Automated Software Engineering (2024) 31:46
https://doi.org/10.1007/s10515-024-00445-w

Enhancing fault localization in microservices systems through span-level using graph convolutional networks

He Kong[1,2] · Tong Li[1] · Jingguo Ge[1,2] · Lei Zhang[1] · Liangxiong Li[1]

Received: 5 December 2023 / Accepted: 10 May 2024

Abstract

In the domain of cloud computing and distributed systems, microservices architecture has become preeminent due to its scalability and flexibility. However, the distributed nature of microservices systems introduces significant challenges in maintaining operational reliability, especially in fault localization. Traditional methods for fault localization are insufficient due to time-intensive and prone to error. Addressing this gap, we present SpanGraph, a novel framework employing graph convolutional networks (GCN) to achieve efficient span-level fault localization. SpanGraph constructs a directed graph from system traces to capture invocation relationships and execution times. It then utilizes GCN for edge representation learning to detect anomalies. Experimental results demonstrate that SpanGraph outperforms all baseline approaches on both the Sockshop and TrainTicket datasets. We also conduct incremental experiments on SpanGraph using unseen traces to validate its generalizability and scalability. Furthermore, we perform an ablation study, sensitivity analysis, and complexity analysis for SpanGraph to further verify its robustness, effectiveness, and flexibility. Finally, we validate SpanGraph's effectiveness in anomaly detection and fault location using real-world datasets.

Keywords Anomaly detection · Fault localization · GCN · Microservice

✉ Tong Li
litong@iie.ac.cn

He Kong
konghe@iie.ac.cn

Jingguo Ge
gejingguo@iie.ac.cn

Lei Zhang
zhanglei@iie.ac.cn

Liangxiong Li
liliangxiong@iie.ac.cn

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China

[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

1 Introduction

Microservices are a type of software system based on a distributed architecture, composed of multiple independent and autonomous services. Each service is designed to perform a distinct function and is equipped for independent deployment, scaling, and upgrades. This architectural style enhances the efficiency and flexibility of developing, deploying, and scaling applications (Shadija et al. 2017). Nonetheless, the proliferation of applications escalates the complexity of fault localization and troubleshooting processes (Zhou et al. 2018a). Quick and accurate fault localization is crucial to minimize downtime and maintain service quality. The current widely used solution for fault localization involves developers manually detecting and fixing issues based on logs or code reviews. However, existing research (Zhou et al. 2018b) indicates that manual fault diagnosis with limitations, notably long turnaround times.

Therefore, researchers have proposed solutions based on machine learning and deep learning to improve fault localization. Zhou et al. (2019) developed trace-level and microservice-level models for fault localization by extracting a set of features from system trace logs (Zhou et al. 2019). Additionally, the effectiveness of using log events in microservice interactions for fault localization has been demonstrated by Zhang et al. (2022a) leveraging the hierarchical structure and contextual features of system logs (Zhang et al. 2022a). Recently, Lee et al. (2023) proposed Eadro, a fault localization framework based on multiple data sources. It has proven the effectiveness of integrating traces, system logs, and key performance indicators Lee et al. (2023). However, despite the significant contributions made by these prior efforts, little attention has been paid to span-level fault localization in microservices systems.

In this paper, we propose an innovative framework that employs graph convolutional networks (GCN) for span-level fault localization within microservices systems. We construct a directed graph based on the collected data and extract node and edge features to define span-level anomalies in the microservice system as a GCN-based edge representation learning and classification task. The graph construction hinges on the detailed analysis of the invocation relationship and execution time among microservice components. We implement four components in Span-Graph: (1) **Data Collection and Parsing**. Collect system-generated monitoring metrics, trace logs, and configuration files for parsing. (2) **Graph Construction**. Construct a directed graph and extract the corresponding node and edge features. (3) **Model Training**. Utilize GCN to learn and understand the edge representations within the graph. (4) **Anomaly Detection and Fault Localization**. Employ the trained GCN model to detect anomalies and localize faults within the system. To evaluate the efficacy of SpanGraph, we construct two comprehensive datasets by capturing trace data, CPU and memory usage metrics, and configuration files from the Sock-Shop and TrainTicket benchmark microservice systems. The experimental outcomes reveal that SpanGraph surpasses all baseline models in performance, exemplified by a 12.52% enhancement in the F1-score on the SockShop dataset and an 8.13% improvement on the TrainTicket dataset.

Furthermore, We conduct incremental experiments with unseen traces to evaluate the generalizability and scalability of SpanGraph. In addition, we conduct an ablation study, sensitivity analysis, and complexity analysis for SpanGraph to further verify its robustness, effectiveness, and flexibility, as well as its ability in anomaly detection and fault localization. We also use an existing real-world fault dataset to verify the effectiveness of SpanGraph in a production environment. In summary, the main contributions of this paper include:

- We introduce an innovative graph construction methodology that assimilates monitoring metrics, trace logs, and configuration files to represent the execution time sequence between microservices.
- We propose SpanGraph, which defines span-level anomaly detection as a task of edge representation learning within a graph through GCN, and fault localization as a task of anomalous edge starting nodes within a graph. Abandoning the previous related work that used two models for the tasks of anomaly detection and fault localization respectively, it greatly improves the efficiency of microservice fault localization.
- To evaluate the performance of SpanGraph, we construct datasets from the Sock-Shop and TrainTicket microservice benchmarks and conduct comparative analyses with baseline methods. The results show the effectiveness and generalization ability of SpanGraph for anomaly detection and fault localization. We also verified the effectiveness of SpanGraph on real-world datasets.

The rest of the paper is organized as follows. We present a detailed literature review in Section 2, outlining existing fault localization techniques and their limitations. Section 3 introduces some background and notations we use. Section 4 describes our proposed methodology, including data collection and parsing, graph builder, model training, anomaly detection and fault localization. Section 5 provides a comparison of our approach with existing methods, demonstrating its efficacy through various experiments. Finally, we conclude this paper in Section 6.

2 Related work

In recent years, researchers have proposed various solutions for addressing the problem of fault localization in microservice systems (Li et al. 2021; Zhang et al. 2023; Sun et al. 2023). These solutions can be categorized into four types, based on the data sources utilized: log-based, trace-based, metric-based, and integrated multi-source data approaches.

2.1 Log-based

The log-based approach refers to constructing problem detection and identification models by parsing logs. Du et al. (2017) used Long Short-Term Memory (LSTM) networks (Hochreiter and Schmidhuber 1997) to transform system logs into natural language sequences, which allows it to automatically learn log patterns during normal execution. In addition, it supports online incremental updates to continuously adapt to new log patterns (Du et al. 2017). Meng et al. (2019) proposed a unified data-driven deep learning framework called LogAnomaly for anomaly detection in unstructured log streams. Le and Zhang (2021) introduced a log-based anomaly detection method that doesn't require log parsing. It uses the BERT (Kenton and Toutanova 2019) model to encode log messages and utilizes a Transformer model for anomaly detection (Le and Zhang 2021). Liu et al. (2023) proposed the log-based anomaly detection framework ScaleAD meets the practical requirements of log-based anomaly detection in cloud systems. This framework consists of a lightweight Triple-Based Detection Agent (TDA) and an expert module (Liu et al. 2023). The expert module combines feedback from language models like ChatGPT (Ouyang et al. 2022). Although log-based methods can uncover more informative causes, they are challenging to work in real time and require hiding anomaly information within the logs.

2.2 Trace-based

Trace-based methods collect information by comprehensively tracing execution paths and identifying root causes by analyzing latency deviations along these paths. Zhou et al. (2019) proposed MEPFL, which trains trace-level and microservice-level prediction models. This approach is based on a set of features defined on system trace logs and involves automatically executing system trace log collection on target applications and their fault versions (Zhou et al. 2019). Liu et al. (2021) introduced MicroHECL, which dynamically constructs service call graphs. It analyzes possible anomaly propagation chains on these graphs and designs customized models using machine learning and statistical methods for detecting various types of service anomalies, thereby improving efficiency (Liu et al. 2021). Zhang et al. (2022b) suggested TraceCRL, which constructs operation call graphs and trains graph neural network models using contrastive learning methods. This approach enhances the representation of microservice traces, and it introduces a trace data augmentation component to tackle category conflicts and representation consistency issues in contrastive representation learning (Zhang et al. 2022b). Chen et al. (2023) proposed TraceGra, a graph-based deep-learning method for microservice anomaly detection. It integrates trace data with performance metrics, employs graph neural networks and Long Short-Term Memory networks to extract topological and temporal features, and calculates anomaly scores using two distinct loss values (Chen et al. 2023).

