DeepTraLog: Trace-Log Combined Microservice Anomaly Detection through Graph-based Deep Learning

Chenxi Zhang*
Fudan University
China

Xin Peng*†
Fudan University
China

Chaofeng Sha*
Fudan University
China

Ke Zhang*
Fudan University
China

Zhenqing Fu*
Fudan University
China

Xiya Wu*
Fudan University
China

Qingwei Lin
Microsoft Research
China

Dongmei Zhang
Microsoft Research
China

ABSTRACT

A microservice system in industry is usually a large-scale distributed system consisting of dozens to thousands of services running in different machines. An anomaly of the system often can be reflected in traces and logs, which record inter-service interactions and intra-service behaviors respectively. Existing trace anomaly detection approaches treat a trace as a sequence of service invocations. They ignore the complex structure of a trace brought by its invocation hierarchy and parallel/asynchronous invocations. On the other hand, existing log anomaly detection approaches treat a log as a sequence of events and cannot handle microservice logs that are distributed in a large number of services with complex interactions. In this paper, we propose DeepTraLog, a deep learning based microservice anomaly detection approach. DeepTraLog uses a unified graph representation to describe the complex structure of a trace together with log events embedded in the structure. Based on the graph representation, DeepTraLog trains a GGNNs based deep SVDD model by combing traces and logs and detects anomalies in new traces and the corresponding logs. Evaluation on a microservice benchmark shows that DeepTraLog achieves a high precision (0.93) and recall (0.97), outperforming state-of-the-art trace/log anomaly detection approaches with an average increase of 0.37 in F1-score. It also validates the efficiency of DeepTraLog, the contribution of the unified graph representation, and the impact of the configurations of some key parameters.

*C. Zhang, X. Peng, C. Sha, K. Zhang, Z. Fu and X. Wu are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China and the Shanghai Collaborative Innovation Center of Intelligent Visual Computing, China
†X. Peng is the corresponding author (pengxin@fudan.edu.cn).

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510180

CCS CONCEPTS

• **Software and its engineering**; • **Computer systems organization → Reliability**; **Maintainability and maintenance**; **Cloud computing**;

KEYWORDS

Microservice, Anomaly Detection, Log Analysis, Tracing, Graph Neural Network, Deep Learning

ACM Reference Format:
Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. DeepTraLog: Trace-Log Combined Microservice Anomaly Detection through Graph-based Deep Learning. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510180

1 INTRODUCTION

Microservice architecture is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms [15]. A microservice system in industry is usually a large-scale distributed system having dozens to thousands of services running in different machines. Running in a highly uncertain and dynamic environment, a microservice system often fails due to various infrastructure problems or application faults such as hardware failures, improper configurations, implementation faults, and incorrect coordination in service interactions [42, 43]. To allow engineers to timely react to potential failures, it is desirable that the anomalies of a microservice system can be automatically detected at runtime.

Powered by specifications like OpenTracing [27] and related infrastructures like SkyWalking [33], distributed tracing [32] has been widely adopted in industrial microservice systems. Each produced trace describes the execution process (i.e., invocation chain) of a request through service instances and each operation (i.e., service invocation) in it is called a span. At the same time, logging has also been widely used by the developers to record the behaviors of each service. A log records significant messages at various critical points for the purpose of debugging and root cause analysis [7]. A trace can include several to hundreds of service invocations (i.e., spans)

and during each invocation a series of log messages are produced by the invoked service instance. Industrial distributed tracing systems can link the log messages of the same trace by injecting the trace ID and span IDs into the log messages produced by different service instances.

Recent researches [20, 26] use deep learning based trace analysis methods to detect runtime anomalies of microservice systems. These approaches treat a trace as a sequence of service invocations. However, a trace can have a complex structure formed by the hierarchy of service invocations and parallel/asynchronous invocations. Existing trace anomaly detection approaches ignore the complex structures of traces. Moreover, they do not consider the log messages which describe the behaviors of individual service instances involved in a trace. Therefore, these approaches cannot well capture microservice anomalies.

On the other hand, logs have been widely used in anomaly detection for distributed systems. Existing log anomaly detection approaches [7, 24, 40] learn log patterns from normal execution and detect anomalies when log patterns deviate from the trained model. These approaches treat a log as a sequence of log events, which are the abstraction of a group of similar log messages [12]. For a distributed system, a log is produced for each request by sorting log messages from different nodes involved in the request by timestamp. For a microservice system, however, a request corresponds to an invocation chain that may involve many service instances and complex invocations among them. If we produce a log for a request in a similar way, it cannot well capture the complex structure of its invocation chain.

In this paper, we propose DeepTraLog, a deep learning based microservice anomaly detection approach. DeepTraLog uses a unified graph representation, which is called trace event graph (TEG), to describe the complex structure of a trace together with log events embedded in the structure. It takes traces and logs as input and trains a graph-based deep learning model for trace anomaly detection. First, it parses the input traces and logs and extracts span relationships and log events from them respectively. Second, it generates vector representations for span events and log events and at the same time constructs a TEG for each trace. Third, it trains a gated graph neural networks (GGNNs) based deep SVDD (Support Vector Data Description) model, which learns a latent representation for each TEG and a minimized data-enclosing hypersphere. When used for anomaly detection, DeepTraLog analyzes a trace and the related logs in a similar way and uses the trained model to generate a latent representation for the trace. It then determines whether the trace is anomalous based on its anomaly score, i.e., the shortest distance from the latent representation of the trace to the hypersphere. Note that DeepTraLog does not rely on trace labelling and just requires that the majority of traces in the training set are produced in normal execution of the system. Moreover, it is able to capture different types of anomalies.

To evaluate the effectiveness and efficiency of DeepTraLog we conduct a series of experimental studies on a microservice benchmark system. The results show that DeepTraLog outperforms existing trace- and log-based anomaly detection approaches by 64.94% and 101.59% on average in terms of precision and recall respectively. The unified graph representation significantly contributes to the improvement of DeepTraLog, making it outperform the variant of DeepTraLog using sequence representation by 7.64% and 26.03% on average in terms of precision and recall respectively. DeepTraLog is efficient in model training and testing and its response time in anomaly detection increases linearly with the size of the trace.

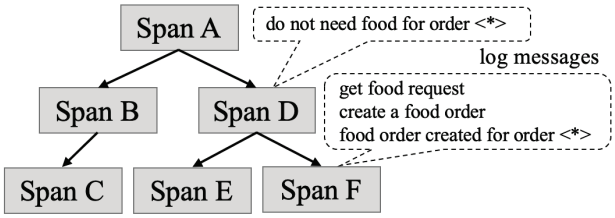


Figure 1: Trace, Span, and Log

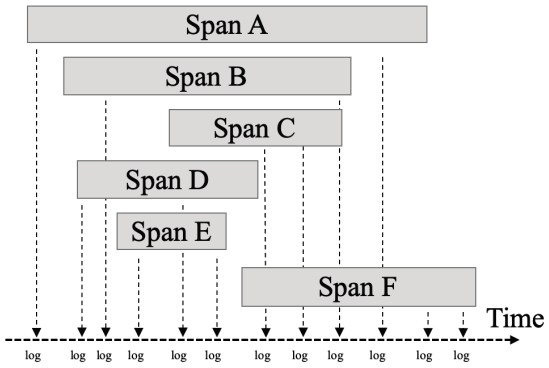


Figure 2: Timeline of Spans in Figure 1

In summary, this paper makes the following contributions:

- a unified graph representation of traces and logs that facilitates the combined analysis of them;
- a GGNNs based deep SVDD model for microservice anomaly detection;
- a series of experimental studies validating the effectiveness and efficiency of DeepTraLog together with the contribution of the unified graph representation and the impact of the configurations of some key parameters.

