main components: (i) Span Embedding, (ii) Graph Building, (iii) Model Training, and (iv) Anomaly Detection. First, a vector representation, including semantic information, is generated for each individual span through the process of Span Embedding. Second, Graph Building constructs an SCG for each trace, capturing the relationship among the spans. Third, the Model Training of trace-based anomaly detection uses GNN and PU learning, which enables the learning of a vector representation for

[1]See https://opentracing.io/ for further information.

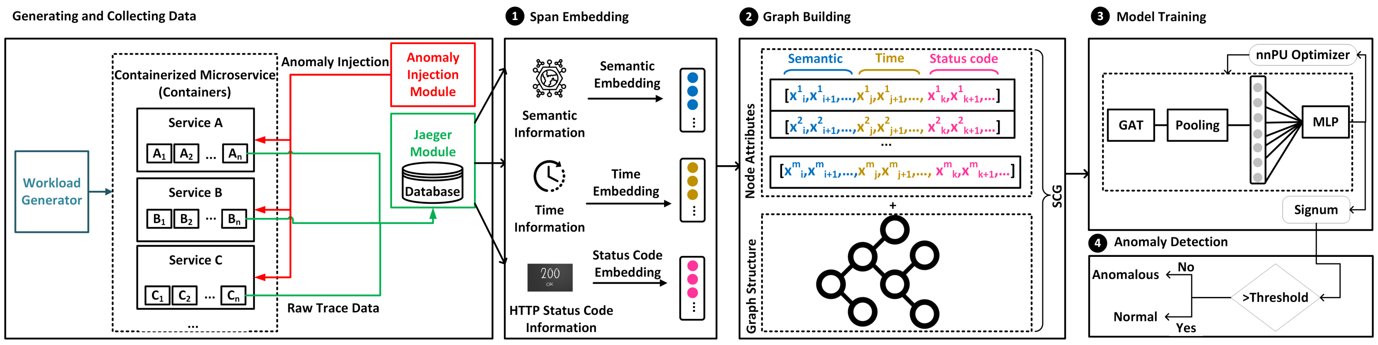


Fig. 2. Overview of TDAD.

each trace derived from its SCG representation. By using PU learning, the model can be trained using a small number of labeled anomalous traces, estimating the empirical risk based on the available data. Next, the trained model is fed with the SCG, employing a signum function in the output layer to generate the final outcome. Finally, TDAD produces a prediction indicating the presence of anomalies. In the following, we describe each component in more detail.

1) Span Embedding: A trace comprises multiple spans that depict the causal connections between various service invocations. These spans are associated with each service invocation, recording details such as service and operation names, start time, duration, and status code. The process of span embedding involves capturing and encoding these invocation contexts to form the trace graph representation. Specifically, it encodes service and operation names, start and duration times, and status codes separately. These components are then combined into a vector representation for each span, which serves as a basis for further analysis and processing of the trace data. Span embedding comprises three main steps: (i) Semantic Embedding, (ii) Time Embedding, and (iii) Status Code Embedding, which are explained in greater detail next.

a) Semantic Embedding: The semantics embedding part is responsible for generating a vector representation for each span of a trace including the concatenated service name and operation name. Firstly, we split the names into words using common separators in microservices (e.g., "/", "-", ":"). Next, all words are transformed to lowercase, removing non-verbal symbols such as punctuation marks and numbers. For instance, the name "ts-travel-service/POST:/api/v1/travelservice/travelPlan/cheapest" becomes "ts", "travel", "service", "post", "api", "v1", "travelservice", "travelplan", "cheapest". To handle the dynamic nature of service and operation names in microservices systems, we employ the WordPiece algorithm [14] for tokenization, which splits words into tokens based on character combinations. This helps address the presence of out-of-vocabulary (OOV) words. For example, "traveldate" and "trainnumber" can be split into tokens like "travel", "date", "train", and "number". Based on the obtained sequence of tokens, we employ a pre-trained BERT model [15], which is a transformer-based language representation model, to generate a vector representation. Specifically, we use the BERT-Base model [16], which uses 12 transformer encoder layers and a hidden layer with 768 dimensions. This model generates a 768-dimensional vector representation for each span, effectively capturing the semantics of its service and operation name. Employing the BERT-Base model [16], which is equipped with 12 transformer encoder layers to capture intricate patterns and dependencies in trace data alongside its 768-dimensional hidden layer for representing nuanced features, has proven beneficial for tasks demanding a deeper comprehension of language semantics and potential performance improvement.