2.3 Metric-based

The metric-based approach involves collecting metrics from individual services and utilizing them for neural network learning. Gan et al. (2019) employed Convolutional Neural Networks (CNN) (Kim 2014) to reduce dimensions and filter out microservices that do not impact end-to-end performance. Subsequently, they used LSTM networks to learn spatial and temporal patterns for identifying failing services and resources (e.g., CPU overhead) that lead to service performance degradation (Gan et al. 2019). Mariani et al. (2018) combined machine learning with graph-centric algorithms to detect anomalies in Key Performance Indicators (KPI) and uncover their causal relationships. This approach is further enhanced with algorithms based on centrality indices to pinpoint the erroneous resources causing and propagating anomalies (Mariani et al. 2018). Chen et al. (2022) proposed an adaptive performance anomaly detection method based on pattern sketches, which rapidly detects anomalies and provides explanations by extracting normal and abnormal patterns from metric time series. Audibert et al. (2020) introduced a fast and stable unsupervised anomaly detection method using adversarial training of autoencoders. Tested on multivariate time series data, this method has demonstrated robustness, efficient training, and high anomaly detection accuracy.

2.4 Multi-source data

Zhang et al. (2022a) combined the complex structure of call hierarchies and parallel/asynchronous calls with log events, employing a unified graph representation to capture the intricacies of traces. This approach trains a Deep Support Vector Data Description (Deep SVDD) model (Ruff et al. 2018) based on Gated Graph Neural Networks (GGNN) (Li et al. 2015) for anomaly detection (Zhang et al. 2022a). Lee et al. (2023) introduced Eadro, an end-to-end troubleshooting framework designed to diagnose faults in microservices using multiple data sources. Eadro seamlessly combines anomaly detection and root cause localization in two stages, leveraging multi-source data including traces, system logs, and key performance indicators (Lee et al. 2023). Ren et al. (2023) utilized three data sources-metrics, traces, and logs-to build a unified graph representation that elucidates complex dependencies among these elements. Their model, based on Spatio-Temporal Graph Convolutional Networks (STGCN) (Yu et al. 2017) and Deep SVDD, is trained for anomaly detection. Additionally, they developed an interpreter to translate binary results into understandable outcomes, aiding engineers in diagnosing and resolving system anomalies (Ren et al. 2023). Huang et al. (2023) introduced MSTGAD, a novel semi-supervised, graph-based anomaly detection method. This method merges three distinct data modalities-metrics, logs, and traces-into a twin graph of a microservice system, utilizing graph neural network technology to model their interrelationships. MSTGAD is designed for automatic and accurate real-time detection of anomalies in microservice systems (Huang et al. 2023).

3 Preliminaries

3.1 Background

In a microservices architecture, the independence and dynamism of individual services underscore the critical role of distributed tracing. Typically, user-initiated requests involve multiple microservices for completion. Consequently, tracing the service invocation chain for each request is pivotal in understanding the real-time dependency relationships and execution processes of these services.

To meticulously monitor the application, distributed tracing technology is employed to record the sequence of service invocations associated with a singular user request. This technology systematically organizes these spans under a globally unique trace ID, thereby constructing a comprehensive end-to-end invocation chain across multiple microservices. Each span represents a process in which a microservice responds to a user request, characterized by a unique span ID and encompassing crucial details such as timestamps and duration. Through distributed tracing, a comprehensive understanding of the entire lifecycle of a user request is attained, encompassing its initiation in the system through to the final response. This includes insights into the invocation relationships and durations of interactions among various microservices. Such detailed visibility is indispensable for both troubleshooting and optimizing performance. In instances of system delays or errors, distributed tracing enables the swift pinpointing of the affected chain via the Trace ID. Subsequently, the span data facilitates the identification of either performance bottlenecks or malfunctioning microservices. Consequently, distributed tracing has emerged as an essential component for observability and monitoring within the realm of microservices architecture.

3.2 Problem definition

The trace of microservices is represented as a directed graph. Each node in the graph represents a separate microservice request, such as "user/address" or "payment/payment verification". Each edge represents the order of invocation between microservice requests. Blue nodes represent primary instances of each microservice, while yellow nodes denote replicated instances

In this paper, our focus is on the automatic and precise detection and localization of span-level request invocation faults within a microservices system. To achieve this, we conceptualize a graph-based representation of invocation requests between microservices. As illustrated in Fig. 1, each node in this graph symbolizes a microservice request, originating from a work node instance within a Kubernetes cluster (2019). We define a trace as a sequence $Trace = \{span_i, i \in \mathbb{N}^+\}$, where each span, denoted as $span = \{v_m\text{-}\ e_{(m,n,i)}\text{-}\ v_n, m, n, i \in \mathbb{N}^+\}$, represents an invocation process from the initiation of a request to the receipt of a response. This process involves a node v, uniquely identified by a four-tuple(NodeId, InstanceId, ServiceName, ApiName). Each edge in the graph delineates the invocation relationship between service requests. We allow multiple edges from node m to node n, signifying diverse service invocation requests among different traces within the microservices system. The index i distinguishes between these multiple edges, indicating the i_{th} of the edge. In addition, our model permits edges from a node to itself, signifying scenarios where a user request triggers a singular microservice API request. Such self-loops correspond to individual traces.

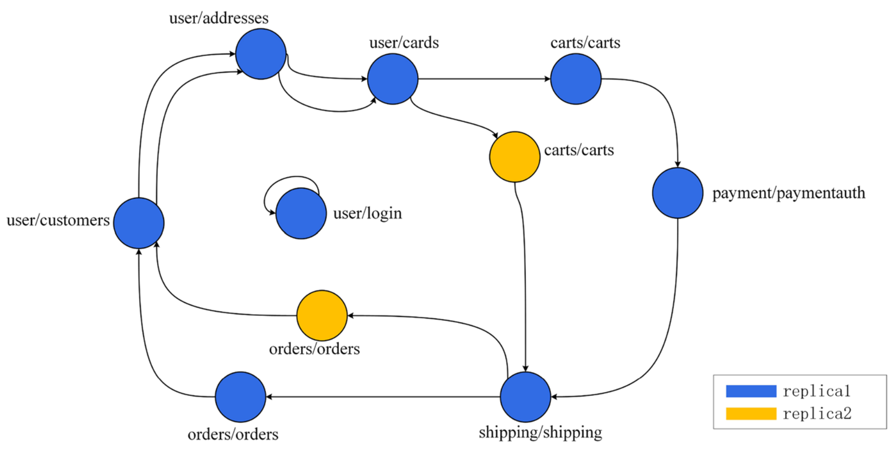


Fig. 1 The trace of microservices is represented as a directed graph. Each node in the graph represents a separate microservice request, such as "user/address" or "payment/payment verification".Each edge represents the order of invocation between microservice requests. Blue nodes represent primary instances of each microservice, while yellow nodes denote replicated instances

We define anomaly detection as an edge classification task on the graph (detecting anomalous microservice invocation relationships). Furthermore, we define fault localization as the start node of an anomaly edge on the graph (the location where a microservice fails).

3.3 Notations

In this paper, we represent the graph as $G = (V, E, X_v, X_e)$, where V denotes the vertex set comprising nodes $\{v_1, ..., v_n\}$. Each node is associated with a feature vector $X_v \in \mathbb{R}^{V\times d}$. Similarly, E represents the edge set, encompassing $\{e_1, ..., e_n\}$, with each edge possessing a feature vector $X_e \in \mathbb{R}^{E\times d}$. The symbol d is employed to denote the dimensionality of the embedding. The primary objective of our research is to develop a model, denoted as f, which is capable of learning these feature vectors for both nodes and edges. This model aims to effectively perform anomaly detection and fault localization within the system.

4 Method

In this section, we provide the overall workflow and details of the SpanGraph. Figure 2 displays the overview of the SpanGraph, which consists of four phases: **Data Collection and Parsing**, **Graph Builder**, **Model Training**, and **Anomaly Detection and Fault Localization**.