Significance. Our work provides a new and effective way for combining traces and logs for microservice anomaly detection which outperforms existing log/trace anomaly detection approaches. It defines a unified graph-based representation for inter-service interactions and intra-service behaviors. The representation can facilitate a variety of different microservice analysis tasks such as anomaly detection, root cause analysis, and architecture comprehension.

2 BACKGROUND AND MOTIVATION

In this section, we first introduce the background about traces and logs, then motivate our work with an example.

2.1 Background

As a kind of large-scale distributed systems, microservice systems widely use distributed tracing [27, 32] to profile and monitor their executions. A trace is the description of the execution process of

a request as it moves through a distributed system [27]. For a microservice system, a trace describes the execution process of a request through service instances, i.e., a service invocation chain. The service invocations in a trace are initially recorded by individual service instances and then collected and restored to a trace. As shown in Figure 1, a trace consists of a set of spans in a tree structure and each span corresponds to a service invocation. Each trace has a unique trace ID and each span has a unique span ID. A span records the invoker and the invoked service instance and has an operation name indicating the operation to be performed. For example, the operation name of a span of REST invocation can be *POST /api/v1/foodservice/orders*. Each span (except the root span) has a parent span which initiates the current span. For example, in Figure 1 Span A is a synchronous invocation and during its execution it initiates two synchronous invocations corresponding to Span B and Span D respectively. Therefore, Span A is the parent of Span B and Span D. Distributed tracing has been an important part of microservice infrastructures and supported by open-source solutions (e.g., Zipkin [35], Jeager [13], and SkyWalking [33]) and cloud service providers (e.g., Amazon's X-Ray [3] and Alibaba Cloud's ARMS [1]).

Asynchronous invocations and parallel invocations are widely used in microservice systems for better performance and availability. A trace can include several to hundreds of spans (i.e., service invocations). Therefore, synchronous invocations are often considered harmful due to the multiplicative effect of downtime [15]. Asynchronous invocations are usually implemented by message-based communication and are thought to be a way for achieving high-availability systems as the invoker does not block while waiting. Parallel invocations mean to invoke multiple services at the same time to reduce the overall response time. They are usually implemented by making service invocations in multiple threads.

As shown in Figure 1, log messages are produced during each service invocation. These log messages are produced by the logging statements written by service developers. They record the internal states and behaviors of the invoked service instance. A log message is an unstructured sentence which contains a constant part (log event) and several variable parts (log parameters). For example, a log message "[assignSeat] Requested seat type is T1 and number is 2" contains a log event "[assignSeat] Requested seat type is <*> and number is <*>" and two log parameters *SeatType* and *SeatNumber*. Log event extraction has been a standard step in log parsing. After log parsing, a log is converted into a sequence of log events.

Traces and logs can be combined for analyzing the runtime behaviors of microservice systems. A trace describes the service interactions for a request, while the logs record the internal states and behaviors in individual service instances. As a service instance may serve for multiple requests at the same time, its log file interleaves log messages for different requests. To support the combined analysis, some distributed tracing systems (e.g., SkyWalking [33], Alibaba ARMS [1]) inject trace IDs and span IDs into log messages and thus the log messages of a service instance can be associated with different requests. For example, SkyWalking uses the Mapped Diagnostic Context mechanism of Java logging frameworks (e.g., Log4j [2], Logback [21]) to inject trace IDs and span IDs to log messages.

2.2 Motivation

Figure 2 shows the timeline of the spans in Figure 1. Span B and Span D are two parallel invocations generated by Span A, so they have some overlap in the timeline. Span F is an asynchronous invocation generated by Span D, so it ends after Span D.

Existing trace anomaly detection approaches [20, 26] treat a trace as a sequence of service invocations, while existing log anomaly detection approaches [7, 24, 40] treat a log as a sequence of log events. These approaches cannot well support the anomaly detection of microservice systems due to the following two reasons.

First, the logs of different service instances need to be combined for anomaly detection. Existing trace anomaly detection approaches do not consider logs, thus can only detect anomalies that are reflected in trace structures. Figure 1 shows an example of log-level anomalies. The log messages in Span D show that the train ticket order does not need food, but those in Span F show that a food order is created for this train ticket order. This anomaly can only be detected by combing the log messages of Span D and F.

Second, a trace may have a complex structure involving invocation hierarchy and parallel/asynchronous invocations. A trace has a tree structure. If all the spans are synchronous invocations, the trace can be represented by a sequence of service invocations ordered by their start time. Even so, sequence-based representation cannot reflect the causal relationships between parent and children spans and temporal relationships between log events of the same spans. For example, two adjacent log events in Span A may be far in the sequence-based representation, as the log events of the descendant spans of Span A are inserted into them. Moreover, a trace may include parallel or asynchronous invocations. The trace shown in Figure 1 and Figure 2 includes two parallel invocations (Span B and Span D) and an asynchronous invocation (Span F). Therefore, the log events in Span B and Span D (and their descendant spans) can interleave in any order; Similarly, the log events in Span F and a part log events in Span D can interleave in any order. If we combine the log events of different spans into a sequence (e.g., by start time), the characteristics of parallel or asynchronous invocations will be lost.

Based on the analysis, we can see that the log events of different spans of a trace need to be combined in a way that the structure of the trace can be kept. Therefore, we propose a unified graph representation that can describe the structure of a trace together with the log events embedded in the structure to facilitate anomaly detection.

3 APPROACH

The objective of DeepTraLog is to automatically and accurately detect anomalous traces of microservice systems. It takes traces and logs as input and trains a graph-based deep learning model. When used for anomaly detection, it analyzes a trace and the associated logs in a similar way and uses the model to generate a representation for the trace to calculate its anomaly score.

An overview of DeepTraLog is presented in Figure 3, which includes six steps. **Log Parsing** parses the input logs and extracts log events from the log messages. **Trace Parsing** parses the input traces and converts their spans into span events, which will be analyzed

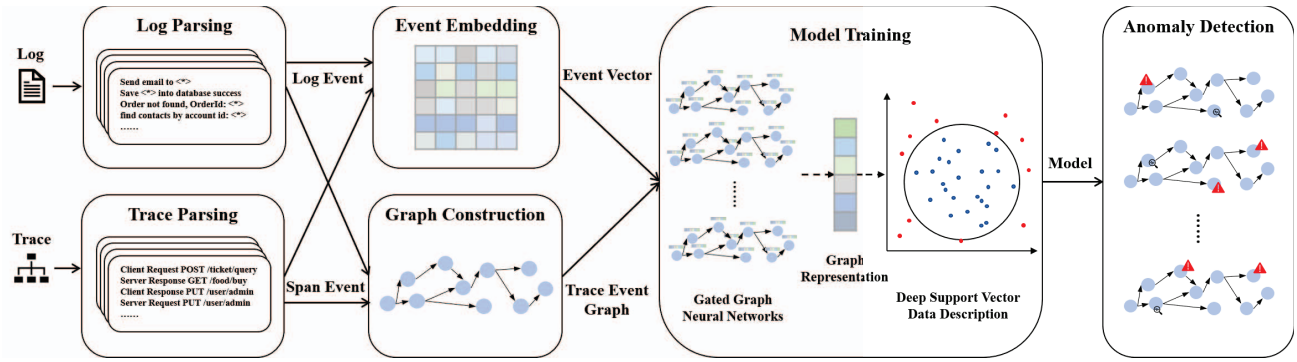


Figure 3: DeepTraLog Overview

together with log events. **Event Embedding** generates vector representations for span events and log events. **Graph Construction** constructs a trace event graph (TEG) for each trace to represent various relationships between the span/log events of the trace. **Model Training** trains a gated graph neural networks (GGNNs) based deep SVDD (Support Vector Data Description) model, which learns a latent representation for each TEG and a minimized hypersphere that encloses the representations of the TEGs. **Anomaly Detection** determines whether a trace is anomalous based on its anomaly score (i.e., the shortest distance from the latent representation of its TEG to the hypersphere).