b) Time Embedding: Every span within the system captures essential information about a service invocation, including the start time and duration of the interaction between the caller and callee. In order to enhance the detection of anomalies related to time, we extract four distinct time-related attributes from these recorded timestamps, namely, (1) duration time, which refers to the span duration, (2) waiting time, which indicates how long the callee waits for a response from other services, (3) local execution time, which represents the time taken by the callee to perform the current invocation, excluding the waiting time, and (4) relative start time, which is the time difference between the start time of the current span and the start time of its root span.

Given the use of the BERT-Base model, each span is represented by a 768-dimensional vector. This vector represents the span and includes the embedding of service and operation names. However, if each time feature occupies only a single dimension within this 768-dimensional vector, there is a possibility of overlooking the time-related attributes in the span representation. Moreover, the considerable variation in time across different traces (which may range from a few dozen to several thousand milliseconds) poses challenges in achieving weight convergence and training efficiency of the model. To address these issues, we project a single-dimensional time feature $\boldsymbol{t}$ into a d-dimensional vector space denoted by $\boldsymbol{E}_{\text{time}}$. Subsequently, we use the softmax function to create a soft one-hot encoding, where each element lies between 0 and 1 and the sum of all elements is equal to 1. The soft one-hot encoding vector $\boldsymbol{s}$ is obtained as follows:

$$\mathbf{s}=\tau(\boldsymbol{t}\boldsymbol{W}+\boldsymbol{b}), \tag{1}$$

where $\tau(\cdot)$ is the softmax function, $\boldsymbol{W} \in \mathbb{R}^p$ is the weight matrix, and $\boldsymbol{b} \in \mathbb{R}^p$ is the bias. Next, we project the vector $\boldsymbol{s}$ into a vector space specifically designed for time embeddings. The soft one-hot encoding $\boldsymbol{s}$ is multiplied by the time embedding vector $\boldsymbol{E}_s \in \mathbb{R}^{p\times d}$, which results in p-dimensional vector $\boldsymbol{E}_{\text{time}}$ as follows:

$$\boldsymbol{E}_{\text{time}} = \mathbf{s} \odot \boldsymbol{E}_s, \tag{2}$$

where $\odot$ denotes the element-wise multiplication of two vectors of the same length. Finally, to create a comprehensive representation of the time-related attributes within a span, we concatenate the four time embedding vectors, i.e., duration time, waiting time, local execution time, and relative start time (which are defined above). This combined representation effectively captures and encodes the temporal information associated with the span.

c) Status Code Embedding: The HTTP/1.1 standard [17] outlines a comprehensive list of 63 status codes categorized into five distinct groups. We utilize one-hot encoding to embed status codes, wherein each status code is represented by a 63-dimensional vector. This means that each dimension corresponds to a specific status code. For instance, a status code 200 can be represented as a vector with a value of 1 on the 5th dimension and 0 on all other dimensions. Similarly, a status code 404 can be encoded as a vector with a value of 1 on the 28th dimension and 0 on all other dimensions. This method of encoding status codes enables an efficient analysis of HTTP traffic.

2) Graph Building: Traces exhibit a hierarchical structure that includes service invocations or spans. In our proposed TDAD method, this hierarchical structure can be effectively represented by SCG, which is a directed acyclic graph (see Fig. 2). In this graph, each node represents a span within the trace, while the edges indicate the parent-child relationship between spans. To capture the characteristics of each span, its vector representation is used as an attribute of the corresponding graph node.