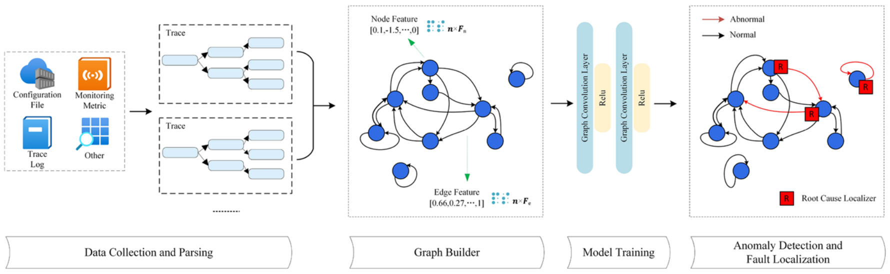


Fig. 2 Overview of SpanGraph Architecture

4.1 Data collection and parsing

We combine diverse data sources including monitoring metrics, trace logs, and configuration files. This integration enables the acquisition of comprehensive insights into various aspects such as the execution process of requests, the consumption of CPU and memory consumption, execution times, and request statuses. Subsequently, the trace incorporates metric data and information from configuration files into the graph construction, thereby facilitating the generation of a unified graph representation. We utilize third-party tools (Apache 2023; Prometheus 2023) to collect trace data and other relevant information for graph builder. Subsequently, the collected data undergoes preprocessing. We fill missing values in the dataset with zeros and normalize the feature values to the range [0,1].

4.2 Graph builder

Figure 1 illustrates an example of our graph builder, which is composed of four-tuples and span events. The construction process involves several key steps:

1. **Creating four-tuple Nodes.** The first step in our graph builder is to create four-tuple nodes. Each four-tuple node is unique and represents a specific request or event within the system. These nodes are identified and created based on the collected trace data, which captures the various transactions and operations performed by the system.
2. **Analyzing Time Relationships between Nodes.** In the second step, we analyze the time relationships between the four-tuple nodes. This involves examining the timestamps associated with each four-tuple to understand the temporal order and dependencies between different interactions or events. This analysis helps to establish the sequence and flow of operations within the system.
3. **Connecting Nodes to Form a Graph.** The third step involves connecting the four-tuple nodes based on their time relationships. We create edges between nodes that are causally related or follow a temporal sequence, forming a directed graph. This graph structure captures the dependencies and flow of operations within the

system, allowing us to model the system's behavior and interactions in a more detailed and accurate manner.

By following these steps, we can construct a graph that represents the system's behavior and interactions in a comprehensive and structured manner. The resulting graph serves as a valuable tool for analyzing the system's performance, identifying anomalies, and understanding the underlying patterns and relationships within the system.

The Graph Builder is meticulously crafted to construct a graph from data derived through trace parsing, enabling the extraction of node and edge features. Consider a trace encompassing the request and response processes of four distinct microservices (A, B, C, D). This trace can be depicted as a directed graph, where each node corresponds to a four-tuple, and each edge delineates the request and response processes. By arranging these four-tuples in the order of their execution times, a directed graph is formed. The structure of this directed graph is defined as follows:

Node execution time order(denoting the total duration of service invocation, calculated as response completion time cr minus service start time cs) set: $\{A_1, A_3, A_2, C_1, B_1, D_1\}$

Edge set:$\{A_1 \rightarrow A_3, A_3 \rightarrow A_2, A_2 \rightarrow C_1, C_1 \rightarrow B_1, B_1 \rightarrow D_1, D_1 \rightarrow A_1\}$

Here, the subscript i denotes distinct requests from the same service. Consequently, an edge, such as from A_1 to A_3, represents the information generated throughout the request and response process of A_1.

4.2.1 Node feature

For nodes, we extract four distinct types of features. The first feature type is the node identifier, in which the four-tuple (NodeId, InstanceId, ServiceName, ApiName) is encoded into a 32-dimensional vector, uniquely distinguishing each node. We use the Word2Vec embedding model (Mikolov et al. 2013) to train and encode the four-tuple strings in the dataset. The embedding size of each word is 8. The second feature type encompasses the execution characteristics of each four-tuple, including execution count, average execution time, average memory usage, and CPU usage. These metrics collectively reflect the node's performance in handling microservice requests. To calculate the average execution time, memory, and CPU usage for each node, the cumulative values of these metrics for identical nodes are aggregated and then divided by the number of executions. The third feature type pertains to trace-related attributes, such as average trace duration and the proportion of the node of the trace, which illuminate the node's function within the trace. The average trace duration is derived by summing the durations of all requests within a trace path and dividing this total by the number of node executions. Similarly, the proportion of the node of the trace is computed by dividing the total number of requests observed during the trace process by the number of node executions. The fourth feature type involves encoding the trace invocation path, where nodes in a trace are represented as a 40-dimensional vector. This encoding captures the positional context of the nodes within the trace path.

4.2.2 Edge feature

For edges, we extract six distinct types of features that are crucial for representing the interaction relationships between microservices. These features include interaction features, temporal features, status features, deployment features, resource features, and trace features. They encompass interaction time, status, deployment status, resource usage, and trace context, which are all key aspects of microservice anomaly detection. By providing the model with a comprehensive representation of microservice interactions, these features contribute to effective anomaly detection and fault localization.

The first feature type pertains to the inter-microservice request invocation, including metrics such as the execution time and status code of the inter-microservice invocation. The second feature type encompasses time-related attributes, derived by subtracting the execution times of each node (four-tuple) in the path from the total trace duration. This process yields a sequence that reflects the relative execution time associated with each edge. The third feature type consists of status features, represented as sequences of status codes for each node along the trace path. These sequences effectively capture the execution status of the edges within the path. The fourth feature type is deployment-oriented, incorporating various parameters such as instance startup time, the count of node instances and replicas, the thread count of the current service instance, the number of instances specified in the configuration file, and the count of ready instances. These features collectively depict the deployment status of the instances by the edge. The fifth feature type relates to resource utilization, including memory and CPU usage of instances and nodes, alongside their resource constraints. This set of features illuminates the resource status of the instances by the edge. The sixth and final feature type is trace-focused, where the four-tuple of the start request, the current request, and the end request within a trace path are encoded into a 96-dimensional vector.

By integrating these node and edge features, we construct a graph, which serves as an information backbone for effective anomaly detection and fault localization.

4.3 Model training

The model training module aims to learn representations of graph edges by capitalizing on the features of neighboring nodes and latent structural characteristics of the graph, thereby augmenting classification performance. Consider a graph $G = (V, E, X_v, X_e)$ that includes a node feature matrix X_v and an edge feature matrix X_e, along with learnable weight matrices W_v and W_e. The algorithm's goal is to generate a d-dimensional edge representation $Z_{e_{(m,n,i)}}$ for each edge. The model consists of two graph convolutional layers, implemented using the DGL (2023) framework. Through a layer of convolution operations, GCN (Bruna et al. 2013) is capable of capturing information pertaining to first-order neighbors. Within a single GCN layer, the updated node feature matrix $H^{(1)}$ derived from the initial input features $H^{(0)} = x_v(x_v \in X_v)$ using the following steps:

$$H^{(1)} = ReLU(W_v^{(1)} \cdot (H^{(0)} + H_{N(v)}^{(1)})) \tag{1}$$

$$H_{N(v)}^{(1)} = Agg(H_m^{(0)}, \forall m \in N(v)) \tag{2}$$

where Agg represents a mean aggregation function that computes the average of node and edge representations, and N is a neighbor selection function that chooses the complete neighbor set of nodes or edges. It's worth noting that different aggregation and neighbor selection functions can be chosen based on specific requirements. For nodes or edges without neighbors, their representations are learned from their initial features. By stacking multiple convolutional layers, GCN can capture multi-scale information in the graph. The output of each layer $H^{(k-1)}$ serves as the input for the next layer:

$$H^{(k)} = ReLU(W_v^{(k)} \cdot (H^{(k-1)} + H_{N(v)}^{(k)})) \tag{3}$$

$$H_{N(v)}^{(k)} = Agg(H_m^{(k-1)}, \forall m \in N(v)) \tag{4}$$

Node representations are initially learned by aggregating neighboring node representations and those from the previous layer. Then, the graph convolutional layer aggregates neighboring edge representations and combines vectors of the two connecting nodes to compute an average. This average is subsequently passed through an activation function to form edge representations:

$$S^{(k)} = ReLU(W_e^{(k)} \cdot Concat(S_{e_{(m,n,i)}}^{(k-1)} + S_{N(e_{(m,n,i)})}^{(k)}, T^{(k)})) \tag{5}$$

$$T^{(k)} = Average(H_m^{(k)}, H_n^{(k)}) \tag{6}$$

$$S_{N(e_{(m,n,i)})}^{(k)} = Agg(S_j^{(k-1)}, \forall j \in N(e_{(m,n,i)})) \tag{7}$$

$$Z_{e_{(m,n,i)}} = S_{N(e_{(m,n,i)})}^{(k)} \tag{8}$$

During training, a cross-entropy loss function fine-tunes W_v and W_e to accurately predict edge labels. The training employs the AdamW (Loshchilov and Hutter 2017) optimizer via mini-batch stochastic gradient descent, with weight decay to mitigate overfitting.