3.1 Log Parsing

Following existing researches on log anomaly detection [24, 40], we adopt a state-of-the-art log parsing approach Drain [11]. It can parse logs in a streaming and timely manner with high parsing accuracy and efficiency. To support the combined analysis with traces, we extract and record the trace ID and span ID of each log message before log parsing. After log parsing, a trace ID and a span ID are attached with each of the extracted log events for further analysis.

3.2 Trace Parsing

The log events of a service instance constitute an event sequence. To combine traces and logs for anomaly detection, we need to convert the spans of a trace into span events. Span events are a special kind of events that represent the request and response of service invocations. When analyzed together with log events, span events can indicate the starts and ends of service invocations.

For each span of a trace, we convert it into multiple span events of related spans according to its type (*Client/Server* or *Producer/Consumer*). A *Client/Server* span represents a synchronous invocation, while a *Producer/Consumer* span represents an asynchronous invocation. For a *Client/Server* span we generate a request event and a response event for the current span (server) and its parent span (client) respectively. The request event and the response event of the client (server) represents the sending (receiving) of the service invocation and the receiving (sending) of the invocation response respectively. For a *Producer/Consumer* span we generate a consumer event for the current span (consumer) and a producer event for the parent span of the current span (producer). The producer event and the consumer event represent the sending and receiving of the message respectively.

The content of a span event includes two parts, an event type and an operation name. For example, for a *Client/Server* span with the operation name *POST /api/v1/foodservice/orders*, we generate two span events "Server Request POST /api/v1/foodservice/orders" and "Server Response POST /api/v1/foodservice/orders" for the current span and two span events "Client Request POST /api/v1/foodservice/orders" and "Client Response POST /api/v1/foodservice/orders" for the parent span of the current span; for a *Producer/Consumer* span with the operation name *RabbitMQ/Topic/Queue/email/sendEmail*, we generate a span event "Consumer RabbitMQ/Topic/Queue/email/sendEmail" for the current span and a span event "Producer RabbitMQ/Topic/Queue/email/sendEmail" for the parent span of the current span. Each span event has a timestamp obtained from the span record. For example, the timestamp of a client request event is the time when the client service instance sends the service invocation.

3.3 Event Embedding

Log event embedding is widely used in log anomaly detection. It generates a vector representation for each log event. The vector representation can identify semantically similar log events and also distinguish different log events [40]. In DeepTraLog, span events are analyzed together with log events for anomaly detection. Therefore, our event embedding generates vector representations for both log events and span events. Each log event or span event is a sequence of English words, thus can be treated as a sentence. Following the common practice of log event embedding, DeepTraLog implements event embedding in three steps.

Step 1. Preprocessing. Log events and span events contain non-character tokens (e.g., separators such as "/" and ",", IP addresses), stop words (e.g., "a", "the", "is"), and compound words (e.g., "verifycode", "tripid"). Following previous works [24, 40], we preprocess log events and span events by removing non-verbal symbols and stop words, and splitting compound words into individual words. For example, a span event "Client Request POST /api/v1/foodservice/orders" is preprocessed into "client request post api v1 food service orders".

Step 2. Word Embedding. We use the widely used pre-trained GloVe model [29] to generate a vector representation for each word

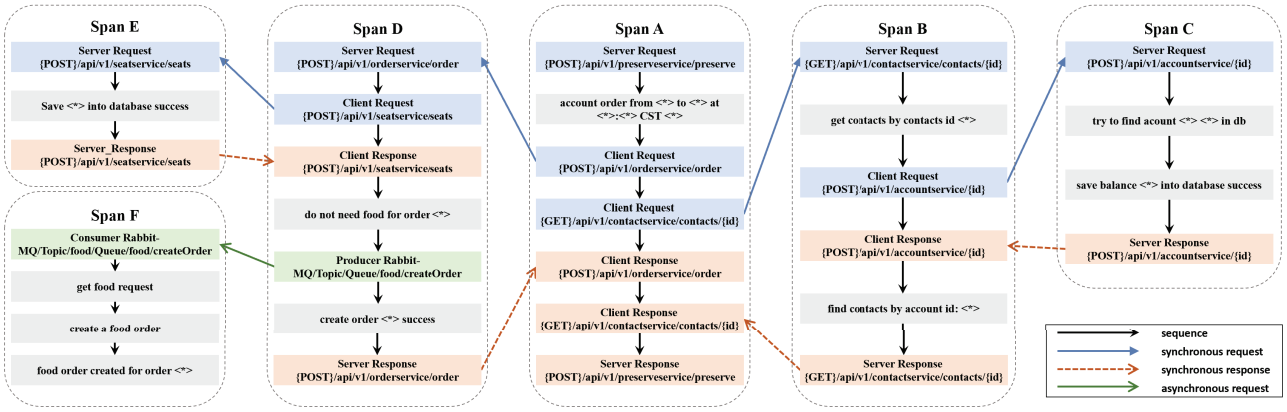


Figure 4: An Example of Trace Event Graph (TEG)

in log events and span events. Particularly, we use the GloVe 300-dimensional word vectors trained on Wikipedia and Gigaword 5 data [9]. Thus for each word we can obtain a 300-dimensional vector as its representation.

Step 3. Sentence Embedding. Sentence embedding generates a vector representation for each log event and span event based on word embedding. The words in these events are not equally important. Some words (e.g., "api", "get") are more common, thus are less important than others (e.g., "food", "submit") in sentence embedding. Therefore, we follow previous works [40] and use TF-IDF [31] to measure the weight of each word in sentence embedding. TF (Term Frequency) of a word w in an event e, which measures its importance in e, is calculated by $TF_{w,e} = \frac{L_{w,e}}{L_e}$, where $L_{w,e}$ is the occurrences of w in e and L_e is the number of words in e. IDF (Inverse Document Frequency) of a word w, which measures its frequency in all events, is calculated by $IDF_w = \log \frac{L}{L_w}$, where L is the total number of events and L_w is the number of events containing w. Then the weight of a word w in an event e can be measured by its TF-IDF score as $W_{w,e} = TF_{w,e} \times IDF_w$. Thus, the vector representation of an event e can be calculated as the weighted sum of the vector representations of all its words as following, where N_e is the number of different words in e and V_w is the vector representation of a word w.

$$V_e = \frac{1}{N_e} \sum_{w=1}^{N_e} W_{w,e} \cdot V_w \tag{1}$$

Note that log events and span events have quite different characteristics, so we calculate the TF-IDF scores and word weights for log events and span events separately.

3.4 Graph Construction

A trace event graph (TEG) consists of the log events and span events of a trace and their relationships. A relationship in a TEG can be one of the following four types.

- **Sequence**: A sequence relationship represents the predecessor/successor relationship between two sequential span/log events of the same span.
- **Synchronous Request**: A synchronous request relationship represents a synchronous request from a parent span to its child span.
- **Synchronous Response**: A synchronous response relationship represents the response of a synchronous request from a span to its parent span.
- **Asynchronous Request**: An asynchronous request relationship represents an asynchronous request from a parent span to its child span.

Given a trace, we construct a TEG in three steps.

Step 1: Connection of Log Events. For each span of the trace, obtain all the log events that belong to the span. Then order the log events by their timestamps and add a sequence relationship from each log event to the one next to it.