3) Model Training: In our proposed approach, trace-based anomaly detection is formulated as a PU learning problem, leveraging historical knowledge of anomalous traces while minimizing the number of labeled traces used for training. PU learning involves training a binary classifier using a small number of positive samples (i.e., anomalous traces) and a relatively large number of unlabeled samples. Essentially, traces are represented as SCGs with span embeddings as node attributes. To obtain meaningful representations of the traces, we employ a graph neural network called Graph Attention Network (GAT), which leverages the multi-head self-attention mechanism [18]. We employ GAT mainly because it aligns well with the characteristics of the trace data (graph structure), learning trace vector representations, and the need for capturing complex relationships and dependencies within the graph. These graph neural networks learn vector representations of the traces, as depicted in Fig. 2. In order to train the binary classifier for trace-based anomaly detection, we employ the non-negative risk estimator (nnPU) algorithm [19], which exhibits robustness against overfitting.

Let $g=\{V,A,X\}$ represent an SCG, where V is the set of nodes, A is the adjacency matrix indicating the edges and X is the node attribute set with each attribute x_i representing the node's vector representation. In the context of an SCG, the GAT layer calculates attention scores for adjacent nodes, indicating their relative importance. We obtain the attention score e_{ij} from node j to node i as follows:

$$e_{ij}=\psi\left(\mathbf{a}^T\cdot\left(\mathbf{W}h_i \| \mathbf{W}h_j\right)\right), \tag{3}$$

where $\psi(\cdot)$ is the LeakyReLU activation function, $\|$ represents concatenation, h_i and h_j denote the vector representations of node i and node j, respectively. The weight matrix $\mathbf{W} \in \mathbb{R}^{F'\times F}$ corresponds to a shared linear transformation, where F is the dimensionality of the input node features and F' is the dimensionality of the transformed features. $\mathbf{a}$ is a learnable attention vector. To obtain the attention coefficients α_{ij} from node j to node note i, the softmax operation is applied to normalize the importance among all neighboring nodes of i:

$$\alpha_{ij}=\frac{\exp\left(e_{ij}\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(e_{ik}\right)}, \tag{4}$$

where $\mathcal{N}_i$ represents the neighborhood of node i. The GAT utilizes multi-head attention to enhance the stability of the attention mechanism's learning process. The attention coefficients are used to compute the output node representation h_i' as follows:

$$h_i'=\|_{k=1}^{K}\sigma\left(\sum_{j\in\mathcal{N}_i}\alpha_{ij}^k\mathbf{W}^k h_j\right), \tag{5}$$

where K represents the number of involved attention heads. Each attention head α_{ij}^k possesses its own attention score. The weight matrix $\mathbf{W}^k$ corresponds to the linear transformation associated with attention head k. Additionally, $\sigma(\cdot)$ denotes the activation function.

After performing the calculations through m GAT layers, TDAD acquires vector representations for all nodes. The overall graph representation v_g is obtained as the average of the vectors over all nodes:

$$v_g=\frac{1}{N_g}\sum_{n=1}^{N_g}h_n^m, \tag{6}$$

where N_g denotes the number of nodes in graph g. The vector representation h_n^m represents the output of node n from the embeddings of GAT layer m.

During the training phase, our proposed TDAD employs the non-negative risk estimation derived from nnPU [19], which is a large-scale PU learning approach to iteratively optimize the GAT parameters. In each epoch, the training set is divided into N mini-batches, and TDAD adjusts the GAT parameters based on the risk estimation for each mini-batch. Suppose we have a two-layer perceptron (MLP) function f with an output dimension of 1. Let L represent the sigmoid loss function. In each mini-batch, there are n_p labeled anomalous traces. We use the symbol p to denote the positive data, while u represents the unlabeled data. The estimated risk $\widehat{R}_{\mathrm{p}}^{+}$ associated with labeled anomalous traces is obtained as follows:

$$\widehat{R}_{\mathrm{p}}^{+} = \frac{1}{n_p} \sum_{i=1}^{n_p} L\left(f\left(v_i^p\right), +1\right). \tag{7}$$