4.4 Anomaly detection and fault localization

We use the trained model for anomaly detection and fault localization through two distinct methods. The first method involves constructing a graph using the entire dataset, which we refer to as the pre-trained graph. This pre-trained graph is then partitioned into training and test sets. The training set has labels, while the test set is designated as an unlabeled edge. SpanGraph trains on the training set and then performs subsequent anomaly detection on that test set. The second approach, which is illustrated in Fig. 3, is particularly suited for scenarios where data is generated incrementally. As new traces become available, we employ a graph builder process to seamlessly integrate this new information into the existing pre-trained graph. This incremental graph construction ensures that the model remains up-to-date with the latest data trends and system behaviors. Once the new graph is merged with the pre-trained graph, the trained model is utilized to detect anomalies on the resulting merge graph. This method is critical for real-time monitoring and adaptive fault localization, as it allows the system to continuously learn and adapt to evolving data streams without necessitating a complete retraining of the model.

The results of employing these methods are discussed in the subsequent sections, where we analyze the performance of SpanGraph in detecting and localizing anomalies, and we compare the efficacy of the two approaches in the context of the diverse datasets and use cases studied. Notably, fault Localization refers to locating the abnormal service (the starting node of the edge that is determined to be abnormal).

5 Experiments

In this section, we commence by delineating the experimental setup. Subsequent to this, a series of experiments are conducted across various datasets and benchmarked against multiple baselines, followed by a thorough analysis of the results. Additionally, to provide a comprehensive evaluation of the model, we address the following research questions (RQs):

RQ1: What is the impact of each component on the performance of SpanGraph? (Sect. 5.3)?
RQ2: How does the complexity of SpanGraph influence its efficiency and scalability? (Sect. 5.4)?

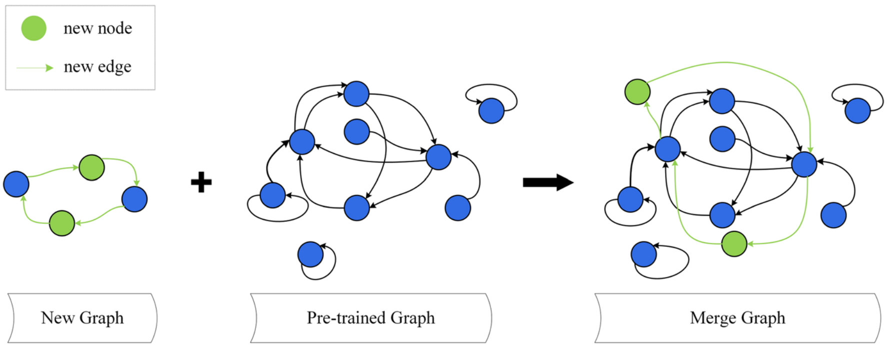


Fig. 3 The new trace is incrementally merged into the pre-trained graph after graph construction, followed by anomaly detection

RQ3: To what degree do variations in hyperparameters affect the effectiveness of SpanGraph (Sect. 5.5)?
RQ4: To what extent does SpanGraph demonstrate effectiveness when applied to real-world fault datasets (Sect. 5.6)?

5.1 Experiment setup

5.1.1 Dataset

Our experiment utilized two open-source microservices systems: Sockshop (2023) and TrainTicket (2023). Sockshop is primarily developed in Java, Golang, and Node.js and functions as a user-oriented online sock-selling e-commerce system, incorporating 8 distinct microservices. TrainTicket, a comprehensive train ticket reservation system, integrates 45 microservices and is implemented in a variety of programming languages, including Java, Golang, and Python. To facilitate these experiments, we established a Kubernetes distributed testing cluster employing 15 virtual machines, of which three served as master nodes. Drawing inspiration from the fault scenarios provided by Trainticket, we systematically injected various faults into the system. The categories of these faults include:

Asynchronous Interaction Fault: Asynchronous invocations between microservices occasionally result in message loss or sequence generation of exceptions. To simulate this, we employed a code modification method that introduced mechanisms for missing messages or injected data dependencies. Expected faults in this scenario include interruptions in business process flows.

Multi-Instance Fault: Data inconsistency issues often arise when multi-instances of a microservice concurrently access a local cache. To replicate this condition, our approach involved altering the shared configuration to one that is local to each instance. The faults expected from this modification include a range of service logic errors, such as calculation inaccuracies and null pointer exceptions.

Configuration Fault: Due to resource constraints in the environment, a microservice is unable to acquire sufficient computing resources to handle requests. To simulate this, we integrated resource-intensive operations into the request processing code. Expected faults include request timeouts and out-of-memory exceptions, typically associated with computational resource limitations.

Fault injection produces a series of faulted versions of a target microservice application by introducing different types of faults into different parts of the application. We inject application faults in the same way as in previous work (Zhou et al. 2019; Zhang et al. 2022a, 2022b; Li et al. 2021). For example, microservice request A_2 is injected with memory faults, then compiled and appropriate test cases are executed to further validate the fault injection results. We construct the trace as a graph following Section 4.2, and the edge $A_2 \rightarrow C_1$ at node A_2 is labeled by us as an anomaly edge, and the starting node A_2 of this anomalous edge is the fault occurrence point.

We utilized automated test cases (Query 2023; Locust 2023) to simulate user requests and generate corresponding data. To collect traces, we employed a distributed tracing framework Apache SkyWalking (Apache 2023). Additionally, we used

Prometheus (2023) to gather performance metrics relevant to both the traces and the containers. Utilizing the collected traces and metrics, we constructed a span-level graph structure. In the end, we obtained two datasets:

Sockshop Dataset: This dataset comprises 191,574 normal traces and 32,238 faulty traces, yielding a normal-to-faulty data ratio of approximately 17:1.

TrainTicket Dataset: This dataset comprises 256,341 normal traces and 47,679 faulty traces, yielding a normal-to-faulty data ratio of approximately 19:1.

5.1.2 Baselines

In evaluating our model, we conducted comparisons with previous methods. We selected baseline methods that are closely related to the problem domain and share similar objectives or underlying principles with our proposed method. Considering the fine-grained of our task, which focuses on span-level anomaly detection as opposed to the more general trace-based anomaly detection, a direct comparison with prior methods may not be entirely equitable. To mitigate this, we selected baselines that are also adept at span-level anomaly detection and all baselines are optimal hyperparameters:

- MEPFL-RF (Zhou et al. 2019) is a random forest algorithm based on system tracing logs to predict latent errors and faults in microservice applications and accurately locate latent errors caused by faults.
- MEPFL-KNN (Zhou et al. 2019) is a K-nearest neighbor algorithm based on system tracing logs to predict latent errors and faults in microservice applications and accurately locate latent errors caused by faults.
- MEPFL-MLP (Zhou et al. 2019) is a multi-layer perceptron neural network approach based on system tracing logs to predict latent errors and faults in microservice applications and accurately locate latent errors caused by faults.
- TLCluster (Sun et al. 2023) is a microservice system fault localization method based on trace log clustering that calculates the similarity of normal and abnormal trace logs.
- Span-RF is a variant of Span-Graph, which replaces GCN with RF in model training.
- Span-KNN is a variant of Span-Graph, which replaces GCN with KNN in model training.
- Span-MLP is a variant of Span-Graph, which replaces GCN with MLP in model training.

