Computer Fraud and Security
ISSN (online): 1873-7056

Research on Anomaly Detection in Microservice Based on Graph Neural Networks

Shuai Fang[1,*], Jingjian Chao[1], Zenghui Xiang[2], Xuan Chen[2], Mei Yan[3], Yinan Dou[3]
[1]China Electric Power Research Institute, Beijing 102299, China
[2]State Grid Jiangsu Electric Power Co., Ltd. Information & Telecommunication Branch, Nanjing 210024, Jiangsu, China
[3]State Grid Information & Telecommunication Branch, Beijing 100052, China
**Corresponding Author.*

Abstract:
With the development of the information and innovation industry, microservice architecture has become mainstream in software development due to its faster delivery, better scalability, and greater independence. However, microservice architecture also faces some operational challenges. The large number of services, complex dependencies, and varying resource demands make traditional log inspection or threshold configuration methods often inadequate for anomaly detection, making it difficult for operations personnel to quickly locate the root cause of failures. To address the difficulty of pinpointing fault nodes in microservice systems, this paper proposes the GraphSAGE with Attention algorithm based on graph neural networks. This algorithm combines the neighbor node sampling method of GraphSAGE with an improved attention mechanism to effectively locate anomalies in real-time using abnormal data. Compared to traditional methods, GraphSAGE with Attention does not require manual intervention and can locate anomalies by periodically collecting operational data that reflect fault characteristics. Additionally, GraphSAGE with Attention achieves better fault localization with lower resource usage while meeting real-time requirements. In this pa- per, we constructed a dataset of fault call chains using fault injection and simulated user calls on the Kubernetes microservice platform and tested the accuracy of the GraphSAGE with Attention algorithm in various anomaly and scenarios. Experimental results show that this algorithm achieves the best performance across multiple metrics. Therefore, the algorithm is of great significance for fault root location and has practical application value.

Keywords: microservices, anomaly detection, graph neural networks

INTRODUCTION

Microservice architecture, a modern software development practice [1-3], plays a crucial role in improving the flexibility, maintainability, and scalability of software systems. It achieves this by breaking down complex applica- tions into small, independent services, each running in its own process and interacting with other services through lightweight communication mechanisms such as HTTP RESTful APIs. This architecture allows information systems to better meet the se- curity, reliability, and rapid iteration needs of the information and innovation sector, providing a solid foundation for building a sustainable domestic software ecosystem. Despite its advantages in scalability, maintainability, and availability, microser- vice architecture faces challenges in real-world applications. As the number of services increases, maintenance costs rise rapidly [4]. Ensuring the smooth operation of dozens to hundreds of service nodes is a significant challenge for operations personnel. When a microservice system experiences anomaly, locating the root cause becomes more difficult. Due to the complex dependencies and interactions among services, an issue in one node can cause abnormalities in upstream and downstream nodes. In such cases, traditional log inspection or threshold configuration methods become ineffec- tive, making it difficult for operations personnel to quickly pinpoint the root cause of the failure, potentially leading to a cascading failure effect. Therefore, accurately and quickly locating the root cause of anomaly in microservice systems can signifi- cantly reduce the pressure on manual operations and is a critical issue that needs to be addressed in the field of microservices.

To address the root cause localization problem in microservices described above, this paper first analyzes and models the issue, presenting a mathematical model of the problem. Subsequently, we constructed a dataset from an actual microservice system suitable for this issue, which included data collection and call chain genera- tion. Lastly, this paper combines the neighbor sampling algorithm of GraphSAGE [5] with an attention mechanism to propose a GraphSAGE with Attention algorithm for root cause localization in microservice call chains. Unlike

the computation of the attention mechanism in the Graph Attention Networks (GAT) [6], which focuses on combining node features, GraphSAGE with Attention utilizes the call relation data between microservices themselves as the basis for attention. This allows the model to assign importance to nodes based on their role in the abnormal behavior, and then rank each node in the problematic call chain according to its likelihood of being the root cause. The entire process only requires the periodic collection of operational data reflecting fault features without manual intervention. However, traditional log inspection or threshold configuration methods require manual operations to quickly identify the root cause. Compared with GrahpSAGE, the GraphSAGE with Attention achieves better performance on metrics while maintaining lower resource consump- tion. Compared to recent methods such as Nezha [7], the advantage of our methods lies in its seamless integration with cloud-native monitoring systems, such as Prometheus. It does not require complex data processing and can directly utilize the monitoring results, thereby ensuring real-time performance.