Similarly, the estimated risk $\widehat{R}_{\mathrm{p}}^{-}$ associated with unlabeled anomalous traces is obtained as follows:

$$\widehat{R}_{\mathrm{p}}^{-} = \frac{1}{n_p} \sum_{i=1}^{n_p} L\left(f\left(v_i^p\right), -1\right). \tag{8}$$

We calculate the estimated risk $\widehat{R}_{\mathrm{u}}^{-}$ for unlabeled traces by treating them as normal as follows:

$$\widehat{R}_{\mathrm{u}}^{-} = \frac{1}{n_u} \sum_{i=1}^{n_u} L\left(f\left(v_i^u\right), -1\right), \tag{9}$$

where n_u is the number of unlabeled traces. Finally, the empirical risk estimation $\widehat{R}_{\mathrm{pu}}$ is given by:

$$\widehat{R}_{\mathrm{pu}} = \pi_{\mathrm{p}} \widehat{R}_{\mathrm{p}}^{+} + \widehat{R}_{\mathrm{u}}^{-} - \pi_{\mathrm{p}} \widehat{R}_{\mathrm{p}}^{-}, \tag{10}$$

where the hyperparameter π_{p} denotes the class prior probability of anomalous traces. Let β be the hyperparameter which ensures the risk is non-negative. If $\widehat{R}_{\mathrm{u}}^{-} - \pi_{\mathrm{p}} \widehat{R}_{\mathrm{p}}^{-} \geq -\beta$, TDAD uses Adam [20] optimization to minimize $\widehat{R}_{\mathrm{pu}}$ and optimize GAT parameters; otherwise, Adam is used to minimize $\pi_{\mathrm{p}} \widehat{R}_{\mathrm{p}}^{-} - \widehat{R}_{\mathrm{u}}^{-}$ and optimize the GAT parameters. It is worthwhile to mention that we employ PU learning, which is a well-known semi-supervised approach, for the purpose of learning representations for each graph and optimizing model parameters by estimating the empirical risk based on the data.

4) Anomaly Detection: The anomaly detection component is responsible for generating the final outcome. After being fed to the trained model, the trace is considered normal when the outcome is equal to or greater than 0; otherwise, it is considered as anomalous.

IV. Results

In this section, we first describe our implementation details and then present our findings.

A. Implementation Details

We provide an overview of our experimental testbed, the benchmark application in use, load generation and trace collection procedures, anomaly injection specifications, and the parameter settings and coding environment utilized for implementing the solution under study.

1) Experimental Testbed: We have set up a lab testbed environment within Ericsson Research's private cloud, also known as Xerces. The Xerces private cloud operates a vast infrastructure of around 300 servers managed by an Infrastructure-as-a-Service (IaaS) OpenStack platform. Our testbed comprises a Kubernetes cluster composed of four Virtual Machines (VMs) running Ubuntu 20.04. In the central site cluster, one VM takes on the role of the master node, while three VMs serve as worker nodes. The monitoring of VMs within the Kubernetes clusters and data collection were performed using Jaeger[2].

2) Benchmark Application: In our evaluations, we considered TrainTicket [6], which is a dynamic real-world microservice benchmark. This benchmark encompasses the essential functionalities of train ticket booking, including ticket inquiries, reservations, payments, changes, and user notifications. TrainTicket follows microservice design principles and incorporates various modes of interaction such as synchronous and asynchronous invocations as well as message queues. The system comprises 41 business logic microservices (excluding database and infrastructure microservices) with the explicit purpose of facilitating the examination and experimentation of existing microservice and cloud-native technologies.