5.1.3 Implementation details

The evaluation metrics are predicated on four fundamental classifications: true positives (TP), which represent the correctly predicted positive samples; true negatives (TN), denoting the accurately predicted negative samples; false positives (FP), indicating the erroneously predicted positive samples; and false negatives (FN), reflecting the incorrectly predicted negative samples. To rigorously assess and quantify the effectiveness of SpanGraph, we employ four conventional metrics: accuracy, precision, recall, and F1-score. These metrics are defined as follows:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{9}$$

$$precision = \frac{TP}{TP + FP} \tag{10}$$

$$recall = \frac{TP}{TP + FN} \tag{11}$$

$$F1 - score = \frac{2 * precision * recall}{precision + recall} \tag{12}$$

In our experiments, we use an AdamW optimizer with an initial learning rate of 0.01 and use ReLU (Agarap 2018) as the non-linear activation function. We set the weight for L2 loss as 5e-4, the total steps are 500 and the dropout rate is 0. SpanGraph included two graph convolutional layers, where the dimension of the hidden layer is 16. For each dataset, data partitioning into training, and testing sets was executed in an 8:2 ratio, respectively. We evaluated baseline models and SpanGraph on three datasets via 5-fold cross-validation. This involved dividing each dataset into five equal-sized subsets, or folds. In each iteration of the cross-validation process, four of these folds were used for training the models, while the remaining fold was reserved for testing. This procedure was repeated five times, with each fold serving as the test set exactly once. We implemented the approach in baseline using the scikit-learn (2023) machine learning library. Our model was implemented using Pytorch. The operating environment is roughly as follows: Ubuntu 20.04 with the kernel 5.4.0-126-generic, 64GB memory, and an NVIDIA Tesla V100S GPU.

5.2 Comparison experiments

5.2.1 Overall performance

The comparison results on the Sockshop and Trainticket datasets are shown in Table 1. According to Table 1, we can draw the following conclusions: SpanGraph achieves the best performance compared to several baselines in the SockShop and TrainTicket datasets. Traditional methods such as Random Forest (RF), K-Nearest Neighbors (KNN), and Multi-Layer Perceptron (MLP) show strong performance, yet they fall short of the results achieved by SpanGraph. This indicates that while traditional methods are still competitive, SpanGraph offers significant improvements, particularly in complex environments like microservices.

The performance of TLCluster on the two datasets is less than satisfactory. TLCluster relies on log data, but microservice systems are typically developed by different teams, resulting in uneven log quality, lack of a unified format, and

Table 1 The Experimental Results in SockShop, TrainTicket Datasets

Dataset	SockShop				TrainTicket			
Method	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score
MEPFL-RF	0.9750	0.8729	0.8318	0.8510	0.9658	0.9068	0.8404	0.8700
MEPFL-KNN	0.9495	0.7096	0.6807	0.6939	0.9533	0.8407	0.8257	0.8330
MEPFL-MLP	0.9554	0.7574	0.6000	0.6397	0.9629	0.9144	0.8072	0.8513
TLCluster	0.7476	0.8454	0.7476	0.7923	0.5017	0.8368	0.5017	0.6128
Span-RF	0.9896	0.9443	0.9167	0.9300	0.9761	0.9121	0.8825	0.8967
Span-KNN	0.9887	0.9337	0.9137	0.9234	0.9703	0.8714	0.8848	0.8780
Span-MLP	0.9866	0.9514	0.9396	0.9454	0.9823	0.9259	0.9269	0.9264
SpanGraph	**0.9978**	**0.9924**	**0.9605**	**0.9762**	**0.9926**	**0.9561**	**0.9465**	**0.9513**

Bold values represent the most significant and noteworthy results of our study, thereby highlighting the effectiveness of our experimental model

absence of contextual information, which hinders the analysis of dependencies between microservices. The Levenshtein distance similarity formula utilized by TLCluster takes into account the invocation order between microservice instances, yet it overlooks the instance count. Conversely, the cosine similarity formula focuses solely on the instance count, disregarding the invocation order, leading to varying effects on different fault types.

Notably, SpanGraph exhibits substantial improvement over the MEPLF-RF, MEPLF-KNN, and MEPLF-MLP methods. The F1-scores show an increase of 12.52%, 28.23%, 33.65% on the SockShop dataset, and 8.13%, 11.83%, 10% on the TrainTicket dataset, respectively. This emphasizes the robustness of SpanGraph in understanding and diagnosing diverse system behaviors. When comparing MEPLF with our method variants (Span-RF, Span-KNN, and Span-MLP), a distinct advantage is observed. Our feature extraction methods prove to be more efficacious in microservice fault localization, as evidenced by the substantial improvement in recall: 8.49%, 23.30%, and 33.96% improvements for Span-RF, Span-KNN, and Span-MLP, respectively, compared to MEPFL-RF, MEPFL-KNN, and MEPFL-MLP.

Our method consistently achieves higher F1-scores in anomaly detection across both datasets, compared to traditional approaches. This success is attributed to our graph-based anomaly detection algorithm's ability to effectively aggregate relevant features among neighboring nodes and comprehend the intricacies of microservice invocation relationships. SpanGraph not only enhances anomaly detection accuracy but also improves fault localization efficiency. The precision of SpanGraph is particularly noteworthy, with scores of 99.24% and 95.61% in the SockShop and TrainTicket datasets, respectively. Such high precision is critical for reducing false positives, which can lead to wasted resources and time in investigating non-problems. Our approach provides more reliable fault localization, which is essential to maintain high system reliability and reduce unnecessary debugging efforts.

The clear advantage of SpanGraph in both datasets suggests that incorporating the structural and temporal information of microservices interactions into the model results in superior fault localization capabilities. This integration is particularly relevant in microservice architectures, where comprehending service interaction patterns is essential for accurately identifying the source of failures.

5.2.2 Few-shot performance

Additionally, to validate the effectiveness and robustness of SpanGraph in few-shot settings, we conducted comparative experiments using varying data proportions in the SockShop and Trainticket datasets. We use data from the full dataset for graph construction as in Section 4.2, then randomly select 10%, 5%, and 1% of the edges to feed them into our model for training, and randomly use 20% of the full dataset as a test set. In Table 2, the comparison results illustrate that SpanGraph maintains a remarkably high level of performance even with limited data. Notably, even with only 1% of the train set, SpanGraph achieves F1-scores of 93% and 88.95% on the SockShop and Trainticket datasets, respectively. This demonstrates SpanGraph's ability to effectively learn and generalize from few-shot samples.

As the proportion of data increases to 5% and 10%, we observe uniform enhancements across all key metrics, such as precision, recall, and F1-score. These enhancements solidify the model's stability and reliability in few-shot learning scenarios. The robust performance in few-shot settings positions SpanGraph as a highly competitive model compared to traditional methods that often require large amounts of data to achieve similar levels of accuracy and precision. The efficiency of SpanGraph in learning from limited data can significantly reduce the need for extensive data collection and labeling efforts, which are often costly and time-consuming.

The results underscore the potential for implementing SpanGraph in real-world scenarios where data may be scarce or difficult to obtain, such as in the early stages of system deployment or in highly dynamic environments. In conclusion, the outcomes represented in Table 2 explicitly demonstrate the efficacy of SpanGraph in few-shot learning environments. Its ability to maintain high precision and recall with limited data exemplifies the adaptability and learning efficiency of SpanGraph, making it an ideal solution for practical fault localization in microservices systems with limited datasets.