Step 2: Insertion of Span Events. For each span of the trace, obtain all the span events that belong to the span. Then for each obtained span event, insert it into the log event sequence of the current span based on its timestamp and add a sequence relationship between it and its predecessor/successor event.

Step 3: Connection of Spans. For each span connect it with its parent span in the following way. If the span is a *Client/Server* span, add a synchronous request relationship from the corresponding client request event of its parent span to the corresponding server request event of the current span, and a synchronous response relationship from the corresponding server response event of the current span to the corresponding client response event of its parent span. If the span is a *Producer/Consumer*) span, add an asynchronous request relationship from the corresponding producer event of its parent span to the corresponding consumer event of the current span.

Figure 4 shows an example of TEG, which corresponds to the trace shown in Figure 1 and Figure 2. In the graph, colored rectangles and uncolored rectangles represent span events and log events respectively. Span events of different colors are of different types, including client/server request, client/server response, and producer/consumer. Note that the dotted rounded rectangles which represent spans in Figure 4 are not the nodes of the TEG but are just used for illustration. The spans are implicitly represented in the TEG by the event sequences starting from server request events.

Arrows of different colors represent different types of relationships. The graph can well describe the structure of the trace and various relationships in it. The log events and span events in the same span form a sequence by their timestamps. Therefore, the temporal relationships between inter-service interactions (span events) and intra-service behaviors (log events) can be described and considered in anomaly detection. The events of different spans are separated and only connected via request/response relationships between span events. The structure of the trace is embodied in these request/response relationships: a synchronous invocation is reflected by a pair of request and response relationships; an asynchronous invocation is reflected by a request relationship without the corresponding response relationship. For example, it can be seen that Span B and Span D are parallel synchronous invocations as their request/response ranges in Span A overlap; Span F is an asynchronous invocation initiated by Span D as there is no response from Span F to Span D.

3.5 Model Training

We frame the task of trace-log combined microservice anomaly detection as an one-class classification problem. One-class classification problem aims to learn a model that can accurately describe the train data which is considered belong to the same class. For anomaly detection task, most of the training samples are normal ones which can be regarded as the same class and the others can identified as outliers (e.g., anomalous traces in our work).

DeepTraLog uses deep SVDD [30] to train a one-class classification model for detecting anomalous traces. Deep SVDD learns useful feature representations of the training data together with a minimized hypersphere that encloses the latent representations of the data. Thus the data that is distant from the center of the hypersphere can be considered anomalous. Existing deep learning based anomaly detection approaches [20, 26] usually use deep autoencoder to detect anomalies based on the reconstruction error of the data. Thus the anomaly detection relies on an empirically determined threshold of the reconstruction error, which is often challenging. In contrast, deep SVDD jointly learns the latent representations of the data and a classification boundary without a threshold. Standard deep SVDD uses multi-layer perception (MLP) or convolutional neural networks (CNN) to learns the latent representations of the data. However, in our work a trace is represented by a graph (TEG) which cannot be well handled by MLP or CNN. Therefore, we use gated graph neural networks (GGNNs) to learn the trace representations and jointly train the GGNNs with deep SVDD. GGNNs is implemented based on the neural message passing mechanism and can work well with a variety types of graphs, such as directed graph, bipartite graph, and undirected graph.

In DeepTraLog, a trace is represented by a TEG, which is a directed attributed graph $g = \{V, A, X\}$ where: V is a set of nodes (i.e., events); A is adjacency matrix of the graph; and $X \in \mathbb{R}^{|V| \times d}$ is the node attribute matrix, where each row x_v of X is the attribute (i.e., event vector) of a node $v \in V$, d is the dimension of the event vector. GGNNs represents the nodes in a graph as units of a neural network and the units are linked to each other according to the adjacency matrix of the graph. GGNNs passes the node attributes as the messages between the units in every iteration and uses GRU [4] to determine which messages to remember or forget during the message passing. The final representation of a node is determined by a combination of its own state and the state of neighbouring nodes. The representation of a node after the t-th iteration is defined by the following equations:

$$h_v^{(0)} = x_v \tag{2}$$

$$m_v^{(t)} = A_v^{\mathrm{T}}[h_1^{(t-1)\mathrm{T}} \ldots h_{|V|}^{(t-1)\mathrm{T}}]^{\mathrm{T}} + b \tag{3}$$

$$h_v^{(t)} = \mathrm{GRU}(m_v^{(t)}, h_v^{(t-1)}) \tag{4}$$

where $h_v^{(i)}$ is the representation of a node v after the i-th iteration; x_v is the initial event vector of node v; $A_v = [a_{v,:}^{\mathrm{T}}, a_{:,v}]$ is the row and column in the adjacency matrix A, which represent the incoming edges and outgoing edges of v; GRU is the GRU function. Finally after T iterations each node in a TEG can have a vector as the latent representation. GGNNs calculates the vector representation of the TEG based on the node vector representations using a soft-attention mechanism. The soft-attention mechanism takes the vector representations of the nodes as input and calculates the weight (attention score) of each node through an attention function to give higher weights to the nodes that contribute more to the graph classification. The graph representation of TEG g is calculated by the following equation:

$$h_g = \tanh\left(\sum_{v \in V} \phi\left(f_i(h_v^{(T)}, x_v)\right) \odot \tanh\left(f_j(h_v^{(T)}, x_v)\right)\right) \tag{5}$$

where h_g is the vector representation of a TEG g; $\phi\left(f_i(h_v(t), x_v)\right)$ is the soft-attention mechanism; f_i and f_j are neural networks; T is the number of layers of GGNNs; $\odot$ is element-wise multiplication. Readers can refer to [17] for more details about GGNNs.

DeepTraLog trains the GGNNs using the following loss function, which expresses the objective of learning a minimized hypersphere to enclose the vector representations of TEGs:

$$Loss = R^2 + \frac{1}{\mu N_g} \sum_{g=1}^{N_g} \max\{0, ||h_g - c||^2 - R^2\} + \frac{\lambda}{2} \sum_{l=1}^{N_l} ||\theta^{(l)}||_F^2 \tag{6}$$

where c is the center of the hypersphere; R is the radius of the hypersphere; $||h_g - c||^2$ is the distance from the latent representation of a TEG g to c; N_g is the number of TEGs; N_l is the number of network layers in GGNNs; the hyperparameter μ controls the trade-off between the hypersphere volume and the violations of the boundary, which allows some training data mapped outside the hypersphere (i.e., allowing some anomaly data in the training set); $\frac{\lambda}{2} \sum_{l=1}^{N_l} ||\theta^{(l)}||_F^2$ is a weight decay regularizer on the GGNNs parameters θ with hyperparameter λ.

In the training phase, we jointly optimize the GGNNs parameters θ and the hypersphere radius R in Equation 6. We use Adam [14] to optimize the GGNNs parameters θ in each epoch. As the radius R is not an inner parameter of GGNNs, we optimize it using a different method as follows. We optimize θ with a fixed R in the first few epochs and after every k epochs we calculate an optimized value for R by linear search. Each time when R is updated its value is

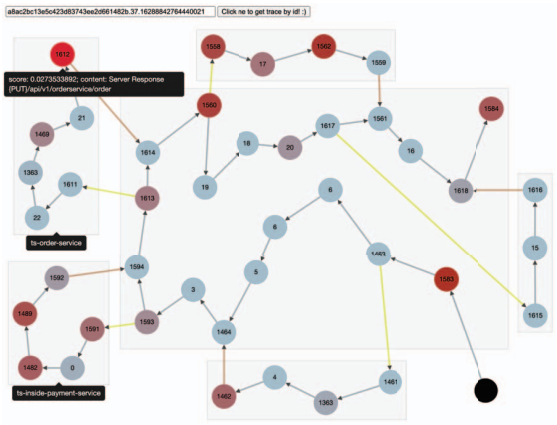


Figure 5: Visualization of An Anomalous TEG

calculated as the $(1-\mu)$ percentile of the distances of all the TEGs in the current epoch. The hypersphere center c is set to the mean of the vector representations of all the TEGs after an initial forward pass.