3) Load Generation and Trace Collection: We simulate various user behaviors to create realistic workloads. These behaviors include users who only visit the homepage and search for trains, while others log in and book tickets. Also, we dynamically adjust the number of simulated users per behavior over time. This approach generates a mixture of different request types that change dynamically, closely resembling real-world scenarios. We utilize Production and Performance Testing-based Application Monitoring (PPTAM) [21] as our load generator, which incorporates 5 distinct user types. We made slight modifications to PPTAM to ensure that the number of users for each request type changes continuously during runtime. Workloads were uniformly generated for each request type, covering all microservice benchmarks (see Fig. 2). We employ OpenTracing to track the sequential traces of request processing across multiple microservices. In order to ensure uniformity in the data format for collecting execution trace information, we used Jaeger, which is a distributed tool specifically designed to support OpenTracing. We collect 189,486 execution traces from the microservice application.

4) Anomaly Injection: We implemented an anomaly injector, providing configurability for the injection targets, anomaly types, injection time, duration, and intensity. The injector is specifically developed to be packaged within microservice containers as a file-system layer, allowing for remote activation during the training phase. It includes different types of anomalies that have the potential to violate the SLOs, as shown in Table I. Anomalies of different types are randomly injected into containers, with adjustable injection timing and intensity. The time interval for anomaly injection follows an exponential distribution with a rate parameter of $\delta = 0.33s^{-1}$, while the

[2]https://www.jaegertracing.io/

TABLE I
CATEGORY OF INJECTED ANOMALIES INTO TRAINTICKET.

Anomaly Category	Anomaly Type	Example
Network Anomaly	Network loss Network delay	A network congestion incident arises, resulting in a notable surge in network latency.
Pod Anomaly	Pod failure CPU stress Memory stress	The container experiences memory depletion, resulting in a notable rise in the response time of service invocations.
Application Anomaly	service invocation failure, Incorrect return results	An error in the implementation of a service introduces a flaw that leads to inaccurate responses during service invocations. For example, modifying the pricing algorithm in the pricing service results in incorrect ticket prices being calculated and returned.

anomaly type and intensity are chosen randomly. 28.7% of the entire trace data consists of anomalous traces.

5) Parameter Setting and Coding Environment: We implemented our TDAD solution using PyTorch 1.10 and Python 3.10.9. The GAT was implemented using PyTorch Geometric 2.2, while the large-scale PU learning algorithm was adapted from nnPU [19], an open-source implementation. Semantic embedding utilized the pre-trained BERT-Base model [16] and WordPiece tokenizer [14]. Time features were projected to 100 dimensions for time embedding. In TDAD, the model parameters are set as follows. The GAT consists of three layers. The first two layers use three attention heads, while the last layer employs one attention head. Batch normalization was applied after each GAT layer. During training, a batch size of 128 was used, and the prior probability π_{p} of positive data was set to 0.15. The hyperparameter β was set to 0 throughout the training process. The training involved 50 epochs using the Adam [20] optimization algorithm with a learning rate of 0.001. We randomly partitioned traces into training, validation, and testing sets with a ratio of 3:1:6. Within the training set, $\sim$ 10% of the anomalous traces were designated as positive samples, accounting for roughly 2% of the entire training set. For the Self-Supervised approach [10], we utilized the same training, validation, and testing sets, but labeled all traces in the training set. To ensure a fair comparison with RPCA [4], TraceAnomaly [7], and Microscope [11], which assume that the training set predominantly consists of normal traces, we adopt a different division strategy. The traces were randomly distributed into three subsets, following a ratio of 3:1:6. The first subset exclusively contained normal traces and served as the training set. The third subset was designated as the testing set. To balance the increase in anomalous traces within the testing set, all the normal traces from the second subset were incorporated into the testing set.