Table 2 Comparison Results on Few-shot Training Dataset

Dataset	SockShop				TrainTicket			
Coverage	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score
1%	0.9935	0.9953	0.8728	0.93	0.9829	0.887	0.892	0.8895
5%	0.9966	0.9844	0.9389	0.9611	0.9861	0.9274	0.8925	0.9096
10%	0.9966	0.9792	0.9482	0.9635	0.989	0.9621	0.9029	0.9315
100%	0.9978	0.9924	0.9605	0.9762	0.9926	0.9561	0.9465	0.9513

5.2.3 Unseen trace performance

Finally, we evaluated the effects of incremental data on the SpanGraph. Our pre-trained graph model consists of 9,474 nodes and 334,529 edges. For testing, we randomly selected 1k, 2k, 3k, and 5k unseen traces as test graphs, which were then integrated into the pre-trained graph to examine the generalization performance of SpanGraph. As shown in Figure 4, the results indicate that with the introduction of unseen incremental data, our failure to learn the newly added trace led to a slight decline in accuracy or recall. However, as we progressively increased the number of unseen traces from 1k to 5k, SpanGraph consistently maintained a precision above 75%. This outcome suggests that SpanGraph possesses a notable degree of generalization ability.

Furthermore, the SpanGraph demonstrates appreciable scalability and generalization performance when handling incremental, previously unseen traces. This attribute is especially beneficial in dynamic systems, characterized by continual data generation. It implies that SpanGraph can adeptly adapt to new data without necessitating frequent retraining, a crucial advantage for effective and efficient fault localization in evolving microservices environments.

5.3 Ablation study (RQ1)

In this section, we conducted an ablation study on the TrainTicket dataset to analyze the performance of the SpanGraph. Specifically, we evaluated the

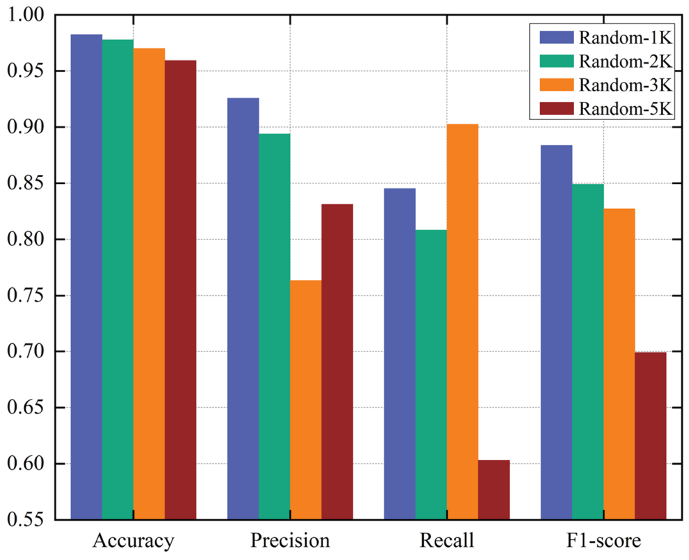


Fig. 4 The generalization performance of SpanGraph model pre-trained graph (incremental testing unseen trace datasets on Random-1K, Random-2K, Random-3K, and Random-5K graphs)

effectiveness of encoding features of nodes and edges and the sequences of duration time and response state, as well as CPU and memory features in our model. According to the ablation experimental results shown in Fig. 5, the following conclusions can be drawn about the impact of various features in the TrainTicket dataset on model performance:

Embedding Feature: The omission of embedding features resulted in a marginal reduction in recall and F1-score, by 1.96% and 0.94% respectively, compared to the full SpanGraph model. Accuracy and precision levels, however, remained largely unaffected. This outcome suggests that embedding features play a pivotal role in enhancing SpanGraph's ability to correctly identify positive samples, thereby improving both recall and F1-score. **Sequence Feature**: Features capturing the sequential information of traces, like duration time and response state, proved vital for the temporal analysis conducted by the model. The removal of these features led to decreased recall and F1-score, which are 3.48% and 2.35% lower than the original SpanGraph, underscoring their significance in anomaly detection. **Metric Feature**: The ablation of CPU and memory features notably impacted SpanGraph's performance, particularly in terms of recall. This implies that these system metrics are critical in signaling the health of microservices and ensuring precise anomaly detection. The decline in recall and F1-score was the most pronounced here, with decreases of 3.88% and 2.41%, respectively, highlighting the role of these features in detecting more abundant anomalies.

Answer to RQ1: The ablation study demonstrates that each feature contributes uniquely to the overall performance of SpanGraph. Their collec-

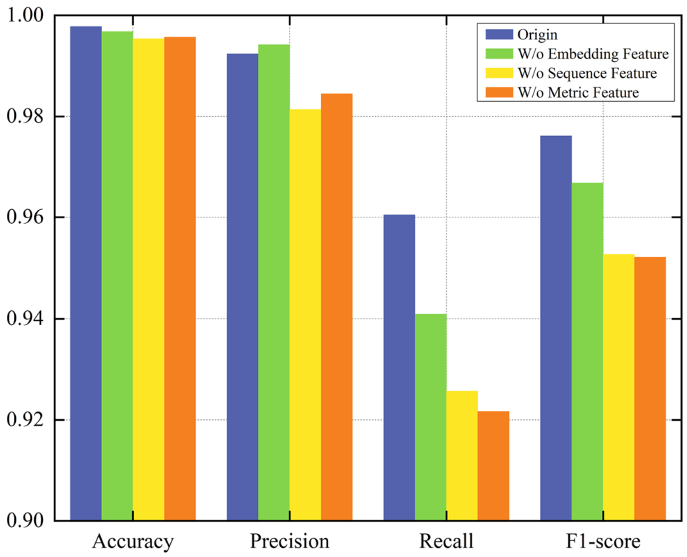


Fig. 5 Ablation study of key features in SpanGraph on the TrainTicket dataset

tive integration results in a synergistic effect, culminating in enhanced fault localization capabilities.

5.4 Model complexity analysis (RQ2)

To thoroughly evaluate the balance between model performance and complexity, we trained each model using the TrainTicket dataset, comprising 55,992 traces. This allowed for a comparative analysis of model complexity versus efficiency. Figure 6 presents the training durations for all methods, including the time taken to train our model on datasets of varying sizes.

From this analysis, it emerges that our model, while necessitating a relatively longer inference time and more computational resources, offers a justifiable trade-off when compared to the inefficiencies presented by the KNN and MLP methods. Notably, the RF method, despite its shorter training duration, delivers subpar performance, rendering it unsuitable for our objectives. Our model meticulously accounts for the interdependencies among service invocations, resulting in an extended training period.

Answer to RQ2: It is observed that the training duration for our model exhibits a gradual increase in correlation with the dataset's size expansion. This increment is indicative of our model's scalability and its ability to manage larger datasets without disproportionate increases in training time.