3.6 Anomaly Detection

DeepTraLog trains a GGNNs based deep SVDD model for anomaly detection. Given a new trace and the corresponding log for anomaly detection, DeepTraLog follows the same process as the training phase (i.e., log parsing, trace parsing, event embedding, and graph construction) to produce a TEG together with its event embedding for the trace. Then DeepTraLog feeds the TEG into the trained model to generate a latent representation of the TEG and uses the latent representation to calculate an anomaly score. The anomaly score is defined as the shortest distance from the latent representation of the TEG to the learned hypersphere, which is calculated by the following equation:

$$ans(h_g) = ||h_g - c||^2 - \hat{R}^2 \tag{7}$$

where h_g is the vector representation of a TEG g; c is the center of the learned hypersphere; $\hat{R}$ is the final radius of the learned hypersphere.

For a TEG g of a trace, if its anomaly score (i.e., $ans(h_g)$) is greater than 0 it is treated as anomalous.

To help the users to understand the results of anomaly detection, DeepTraLog provides a visualization of the TEGs of anomalous traces. Figure 5 presents an example of the visualization of an anomalous TEG. Specifically, DeepTraLog highlights the nodes in the TEG that have high attention scores calculated using Equation 5, which helps the users to quickly locate the anomalous parts of the trace.

4 EVALUATION

We implement DeepTraLog using Python 3.8.8, PyTorch 1.8.0, and PyTorch Geometric 1.7.2 (for GGNNs learning). To evaluate it we conduct a series of experimental studies to answer the following research questions:

Table 1: Fault Types in the TrainTicket Dataset

Fault Type	Fault Cases	Example
Asynchronous Interaction	F1, F2, F13	F1: asynchronous message delivery without sequence control
Multi-Instance	F8, F11, F12	F12: service states not synchronized among different instances of the service
Configuration	F3, F4, F5, F7	F4: improper configurations of SSL
Monolithic	F6, F9, F10, F14	F14: wrong calculating process of train ticket price

- **RQ1**: How effective is DeepTraLog in microservice anomaly detection compared with baseline approaches? How much does the unified graph representation contribute to the effectiveness of DeepTraLog?
- **RQ2**: How efficient is DeepTraLog in model training and anomaly detection compared with baseline approaches? How does DeepTraLog scale with the size of trace in online prediction?
- **RQ3**: How do different configurations of the GGNNs-based deep SVDD model impact the effectiveness of DeepTraLog?

4.1 Experimental Design

4.1.1 Benchmark System and Dataset. Our studies are conducted on the latest release V0.2.0 of TrainTicket[1] [42, 44]. It is a medium-scale open-source microservice system for train ticket booking and has been widely used in researches on microservice architecture, infrastructure, and AIOps (Artificial Intelligence for IT Operation) [20, 41, 43]. It has 45 services written by different languages (e.g., Java, JavaScript, Python) and communicating with synchronous REST invocations and asynchronous messaging.

TrainTicket replicates a variety of different types of fault cases from industrial microservice systems and provides the fault cases in different fault branches. The latest release of TrainTicket provides 14 compatible fault cases of different types as shown in Table 1. Each fault case may include multiple fault instances in different services. The fault types include [43]:

- **Asynchronous Interaction** - faults caused by missing or improper coordination of asynchronous service invocations;
- **Multi-Instance** - faults related to the existence of multiple instances of the same service at runtime;
- **Configuration** - faults caused by improper or inconsistent configurations of services and/or environments (e.g., containers and virtual machines);
- **Monolithic** - faults caused by internal implementations of individual services, which can cause failures even when the application is deployed in a monolithic mode.

Thus we can have a normal version of TrainTicket and 14 faulty versions each corresponding to the branch of a fault case. We deploy different versions of TrainTicket on a Kubernetes cluster with 8 virtual machines, each of which has a 16-core 3.0GHz CPU and 32GB RAM. We use Python to implement an execution controller that can execute automated test cases. For each version we use the execution

[1]TrainTicket Project: https://github.com/FudanSELab/train-ticket

controller to simulate user requests by executing automated test cases. We use Apache SkyWalking [33] as the distributed tracing framework to collect traces and logs and use ElasticSearch [8] to store the collected traces and logs.

The resulted dataset includes 132,485 traces and 7,705,050 log messages. Among the traces 23,334 (17.6%) are anomalous ones caused by 73 faults of the 14 fault cases which are located in different services. This dataset is used throughout the whole experimental studies. It is available in our replication package [5].

4.1.2 Baselines. We use the following four state-of-the-art log-based or trace-based anomaly detection approaches as the baselines.

- **TraceAnomaly** [20] is a trace anomaly detection approach that only considers service-level traces (operations not considered). It adopts posterior flow based variational auto-encoders (VAE) to detect anomalous traces.
- **MultimodalTrace** [26] is a trace anomaly detection approach that treats a trace as a span sequence and a response time sequence. It adopts a multi-modal LSTM (Long Short-Term Memory) model to learn the sequential patterns of normal traces.
- **DeepLog** [7] is a log anomaly detection approach that treats a log as an event sequence. It adopts an LSTM model to predict the next log event in the sequence and identify possible anomalies.
- **LogAnomaly** [24] is a log anomaly detection approach that treats a log as an event sequence and considers the counts of different log events as an additional feature. It adopts an LSTM model to learn sequential and quantitative patterns. Similar to DeepLog, it detects anomalies by predicting the next log event.

As TraceAnomaly and MultimodalTrace only consider traces, we feed them with only the traces when using them. LogAnomaly and DeepLog only consider logs and treat them as event sequences. Therefore, we combine all the events (including span events and log events) of different spans of a trace into a single event sequence ordered by timestamp and provide the event sequences of all the traces as the input for these two approaches. TraceAnomaly and DeepLog provide open-source implementations [22, 34] and we use them directly. The other two approaches have no publicly available implementations, so we develop our own implementations based on their papers.

To evaluate the contribution of the unified graph representation of traces and logs (i.e., TEG), we derive a variant of DeepTraLog called **GRU-based Deep SVDD** which represents all the events of a trace as an event sequence ordered by timestamp and uses GRU instead of GGNNs.

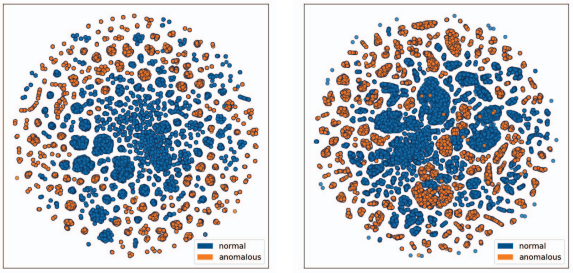
For all these baseline approaches we experimentally choose the best parameters and use their optimal results for comparison.

4.1.3 Metrics. We use the precision, recall, and F1-score to measure the effectiveness of anomaly detection based on TP (True Positive), FP (False Positive), and FN (False Negative).

- **Precision**: the percentage of anomalous traces out of all traces detected as anomalies, represented as $precision = \frac{TP}{TP+FP}$.