B. Experimental Results

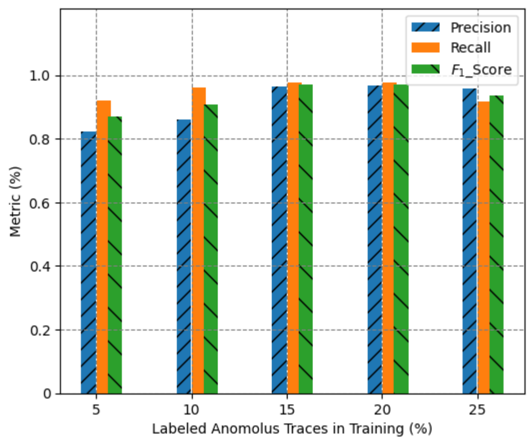
1) Evaluation Metrics: We consider precision, recall, and F_1-score to evaluate the performance of different trace-based anomaly detection methods under study. Moreover, time efficiency is measured using the detection time, which is the time required for processing a single trace data including the time needed for anomaly detection.

TABLE II
PERFORMANCE OF DIFFERENT TRACE-BASED ANOMALY DETECTION METHODS.

Model	Precision (%)	Recall (%)	F_1-score (%)	Detection Time (ms)
RPCA [4]	71.2	92.2	80.2	2.695
TraceAnomaly [7]	65.9	58.0	61.5	1.860
Microscope [11]	39.6	94.3	55.7	4.791
Self-Supervised [10]	92.3	90.2	91.3	0.349
Proposed TDAD	87.1	93.0	89.8	0.131

2) Anomaly Detection Results: As described in Section IV-A5, in TDAD, we adopt a randomized approach to select a subset of 10% anomalous traces from the training set. The selected traces are subsequently labeled as positive samples for the training process. This procedure is repeated in 10 iterations, with each iteration involving training a model using the modified training set and evaluating its performance on the testing set. Each shown result is the average over 10 iterations. Table II depicts the obtained recall, precision, and F_1-score for different anomaly detection solutions. The proposed TDAD achieves a precision of 87.1%, recall of 93%, and an F_1-score of 89.8%. Although TDAD underperforms Self-Supervised [10] in precision and F_1-score, it outperforms RPCA [4], TraceAnomaly [7], and Microscope [11] significantly in terms of precision, recall, and F_1-score. The inferior performance of the unsupervised approaches, RPCA [4], TraceAnomaly [7], and Microscope [11], is evident in their low F_1-score 80.2% and 61.5%, and 55.7%, respectively. These approaches rely on sequence-based trace representation, which lacks the ability to capture the causal relationships between spans. Additionally, they do not account for status codes. In contrast, Self-Supervised [10] achieves high precision and F_1-score mainly because it uses fully labeled training data.

Next, we evaluate the detection time performance of our proposed TDAD method. We observe from Table II that the proposed TDAD outperforms RPCA [4], TraceAnomaly [7], Microscope [11], and Self-Supervised [10], with significantly shorter detection times of only 0.131 ms. In comparison, RPCA [4], TraceAnomaly [7], Microscope [11], and Self-Supervised [10] had higher detection times of 2.695, 1.860, 4.791, and 0.349 ms. This is mainly due to the fact that the process of network units, represented by the nodes in an SCG in our proposed approach, benefits from parallel processing. Even though the Self-Supervised method [10] achieved a higher F_1-score, it is outperformed by our proposed TDAD in terms of detection time (see Table II). This is because the Self-Supervised method employs a softmax function for generating the final prediction, whereas TDAD uses a signum function.



e

Fig. 3. Precision, recall, and F_1-score of our proposed TDAD method vs. ratio (%) of labeled anomalous traces.

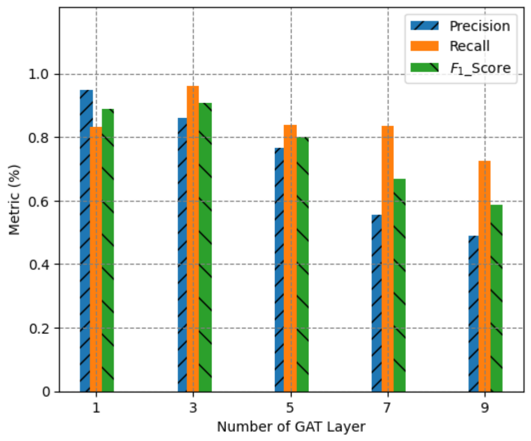


Fig. 4. Precision, recall, and F_1-score of our proposed TDAD method vs. the number of GAT layers (ratio of labeled anomalous traces=10%).