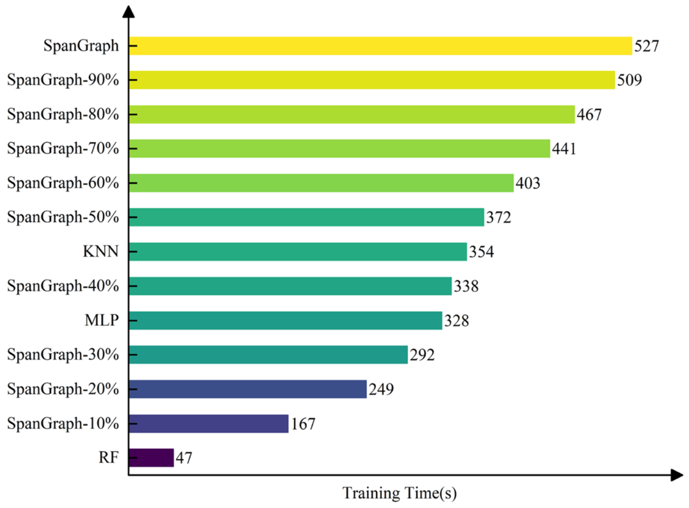
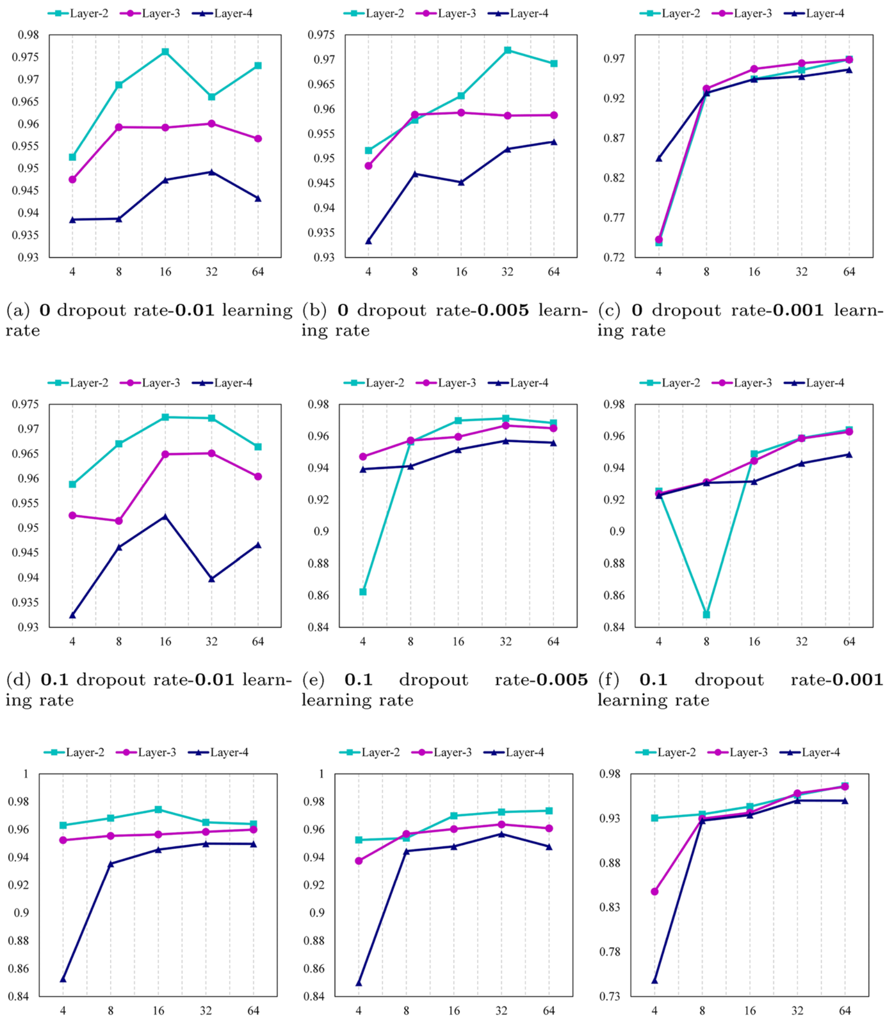


Fig. 6 The training time of different methods (such as RF, KNN, MLP, and SpangGraph with different dataset sizes) on the TrainTicket dataset

5.5 Model sensitivity analysis (RQ3)

In order to study the impact of hyperparameters on the model, we conducted sensitivity experiments, and the results are shown in Fig. 7. These experiments investigated the influence of varying the number of hidden layers, GCN layers, the learning rate, as well as the dropout rate on the F1-score of the model. From the results, we can draw the following insights:



(a) **0** dropout rate-**0.01** learning rate (b) **0** dropout rate-**0.005** learning rate (c) **0** dropout rate-**0.001** learning rate

(d) **0.1** dropout rate-**0.01** learning rate (e) **0.1** dropout rate-**0.005** learning rate (f) **0.1** dropout rate-**0.001** learning rate

(g) **0.2** dropout rate-**0.01** learning rate (h) **0.2** dropout rate-**0.005** learning rate (i) **0.2** dropout rate-**0.001** learning rate

Fig. 7 The F1-score results (Y-axis) of models with different hidden layers (X-axis) of the learning rates (e.g., 0.01, 0.005, 0.001) and the dropout rate (e.g., 0,0.1,0.2) on different GCN layer SpanGraph

Optimal Number of GCN Layers: The F1-score is notably influenced by the number of GCN layers, peaking at two layers. Additional layers tend to aggregate more noise, potentially impacting efficiency. Moreover, deeper GCN networks may lead to over-smoothing of features. **Impact of Hidden Layers**: An increase in hidden layers from 4 to 16 correlates with an enhanced F1-score. However, beyond 16 layers, this improvement plateaus or becomes less pronounced. At higher dropout rates (0.1 and 0.2), the benefit of more hidden layers persists. **Learning Rate Sensitivity**: The learning rate is a critical hyperparameter, influencing the rate of convergence. A learning rate of 0.01 consistently yields optimal performance across different configurations. At increased dropout rates(0.1 increased to 0.2), this learning rate remains preferable, though the model becomes more sensitive to learning rate variations, evidenced by sharper declines in F1-score at lower rates. Slower learning rates, like 0.001, may lead to lower performance due to slower convergence and the model's inability to adapt quickly to the complexities of the data within the given training epochs. **Dropout Rate Impact**: The dropout rate also plays a role in model performance. At a 0.01 learning rate, higher dropout rates negatively impact performance in models with fewer layers, indicating that excessive regularization could hinder learning. Conversely, at lower learning rates(0.005 and 0.001), the dropout rate's impact diminishes, suggesting the need for a balanced approach to prevent overfitting while enabling adequate learning.

It is noteworthy that there are points of instability, particularly observed in configurations with three or four GCN layers at a 0.001 learning rate and dropout rates of 0 and 0.2. This instability might indicate that this particular configuration is sensitive to initialization or may require a more tuned learning rate or regularization approach. Optimal performance is typically achieved with a balanced configuration: a dropout rate of 0, a learning rate of 0.01, a GCN layer of 2, and 16 hidden layers. While increased complexity (more hidden layers) can be beneficial, there is a threshold beyond which additional complexity does not yield proportional performance enhancements.

Answer to RQ3: Overall, SpanGraph exhibits robustness across various hyperparameter settings, maintaining high F1-scores.

5.6 Effectiveness for real-world fault (RQ4)

We conducted additional experiments based on the real-world datasets collected in Zhou et al. (2019) to further verify SpanGraph's effectiveness in the real-world. The dataset was generated by five student volunteers who acted as users to manually execute scenarios that might involve the target microservices. It contains 10 fault cases derived from the fault scenarios outlined in the TrainTicket benchmark. We extracted an average of 2,000 traces from each case to construct real-world fault dataset for this paper. This dataset contains a total of 20,000 traces, including 3,000 faulty traces. We conducted 5-fold cross-validation, partitioning our data into 80% training and 20% testing subsets at random for each fold. Table 3 lists the experimental results of four methods: Span-RF, Span-KNN, Span-MLP, and SpanGraph on the real-world fault dataset. According to Table 3, we can draw the following

Table 3 Effectiveness for Real-World Fault Dataset

Method	Accuracy	Precision	Recall	F1-score
Span-RF	0.9907	0.7032	0.8769	0.7805
Span-KNN	0.9904	0.7166	0.8323	0.7701
Span-MLP	0.9790	0.7959	0.8728	0.8326
SpanGraph	0.9888	0.8862	0.9301	0.9076

conclusions: SpanGraph achieves the best performance on the real-world fault dataset, with accuracy, precision, recall, and F1-score of 98.88%, 88.62%, 93.01%, and 90.76% respectively. This indicates that SpanGraph possesses strong anomaly detection and fault localization capabilities in microservice systems. Traditional methods such as Span-RF, Span-KNN, and Span-MLP also exhibit good performance, but in comparison, SpanGraph still holds significant advantages. SpanGraph's precision is particularly notable, reaching 88.62% on the real-world fault dataset. High precision is crucial for reducing false positives and avoiding unnecessary resource wastage.

Answer to RQ4: In short, the excellent performance of SpanGraph on the real-world fault dataset further proves its effectiveness in microservice anomaly detection and fault localization.

6 Summary and conclusions

In conclusion, our research presents a substantial advancement in the field of fault localization within microservices systems. Our proposed framework, SpanGraph, effectively leverages graph convolutional networks to offer a robust and efficient approach for span-level anomaly detection and fault localization. By integrating trace logs with monitoring metrics and configuration files, the directed graph model constructed by SpanGraph affords a comprehensive and intricate understanding of microservice interactions. This approach not only enables precise anomaly detection but also represents a considerable advancement over conventional manual diagnostic techniques. Our extensive experimental analysis, which includes ablation study, sensitivity analysis, and complexity analysis, confirms that SpanGraph not only surpasses existing baseline methods but also adapts well to unseen data, ensuring its applicability in real-world scenarios.