Table 2: Effectiveness of Different Approaches

Approach	Precision	Recall	F1-Score
TraceAnomaly	0.742	0.205	0.321
MultimodalTrace	0.591	0.776	0.671
DeepLog	0.608	0.948	0.741
LogAnomaly	0.415	0.977	0.582
GRU-based Deep SVDD	0.864	0.776	0.818
DeepTraLog	**0.930**	**0.978**	**0.954**



(a) DeepTraLog (b) GRU-based Deep SVDD

Figure 6: Distribution of Latent Representations of Traces

- **Recall**: the percentage of all anomalous traces that are detected as anomalies, represented as $recall = \frac{TP}{TP+FN}$.
- **F1-Score**: the harmonic mean of precision and recall, represented as $F_1 = \frac{2 \cdot Precision \cdot Recall}{Precision+Recall}$.

4.1.4 Settings. All the experiments are conducted on a Linux server with Intel Core i9-10900X 3.70GHz CPU, 128 GB RAM, RTX 3090 with 24GB GPU memory and running Ubuntu 18.04.5. The setting of DeepTraLog are the following: the embedding size of each event set to 300, the hidden layer of GGNNs set to 3, the hidden size of each hidden layer set to 300, the λ and μ in Equation 6 set to 0.001 and 0.05 respectively, and the batch size set to 32. Following the practice in [30], we employ a simple two-phase learning rate schedule with an initial learning rate 0.0001 in the first 60 epochs and subsequently 0.00001 in the last 40 epochs. For each approach we leverage 60% of the normal traces as the training set, 10% of the normal traces as the validation set, and the rest of the traces (include the rest 30% of the normal traces and all the anomalous traces) as the test set.

4.2 RQ1: Effectiveness

Table 2 shows the effectiveness evaluation results of different approaches. DeepTraLog outperforms all the baseline approaches and achieves a high precision (0.930), recall (0.978), and F1-score (0.954).

The two trace-based approaches (i.e., TraceAnomaly and MultimodalTrace) achieve low precision and recall. These two approaches do not consider logs, thus cannot detect anomalies in log events. As they use sequence-based trace representation and have a special focus on response time, they can only detect anomalies that have significant impact on span sequences or distribution of response time.

Table 3: Training and Testing Time of Different Approaches

Methods	Training Time	Testing Time
TraceAnomaly	30m	137s
MultimodalTrace	30.4m	148s
DeepLog	77.6m	278s
LogAnomaly	816m	3,136s
GRU-based Deep SVDD	138.8m	105s
DeepTraLog	67.1m	77s

The two log-based approaches (i.e., DeepLog and LogAnomaly) achieve high recall and low precision. These two approaches consider events within spans (we provide both span events and log events for them), thus can detect anomalies in span sequences and log event sequences. However, they rely on the prediction of the next event in a sliding window of the event sequence, thus have no global view of the trace and its structure. Therefore, they are likely to falsely report unseen event subsequences as anomalies. Actually, this kind of log anomaly detection approaches usually work on a single log event sequence such as the HDFS dataset [38]. Thus, they cannot well work for microservice anomaly detection which involves many service instances and distributed log events in parallel and asynchronous service invocations.

GRU-based Deep SVDD achieves better precision and recall than the four baseline approaches. It considers all the span events and log events of a trace, thus performs better than log anomaly detection approaches which rely on sliding window based log analysis. However, its sequence-based representation of span events and log events ignores the complex structures of traces brought by their invocation hierarchies and parallel/asynchronous invocations. Therefore, it does not perform as well as DeepTraLog.

Figure 6 shows the comparison of the results produced by DeepTraLog and GRU-based deep SVDD. We use t-SNE [36] to visualize the latent representations of the traces in the test set by projecting them to 2D space. It can be seen that DeepTraLog can learn a hypersphere that can better enclose normal traces and distinguish anomalous traces from normal ones. We believe that the improvement is brought by the unified graph representation of traces and logs.

In conclusion, DeepTraLog is effective in microservice anomaly detection and outperforms existing trace- and log-based anomaly detection approaches by 64.94% and 101.59% on average in terms of precision and recall respectively. The unified graph representation significantly contributes to the improvement of DeepTraLog, making it outperform the variant of DeepTraLog using sequence representation by 7.64% and 26.03% in terms of precision and recall respectively.

4.3 RQ2: Efficiency

Using each of the approaches, we train an anomaly detection model with the training set (including 65,490 traces) and test the model with the test set (including 56,080 traces). Table 3 shows the training time and testing time of each approach. These approaches take 30-816 minutes to train the models and 77-3,136 seconds to finish the test. In general, all the approaches except LogAnomaly are efficient, for example training a model in about 30-138 minutes and finishing the test (the whole test set) in 77-278 seconds. DeepTraLog is slower than the two trace-based approaches (i.e., TraceAnomaly and MultimodalTrace) but much faster than the others in model training. It is much faster than all the other approaches in testing.

The two trace-based approaches (i.e., TraceAnomaly and MultimodalTrace) are much faster than DeepTraLog in training, as they only consider traces. DeepTraLog considers both traces and logs, thus uses more time in training. However, DeepTraLog is much faster than them in testing. The reason is that the network parameters θ completely characterize the anomaly detection model and no data has to be stored for anomaly detection [30].

The two log-based approaches (i.e., DeepLog and LogAnomaly) use sliding windows to segment event sequences, thus produce a large number of subsequences for training and testing. Moreover, LogAnomaly uses an additional count vector sequence for each subsequence and the dimension of the count vectors is determined by the number of log events. Public log datasets usually have a small number of log events. For example, the HDFS log dataset [38] contains only about 40 log events. In contrast, our dataset has more than 800 log/span events, causing the curse of dimensionality for LogAnomaly. The large number of log/span events is popular in microservice systems, thus traditional log anomaly detection approaches cannot work well for microservice systems.

GRU-based Deep SVDD uses much simpler sequence representation for traces and logs, but is slower than DeepTraLog in both training and testing. The reason is that GGNNs treats nodes (events) in the graph as network units and conducts message passing in parallel, while GRU serially processes every event in a very long sequence.

The response time of online prediction is important for achieving timely anomaly detection. It highly depends on the size of the trace, i.e., the number of span/log events in it. To evaluate the scalability of DeepTraLog with trace size, we conduct an experiment on the changes of response time with the increase of event number. We divide the event number 0-900 into nine ranges, e.g., 0-100, 100-200, and for each range randomly sample 100 traces whose event numbers are within the range. For each trace we run the anomaly detection model 300 times and calculate the average response time of prediction. We compare the average response time of DeepTraLog and GRU-based Deep SVDD for traces of different sizes. Figure 7 shows the results. It can be seen that the response time of both approaches increases linearly with the size of the trace and DeepTraLog is always faster than GRU-based Deep SVDD.

In conclusion, DeepTraLog is slower than trace-based approaches by 122.19% and faster than log-based approaches by 52.65% in training; it is faster than trace-based approaches by 45.88% and faster than log-based approaches by 84.92% in testing. Moreover, the response time of DeepTraLog increases linearly with the size of the trace and it costs DeepTraLog around 4 milliseconds to make a prediction when the trace includes 800-900 events.