3) Impact of Configuration Parameters on the Output: We evaluate the impact of the number of labeled anomalous traces on the performance of our proposed TDAD method. Specifically, we employ TDAD to train a model using 5%, 10%, 15%, 20%, and 25% of labeled anomalous traces from the training set. Fig. 3 illustrates the performance metric of the proposed TDAD method for different percentages of labeled anomalous traces. It is apparent that TDAD's performance can be enhanced by increasing the number of labeled anomalous traces. With only 10% anomalous traces labeled in the training data, TDAD significantly outperforms the RPCA [4], TraceAnomaly [7], and Microscope [11], and only slightly underperforms the Self-Supervised [10] which uses fully labeled training data. When 20% of the anomalous traces are labeled, TDAD achieves an F_1-score of 97%, which is comparable to that of Self-Supervised with an F_1-score of 98.5%. However, a further increase in the number of labeled anomalous traces results in a decline in model performance. We attribute this decline to the fixed value of hyperparameter β during model training. As the number of labeled anomalous traces increases, the proportion of normal traces within the unlabeled data also grows. This leads to a more precise estimation of the risk associated with the unlabeled data, denoted as $\widehat{R}_{\mathrm{u}}^{-}$ in Eq. (10), without subtracting a substantial portion of $\pi_{\mathrm{p}}\widehat{R}_{\mathrm{p}}^{-}$ as previously done. However, $\pi_{\mathrm{p}}\widehat{R}_{\mathrm{p}}^{-}$ remains unchanged, leading to reduced learning from the unlabeled data and eventually causing underfitting. In order to address this issue, we suggest adjusting the value of β appropriately when the number of labeled anomalous traces increases. This adjustment ensures that the model parameters are optimally tuned, even in cases where $\widehat{R}_{\mathrm{pu}}$ is slightly below 0.

Fig. 4 illustrates the impact of varying the number of GAT layers on TDAD's performance. Notably, the performance of TDAD initially improves and subsequently declines as the number of layers increases. The highest F_1-score can be obtained when the number of GAT layers is 3. This is due to the dynamics of interaction within the SCG. When the number of GAT layers is small, the flow of information between nodes in the graph might be inadequate, which can result in incomplete learning of the graph's features. Conversely, an excessive number of GAT layers can lead to over-smoothing [22], where the model becomes excessively generalized and therefore incapable of distinguishing the learned representations among diverse graphs.

V. Conclusions and Future Directions

In this paper, we proposed TDAD, a novel approach that combines GNN and PU learning for the accurate detection of anomalous traces. Our approach demonstrates high performance even with a small number of labeled anomalous traces and a relatively large number of unlabeled traces. TDAD employs an SCG to capture the intricate hierarchical structure of traces effectively. The graph ensures the preservation of span relationships and embeds three types of information into the corresponding node representation vectors, including the semantics of the invoked service name and operation name, time-related attributes, and the status code. By utilizing the SCG, TDAD employs a graph attention network and a trace-based anomaly detection model based on PU learning. The training process involves the utilization of both a small number of labeled anomalous traces and a relatively large number of unlabeled traces. Our trace-driven evaluation on a microservice benchmark demonstrates that the proposed method outperforms not only the existing unsupervised trace-based anomaly detection methods by 11.9% in terms of F_1-score but also an existing supervised learning-based approach by 12x in terms of detection time. An interesting future work is to evaluate TDAD across diverse microservice systems as well as develop new techniques that can consider microservice application logs, resource metrics, and trace data to address the challenges associated with anomaly detection in large-scale complex microservice systems.