SpanGraph has several limitations for future improvement. Firstly, there is uncertainty about its real-world effectiveness as the model's complexity escalates with larger datasets. Secondly, the paper only explores a limited number of faults, raising questions about the model's capability to identify faults under real-world conditions.

In future work, we intend to deploy our proposed method in a real production environment to assess its adaptability to dynamic changes in system environments. Additionally, given the reliance on deep learning for anomaly detection, we will focus on exploring methods to minimize the training overhead of the model.

Acknowledgements This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB3103402.

Author contributions He Kong: Data analysis and Writing. Tong Li: Project administration. Jingguo Ge: Supervision. Lei Zhang: Validation. Liangxiong Li: Visualization. All authors reviewed the manuscript.

Funding This research was funded by the National Key Research and Development Program of China under Grant 2022YFB3103402.

Data and materials availability Not applicable.

Declarations

Conflict of interest The authors declare no competing interests.

Ethical approval Not applicable.

References

Agarap, A.F.: Deep learning using rectified linear units (RELU). arXiv:1803.08375 (2018)
Apache: Apache SkyWalking. http://skywalking.apache.org (2023)
Audibert, J., Michiardi, P., Guyard, F., Marti, S., Zuluaga, M.A.: USAD: Unsupervised anomaly detection on multivariate time series. In: Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining, pp. 3395–3404 (2020)
Bruna, J., Zaremba, W., Szlam, A., LeCun, Y.: Spectral networks and locally connected networks on graphs. arXiv:1312.6203 (2013)
Chen, Z., Liu, J., Su, Y., Zhang, H., Ling, X., Yang, Y., Lyu, M.R.: Adaptive performance anomaly detection for online service systems via pattern sketching. In: Proceedings of the 44th international conference on software engineering, pp. 61–72 (2022)
Chen, J., Liu, F., Jiang, J., Zhong, G., Xu, D., Tan, Z., Shi, S.: TraceGra: a trace-based anomaly detection for microservice using graph deep learning. Comput. Commun. **204**, 109–117 (2023)
DGL: Deep Graph Library. https://github.com/dmlc/dgl (2023)
Du, M., Li, F., Zheng, G., Srikumar, V.: DeepLog: anomaly detection and diagnosis from system logs through deep learning. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp. 1285–1298 (2017)
Gan, Y., Zhang, Y., Hu, K., Cheng, D., He, Y., Pancholi, M., Delimitrou, C.: SEER: leveraging big data to navigate the complexity of performance debugging in cloud microservices. In: Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems, pp. 19–33 (2019)
Hochreiter, S., Schmidhuber, J.: Long short-term memory **9**(8), 1735–1780 (1997)
Huang, J., Yang, Y., Yu, H., Li, J., Zheng, X.: Twin graph-based anomaly detection via attentive multi-modal learning for microservice system. arXiv:2310.04701 (2023)
Kenton, J.D.M.-W.C., Toutanova, L.K.: BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of NAACL-HLT, **1**, p. 2 (2019)
Kim, Y.: Convolutional neural networks for sentence classification. arXiv:1408.5882 (2014)
Kubernetes: Kubernetes. https://kubernetes.io (2019)
Le, V.-H., Zhang, H.: Log-based anomaly detection without log parsing. In: 2021 36th IEEE/ACM international conference on automated software engineering (ASE), IEEE. pp. 492–504 (2021)
Lee, C., Yang, T., Chen, Z., Su, Y., Lyu, M.R.: Eadro: An end-to-end troubleshooting framework for microservices on multi-source data. In: 45th IEEE/ACM international conference on software engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023, pp. 1750–1762 (2023)
Li, Z., Chen, J., Jiao, R., Zhao, N., Wang, Z., Zhang, S., Wu, Y., Jiang, L., Yan, L., Wang, Z., et al.: Practical root cause localization for microservice systems via trace analysis. In: 2021 IEEE/ACM 29th international symposium on quality of service (IWQOS), IEEE. pp. 1–10 (2021)

Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.: Gated graph sequence neural networks. arXiv:1511.05493 (2015)

Liu, D., He, C., Peng, X., Lin, F., Zhang, C., Gong, S., Li, Z., Ou, J., Wu, Z.: MicroHECL: high-efficient root cause localization in large-scale microservice systems. In: 2021 IEEE/ACM 43rd international conference on software engineering: software engineering in practice (ICSE-SEIP), IEEE. pp. 338–347 (2021)

Liu, J., Huang, J., Huo, Y., Jiang, Z., Gu, J., Chen, Z., Feng, C., Yan, M., Lyu, M.R.: Log-based anomaly detection based on EVT theory with feedback (2023)

Locust: Locust. https://locust.io/ (2023)

Loshchilov, I., Hutter, F.: Decoupled weight decay regularization. arXiv:1711.05101 (2017)

Mariani, L., Monni, C., Pezzé, M., Riganelli, O., Xin, R.: Localizing faults in cloud systems. In: 2018 IEEE 11th international conference on software testing, verification and validation (ICST), IEEE. pp. 262–273 (2018)

Meng, W., Liu, Y., Zhu, Y., Zhang, S., Pei, D., Liu, Y., Chen, Y., Zhang, R., Tao, S., Sun, P., et al.: LogAnomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In: IJCAI, vol. 19, pp. 4739–4745 (2019)

Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv:1301.3781 (2013)

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al.: Training language models to follow instructions with human feedback. Adv. Neural. Inf. Process. Syst. **35**, 27730–27744 (2022)

Prometheus: Prometheus. https://prometheus.io (2023)

Query, T.A.: TrainTicket Auto Query. https://github.com/FudanSELab/train-ticket-auto-query (2023)

Ren, R., Wang, Y., Liu, F., Li, Z., Xie, G.: Triple: the interpretable deep learning anomaly detection framework based on trace-metric-log of microservice. In: 2023 IEEE/ACM 31st international symposium on quality of service (IWQoS), IEEE. pp. 1–10 (2023)

Ruff, L., Vandermeulen, R., Goernitz, N., Deecke, L., Siddiqui, S.A., Binder, A., Müller, E., Kloft, M.: Deep one-class classification. In: International conference on machine learning, PMLR. pp. 4393–4402 (2018)

ScikitLearn: ScikitLearn. https://scikit-learn.org (2023)

Shadija, D., Rezai, M., Hill, R.: Towards an understanding of microservices. In: 2017 23rd international conference on automation and computing (ICAC), IEEE. pp. 1–6 (2017)

SockShop: SockShop. https://github.com/microservices-demo/microservices-demo (2023)

Sun, C.-A., Zeng, T., Zuo, W., Liu, H.: A trace-log-clusterings-based fault localization approach to microservice systems. In: 2023 IEEE international conference on web services (ICWS), IEEE. pp. 7–13 (2023)

TrainTicket: TrainTicket. https://github.com/FudanSELab/train-ticket (2023)

Yu, B., Yin, H., Zhu, Z.: Spatio-temporal graph convolutional networks: a deep learning framework for traffic forecasting. arXiv:1709.04875 (2017)

Zhang, S., Jin, P., Lin, Z., Sun, Y., Zhang, B., Xia, S., Li, Z., Zhong, Z., Ma, M., Jin, W., et al.: Robust failure diagnosis of microservice system through multimodal data. arXiv:2302.10512 (2023)

Zhang, C., Peng, X., Sha, C., Zhang, K., Fu, Z., Wu, X., Lin, Q., Zhang, D.: DeepTraLog: Trace-log combined microservice anomaly detection through graph-based deep learning. In: Proceedings of the 44th international conference on software engineering, pp. 623–634 (2022a)

Zhang, C., Peng, X., Zhou, T., Sha, C., Yan, Z., Chen, Y., Yang, H.: TraceCRL: contrastive representation learning for microservice trace analysis. In: Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering, pp. 1221–1232 (2022b)

Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Liu, D., Xiang, Q., He, C.: Latent error prediction and fault localization for microservice applications by learning from system trace logs. In: Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp. 683–694 (2019)

Zhou, X., Peng, X., Xie, T., Sun, J., Li, W., Ji, C., Ding, D.: Delta debugging microservice systems. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, pp. 802–807 (2018a)

Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., Ding, D.: Fault analysis and debugging of microservice systems: industrial survey, benchmark system, and empirical study. IEEE Trans. Software Eng. **47**(2), 243–260 (2018b)