4.4 RQ3: Impact of Configurations

The hyperparameter μ in the loss function (see Equation 6) and the hidden layer number of GGNNs are two important parameters of the GGNNs based deep SVDD model. As stated in Section 3.5, μ controls the trade-off between the hypersphere volume and the

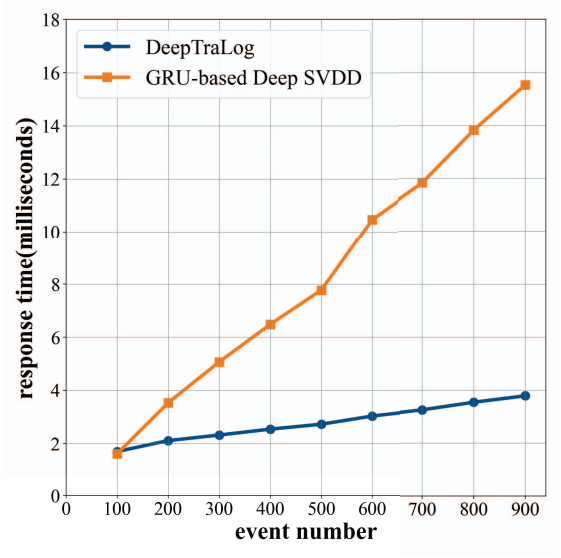
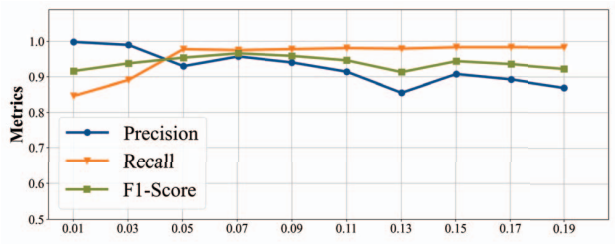
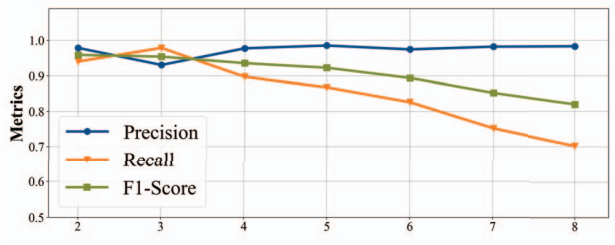


Figure 7: Changes of Response Time with the Increase of Event Number



(a) hyperparameter μ in Equation 6



(b) hidden layer number of GGNNs

Figure 8: Impact of Different Configurations

violations of the boundary. The hidden layer number of GGNNs determines the iterations of message passing between the nodes in a TEG, thus has great impact on the latent representation the TEG. The regularization parameter λ in Equation 6 usually can be empirically set (e.g., 0.001 in our implementation) as previous work [30].

Figure 8(a) shows the impact of μ on the precision, recall, and F1-score of DeepTraLog. In deep SVDD, μ usually can take a value between 0.01 and 0.1 [30]. Bigger μ usually leads to lower precision and higher recall as it makes a smaller hypersphere enclosing less normal traces and anomalous traces. According to the results, μ = 0.07 achieves the best trade-off in terms of F1-score. Note that the best configuration of μ highly depends on the characteristics of the dataset. It usually can be determined based on the preference on precision and recall.

Figure 8(b) shows the impact of the hidden layer number of GGNNs. It is usually set to two to four. It can be seen that both recall and F1-score decrease with the increase of hidden layer number. It is usually because that more hidden layers are more likely to lead to over-smoothing, which makes the features indistinguishable and thus hurts the classification accuracy [16].

4.5 Threats to Validity

The threats to the internal validity mainly lie in the implementation and configuration of baseline approaches and the process of dataset generation. We implement MultimodalTrace and LogAnomaly by ourselves as they have no publicly available implementations. These two approaches are based on standard deep learning (e.g., LSTM) and log event extraction (e.g., FT-Tree) components. We follow their papers and assemble the components in the same way. Regarding the impact of configuration, we experimentally choose the best configurations for all the baseline approaches. For DeepTraLog we experimentally investigate the impact of some key parameters on its effectiveness and report the results. The normal and anomalous traces in our dataset are generated by automatically executing the normal and faulty versions of the benchmark system respectively. It is thus possible that the normal traces may include latent anomalies. To alleviate the threat, we carefully test the involved scenarios of the normal version before execution and manually check the quality of a set of sampled traces after execution.

The threats to the external validity mainly lie in the benchmark system and fault cases. Our approach is only evaluated on TrainTicket and its fault cases. It is unclear whether it can be effectively used for more complex industrial microservice systems and fault cases. These threats are alleviated from two aspects. First, the dataset we construct includes complex traces including about 900 events and involving parallel and asynchronous service invocations. Typically a trace in industrial microservice systems involves dozens of service invocations and hundreds of log events. Therefore, the size of trace is comparable. Second, the fault cases of TrainTicket are replicated from real faults in industrial systems and cover different types of typical faults [42]. Therefore, the fault cases used in the evaluation are representative.

5 RELATED WORK

Traditional software anomaly detection approaches are mainly based on logs [7, 12, 18, 23, 24, 37, 39, 40]. Typically log anomaly detection consists of two stages. First, it extracts log events from log messages via log parsing. Widely used log parsing approaches include Drain [11] and Spell [6]. Second, it conducts anomaly detection on log event sequences. Early studies use numeric vectors to represent log sequences, which are usually generated by counting the numbers of various log event in log sequences. Lou et al. [23] propose an approach that mines the invariant relationships among

log events for anomaly detection. Xu et al. [37] use Principal Component Analysis (PCA) to detect log anomalies. Lin et al. [18] use the hierarchical clustering technique to identify log anomalies and He et al. [12] extend their work by identifying the correlations between logs and system metrics.

Recently, there are some approaches using deep learning to detect log anomalies. Du et al. [7] propose a log anomaly detection approach called DeepLog, which detects anomalies by predicting the next log event using LSTM. Similarily, Meng et al. [24] propose a log anomaly detection approach called LogAnomaly. It combines semantic vectors produced by word embedding and numeric vectors for the prediction of the next log event using LSTM. Zhang et al. [40] also use semantic vectors to represent log events and use BiLSTM to detect anomalies of the whole log sequence in an supervised manner. Yang et al. [39] extend [40] to a semi-supervised approach which estimates the labels of log sequences using PU learning. These log anomaly detection approaches are based on sequence representations of log events, thus cannot well detect anomalous traces that involve complex structures.

In microservice systems, traces are widely used in anomaly detection and root cause analysis [10, 19, 20, 25, 26, 28]. Traditional approaches [10, 42] rely on the visualization of traces to support manual trace analysis. Zhou et al. [42] conduct an empirical study on industrial practices of microservice fault analysis and debugging and propose an improved trace visualization method. Guo et al. [10] present an approach that uses trace aggregation and visualization to help the analysis of error propagation chains.

Recently, some researchers propose automated approaches for trace anomaly detection. Zhou et al. [43] propose a supervised approach to detect anomalous traces based on a set of features extracted from traces. It relies on fault injection to produce a large number of normal and anomalous traces for training and a set of predefined trace features extracted from different aspects. Other approaches detect various types of anomalies by learning patterns from the traces produced by normal execution [20, 25, 26]. Nedelkoski et al. [25] detect response time anomalies in microservice systems by training an Auto-Encoding Variational Bayes model. Nedelkoski et al. [26] represent each trace as a span sequence and a response time sequence and design a multimodal LSTM model to detect anomalous traces. Liu et al. [20] use a service trace vector to represent each trace, which treats each possible path as a dimension in the vector. These trace anomaly detection approaches do not consider logs. Moreover, they usually represent a trace as a sequence and do not consider the complex structures of traces.

6 CONCLUSION

In this paper, we have proposed DeepTraLog, a deep learning based microservice anomaly detection approach. It uses a unified graph representation to depict the complex structure of a trace together with log events embedded in the structure. Based on the representation we design a GGNNs based deep SVDD model which can learn a latent representation for each trace and a minimized data-enclosing hypersphere. We use the model to detect anomalous traces by calculating their distances to the center of the hypersphere. We have evaluated DeepTraLog on a microservice benchmark. The results show that DeepTraLog significantly outperforms state-of-the-art trace/log anomaly detection approaches. Our future work will extend DeepTraLog to support more different kinds of anomalies of microservice systems such as response time anomalies, and on the other hand evaluate DeepTraLog with different kinds of microservice systems.