Acknowledgment

This work is funded by the Ericsson/ENCQOR-5G Senior Industrial Research Chair on Cloud and Edge Computing for 5G and Beyond.

References

[1] T. Davidson, E. Wall, and J. Mace, “A qualitative interview study of distributed tracing visualisation: A characterisation of challenges and opportunities,” *IEEE Transactions on Visualization and Computer Graphics*, 2023. IEEE Xplore Early Access.

[2] 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,” *Empirical Software Engineering*, vol. 27, pp. 1–28, 2022.

[3] P. Chen, Y. Qi, and D. Hou, “Causeinfer: Automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment,” *IEEE Transactions on Services Computing*, vol. 12, no. 2, pp. 214–230, 2016.

[4] M. Jin, A. Lv, Y. Zhu, Z. Wen, Y. Zhong, Z. Zhao, J. Wu, H. Li, H. He, and F. Chen, “An anomaly detection algorithm for microservice architecture based on robust principal component analysis,” *IEEE Access*, vol. 8, pp. 226397–226408, 2020.

[5] T. Wang, W. Zhang, J. Xu, and Z. Gu, “Workflow-aware automatic fault diagnosis for microservice-based applications with statistics,” *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2350–2363, 2020.

[6] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, “Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study,” *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2018.

[7] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue, *et al.*, “Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks,” in *Proc. IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pp. 48–58, 2020.

[8] 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 *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 19–33, 2019.

[9] S. Nedelkoski, J. Cardoso, and O. Kao, “Anomaly detection and classification using distributed tracing and deep learning,” in *Proc. IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 241–250, 2019.

[10] J. Bogatinovski, S. Nedelkoski, J. Cardoso, and O. Kao, “Self-supervised anomaly detection from distributed traces,” in *Proc. IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, pp. 342–347, 2020.

[11] J. Lin, P. Chen, and Z. Zheng, “Microscope: Pinpoint performance issues with causal graphs in micro-service environments,” in *Proc. Springer International Conference on Service-Oriented Computing (ICSOC)*, pp. 3–20, 2018.

[12] L. Meng, F. Ji, Y. Sun, and T. Wang, “Detecting anomalies in microservices with execution trace comparison,” *Future Generation Computer Systems*, vol. 116, pp. 291–301, 2021.

[13] H. Chen, P. Chen, and G. Yu, “A framework of virtual war room and matrix sketch-based streaming anomaly detection for microservice systems,” *IEEE Access*, vol. 8, pp. 43413–43426, 2020.

[14] M. Schuster and K. Nakajima, “Japanese and korean voice search,” in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5149–5152, 2012.

[15] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: pre-training of deep bidirectional transformers for language understanding,” *CoRR*, vol. abs/1810.04805, 2018.

[16] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, *et al.*, “Transformers: State-of-the-art natural language processing,” in *Proc. International Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, 2020.

[17] R. Fielding and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Semantics and content,” Technical Report, RFC 7231, 2014.

[18] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio, *et al.*, “Graph attention networks,” in *Proc. International Conference on Learning Representations (ICLR)*, pp. 10–48550, 2017.

[19] R. Kiryo, G. Niu, M. C. Du Plessis, and M. Sugiyama, “Positive-unlabeled learning with non-negative risk estimator,” *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[20] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *Proc. International Conference on Learning Representations (ICLR) Engineering*, 2015.

[21] A. Avritzer, D. Menasché, V. Rufino, B. Russo, A. Janes, V. Ferme, A. van Hoorn, and H. Schulz, “PPTAM: production and performance testing based application monitoring,” in *Proc. ACM/SPEC International Conference on Performance Engineering*, pp. 39–40, 2019.

[22] Q. Li, Z. Han, and X.-M. Wu, “Deeper insights into graph convolutional networks for semi-supervised learning,” in *Proc. AAAI Conference on Artificial Intelligence*, vol. 32, 2018.