To validate the effectiveness of the algorithm, experiments were conducted on the constructed dataset, and the results show that GraphSAGE with Attention achieved the best performance across various evaluation metrics. Ablation studies also demon- strated that the designed attention method significantly improved the accuracy of anomaly detection.

The primary contributions of this paper are described as follows:

(1) Proposing an attention mechanism algorithm more suitable for anomaly de- tection scenarios. The attention formula in graph attention neural networks is modified to determine the attention between nodes based on operational data from their call relationships, fully utilizing the characteristics of the call data. Experiments demonstrate that the anomaly detection model with the improved attention mechanism achieves better results.

(2) Proposing the GraphSAGE with Attention algorithm for solving root cause localization. Unlike traditional methods, the implementation of the Graph- SAGE with Attention algorithm does not require manual intervention, relying only on periodically collected operational data that reflect anomaly character- istics. It meets real-time anomaly localization requirements with low resource consumption.

(3) Experimental results on the dataset indicate that the GraphSAGE with At- tention algorithm achieves the best performance across all evaluation metrics. Therefore, this algorithm is of great significance for anomaly detection and has practical application value.

RELATED WORK

This section explores the current state of anomaly detection, with a particular focus on its practical applications and challenges.

Microservice

In recent years, as technology has advanced and the inter- net has evolved, modern internet applications have become increasingly large and complex. To meet the demands for high availability, maintainability, and scalability, application architectures have continually evolved, leading to the adoption of microservices architecture [8] This architecture breaks down a large application into multiple small services, each running in its own process and communicating via lightweight mechanisms, typically HTTP resource APIs. Compared to SOA (Service-Oriented Architecture), microservices rely more on lightweight technologies and native web development patterns such as REST and HTTP.

Microservices architecture divides a system into several independent services based on business functions, communicating through lightweight methods. Each service functions as an independent application, with its own development team and life- cycle, and can be deployed independently using the chosen technology stack. With the widespread adoption of Docker container technology [9] companies like Netflix, Twitter, and Amazon are increasingly employing microservices architecture.

The advantages of microservices architecture include faster delivery, improved scalability, and enhanced independence. It allows developers and testers to focus on their specific modules, facilitates the addition of new functionalities without altering existing code, and is suitable for agile development. Additionally, it enables targeted solutions to performance bottlenecks.

However, microservices architecture also faces challenges. As the number of ser- vices increases, the system becomes more difficult to manage and maintain. Opera- tions teams must invest considerable resources to ensure the smooth functioning of numerous service nodes. Furthermore, the complex dependencies between services make fault localization challenging once issues arise. Thus, accurate and rapid service fault localization is crucial for reducing operational pressure and is essential for the development of intelligent operational platforms.

Anomaly Detection

In the microservices context, the main task of anomaly detection is root cause localization, which involves identifying the anomal service among a large number of services. Root cause localization methods are closely re- lated to specific scenarios and can be broadly divided into two categories: log-based and call chain relationship/service metric-based approaches.

Early root cause localization methods primarily relied on analyzing log information. For example, Gertler [10] proposed threshold detection on monitoring data of individual components to identify anomalies. However, threshold-based detection methods are impractical in microservice architectures, as different services have distinct characteristics, making it impossible to apply a one-size-fits-all approach. Given the diversity of service characteristics, setting appropriate thresholds for each service is challenging. Logsurfer [11], which is based on pattern matching, analyzes and clus- ters logs, classifying service node logs based on the number of warnings and errors to determine the location of the anomaly service. While this pattern matching ap- proach is simple to implement, it requires regular updates to the regular expressions used for matching during service updates and maintenance, leading to potential reliability issues and false positives. X-trace [12], a cross-layer, cross-application network tracing framework, enables devices to log information related to each tagged network operation to reconstruct the user's task tree. However, this framework requires mod- ifications to the application platform's source code to capture service dependencies, making it unsuitable for the dynamic nature of microservice environments.