7 DATA AVAILABILITY

All the data and results of the work can be found in our replication package [5].

REFERENCES

[1] Alibaba. 2021. ARMS. Retrieved August 15, 2021 from https://www.aliyun.com/product/arms

[2] Apache. 2021. Log4j. Retrieved August 15, 2021 from https://logging.apache.org/log4j/2.x/

[3] AWS. 2021. X-Ray. Retrieved August 15, 2021 from https://aws.amazon.com/xray

[4] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014*. ACL, 1724–1734. https://doi.org/10.3115/v1/d14-1179

[5] DeepTraLog. 2021. DeepTraLog. Retrieved August 25, 2021 from https://fudanselab.github.io/DeepTraLog/

[6] Min Du and Feifei Li. 2016. Spell: Streaming Parsing of System Event Logs. In *IEEE 16th International Conference on Data Mining, ICDM 2016*. IEEE Computer Society, 859–864. https://doi.org/10.1109/ICDM.2016.0103

[7] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In *2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*. ACM, 1285–1298. https://doi.org/10.1145/3133956.3134015

[8] elastic. 2021. Elasticsearch. Retrieved August 15, 2021 from https://www.elastic.co/elasticsearch/

[9] GloVe. 2021. GloVe. Retrieved August 15, 2021 from https://nlp.stanford.edu/projects/glove/

[10] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. 2020. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2016*. ACM, 1387–1397. https://doi.org/10.1145/3368089.3417066

[11] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. 2017. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *2017 IEEE International Conference on Web Services, ICWS 2017*. IEEE, 33–40. https://doi.org/10.1109/ICWS.2017.13

[12] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2018. Identifying impactful service system problems via log analysis. In *2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018*. ACM, 60–70. https://doi.org/10.1145/3236024.3236083

[13] Jaegertracing.Io. 2021. Jaeger. Retrieved August 15, 2021 from https://www.jaegertracing.io/

[14] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015*. http://arxiv.org/abs/1412.6980

[15] James Lewis and Martin Fowler. 2014. Microservices a definition of this new architectural term. Retrieved August 25, 2021 from https://www.martinfowler.com/articles/microservices.html

[16] Qimai Li, Zhichao Han, and Xiao-Ming Wu. 2018. Deeper Insights Into Graph Convolutional Networks for Semi-Supervised Learning. In *Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18)*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 3538–3545. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16098

[17] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *4th International Conference on Learning Representations, ICLR 2016*. http://arxiv.org/abs/1511.05493

[18] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In *38th IEEE/ACM International Conference on Software Engineering, ICSE 2016*. ACM, 102–111. https://doi.org/10.1145/2889160.2889232

[19] Dewei Liu, Chuan He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshun Wu. 2021. MicroHECL: High-Efficient Root Cause Localization in Large-Scale Microservice Systems. In *43rd IEEE/ACM International*

Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021. IEEE, 338–347. https://doi.org/10.1109/ICSE-SEIP52600.2021.00043

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

[21] Logback. 2021. Logback. Retrieved August 15, 2021 from http://logback.qos.ch/

[22] LogPAI. 2021. Loglizer. Retrieved August 15, 2021 from https://github.com/logpai/loglizer

[23] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. 2010. Mining Invariants from Console Logs for System Problem Detection. In *2010 USENIX Annual Technical Conference*. USENIX Association. https://www.usenix.org/conference/usenix-atc-10/mining-invariants-console-logs-system-problem-detection

[24] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, and Rong Zhou. 2019. LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs. In *Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*. ijcai.org, 4739–4745. https://doi.org/10.24963/ijcai.2019/658

[25] Sasho Nedelkoski, Jorge S. Cardoso, and Odej Kao. 2019. Anomaly Detection and Classification using Distributed Tracing and Deep Learning. In *19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2019*. IEEE, 241–250. https://doi.org/10.1109/CCGRID.2019.00038

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

[27] Opentracing.io. 2021. OpenTracing. Retrieved August 15, 2021 from https://opentracing.io/

[28] Yicheng Pan, Meng Ma, Xinrui Jiang, and Ping Wang. 2021. Faster, deeper, easier: crowdsourcing diagnosis of microservice kernel failure from user space. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 646–657. https://doi.org/10.1145/3460319.3464805

[29] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014*. ACL, 1532–1543. https://doi.org/10.3115/v1/d14-1162

[30] Lukas Ruff, Nico Görnitz, Lucas Deecke, Shoaib Ahmed Siddiqui, Robert A. Vandermeulen, Alexander Binder, Emmanuel Müller, and Marius Kloft. 2018. Deep One-Class Classification. In *35th International Conference on Machine Learning, ICML 2018 (Proceedings of Machine Learning Research, Vol. 80)*. PMLR, 4390–4399. http://proceedings.mlr.press/v80/ruff18a.html

[31] Gerard Salton and Chris Buckley. 1988. Term-Weighting Approaches in Automatic Text Retrieval. *Inf. Process. Manag.* 24, 5 (1988), 513–523. https://doi.org/10.1016/0306-4573(88)90021-0

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

[33] skywalking.apache.org. 2021. Apache SkyWalking. Retrieved August 15, 2021 from http://skywalking.apache.org/

[34] TraceAnomaly. 2021. TraceAnomaly. Retrieved August 15, 2021 from https://github.com/NetManAIOps/TraceAnomaly

[35] Twitter. 2021. Zipkin. Retrieved August 15, 2021 from https://zipkin.io/

[36] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing Data using t-SNE. *Journal of Machine Learning Research* 9, 86 (2008), 2579–2605. http://jmlr.org/papers/v9/vandermaaten08a.html

[37] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. 2009. Largescale system problem detection by mining console logs. *Proceedings of SOSP'09* (2009).

[38] Wei Xu, Ling Huang, Armando Fox, David A. Patterson, and Michael I. Jordan. 2009. Detecting large-scale system problems by mining console logs. In *22nd ACM Symposium on Operating Systems Principles, SOSP 2009*. ACM, 117–132. https://doi.org/10.1145/1629575.1629587

[39] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. 2021. PLELog: Semi-Supervised Log-Based Anomaly Detection via Probabilistic Label Estimation. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021*. IEEE, 230–231. https://doi.org/10.1109/ICSE-Companion52605.2021.00106

[40] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, Junjie Chen, Xiaoting He, Randolph Yao, Jian-Guang Lou, Murali Chintalapati, Furao Shen, and Dongmei Zhang. 2019. Robust log-based anomaly detection on unstable log data. In *2019 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019*. ACM, 807–817. https://doi.org/10.1145/3338906.3338931

[41] Nengwen Zhao, Junjie Chen, Zhaoyang Yu, Honglin Wang, Jiesong Li, Bin Qiu, Hongyu Xu, Wenchi Zhang, Kaixin Sui, and Dan Pei. 2021. Identifying bad software changes via multimodal anomaly detection for online service systems. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 527–539. https://doi.org/10.1145/3468264.3468543

[42] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Trans. Software Eng.* 47, 2 (2021), 243–260. https://doi.org/10.1109/TSE.2018.2887384

[43] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *2019 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019*. ACM, 683–694. https://doi.org/10.1145/3338906.3338961

[44] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Benchmarking microservice systems for software engineering research. In *40th International Conference on Software Engineering, ICSE 2018*. ACM, 323–324. https://doi.org/10.1145/3183440.3194991