With the advent of container cluster management systems like Kubernetes [13,14], which provide service registration and discovery functions, deploying an Istio service mesh has become feasible. This allows for real-time collection of call relationships and metric data between services during operation, thereby generating service call chains. Recent research on root cause localization methods has focused on analyzing the relevant data from these generated service call chains. Brandon et al. [4] explained why graph representations are highly suitable for the state of microservices and proposed a graph similarity-based method to find the root cause of microservice anomalies. Pham et al. [15] used fault injection techniques to store call chain characteristics under various fault conditions and later determine fault locations based on the similarity of call chains when a fault occurs.

With the rise of machine learning and deep learning, some studies have introduced these methods into anomaly detection. Chen et al. [16] used Bayesian networks to study the dependency between machine metrics and downtime, and then employed the XGBoost model for prediction. Seer, developed by Gan et al. [17], uses deep learning to train on a large amount of collected and tracked cloud service data to identify the root cause of anomaly and the bottleneck resources leading to the fault. As suggested by its name, Seer mainly predicts which metric is likely to cause a system fault next by analyzing various service node metrics. Loud [18] is an online metricdriven root cause localization technique that collects all potentially fault-inducing metrics on the platform. It calculates the causal relationships between nodes based on previously collected metric changes and then computes the centrality index of each node in the abnormal subgraph to identify the likely anomaly service nodes. Loud is lightweight and does not require fault injection or system-impacting training to achieve root cause localization in specific systems. However, this method was only experimentally validated on a service with six nodes, selecting three nodes, and not all nodes demonstrated good localization accuracy, resulting in unstable localization accuracy.

Currently, some research uses various types of features, such as metrics, traces, and logs, achieving better detection results. Nezha [7] proposes an interpretable and fine-grained root cause analysis method that integrates multimodal data analysis. It transforms anomalies from different data sources into event patterns and compares the event patterns during non-failure and failure phases to pinpoint the root cause in an interpretable manner. Wang et al. [19]. used Hierarchical Reinforcement Learning from Human Feedback (HRLHF), addressing the

challenge of identifying root causes of anomalies in highly dynamic and complex microservice systems. By leveraging the system's static topology and engineer feedback to reduce uncertainties in the service dependency graph discovery, and significantly reducing query times through rein- forcement learning, they improved the accuracy and efficiency of dependency graph construction, thus enhancing the accuracy and robustness of root cause analysis.

Graph Neural Network

Graph-based data is structured data composed of a series of objects (nodes) and relationship types (edges). As non-Euclidean data, graph analysis is applied in areas such as node classification, link prediction, and clustering. Graph neural networks (GNNs) are deep learning methods based on graph domain analysis [20].

Graph Convolutional Networks (GCNs) extend CNNs to graphs, allowing them to directly process graph-structured data without converting it. GCNs are divided into two categories: spectral-based and spatial-based [21]. Since GCNs bridged the gap between spectral and spatial methods, spatial-based approaches have gained rapid development due to their attractive efficiency, flexibility, and versatility.

GraphSAGE [5] introduced a new node representation method that facilitates obtaining new node representations when the original graph changes. Unlike traditional GCNs [21], which sample the entire graph, GraphSAGE neighbors sampling algorithm samples only the k-hop neighbors of the current node (for k=1, only the directly connected neighbors). This sampling method is more suitable for root cause localization in microservices for two reasons: first, for real-time requirements, sampling only the nearby nodes can expedite the process, leading to quicker root cause localization results; second, fault injection practices reveal that anomaly in any node rarely affect the entire call graph, usually propagating only one or two hops. Thus, the two-hop neighbors around the root cause node better reflect fault characteristics. Sampling the entire graph might dilute the surrounding abnormal information. This paper adopts GraphSAGE sampling method for neighbor nodes. Additionally, various aggregators can be used for specific aggregation methods. GraphSAGE presents several aggregation functions, such as MEAN, Pooling, and LSTM aggregators. Given that the MEAN aggregator is suitable for unordered neighbor nodes and does not require concatenating the current node features with neighbor node features, it is chosen for the subsequent root cause localization algorithm in service operations.

The attention mechanism is now widely used in many deep learning tasks to amplify the impact of important parts of the data while reducing the influence of insignificant data or noise. Graph Attention Networks (GATs) [6] utilize the attention mechanism to assign different weights to different neighbors of a node. The training relies on pairs of neighbor nodes rather than the graph's structure. This paper integrates call chain relationships into the attention mechanism based on the original GAT, significantly enhancing the model performance.

METHODOLOGY

The Model of Root Cause

In the operation and maintenance of microservices, the primary requirement is typically to perform anomaly detection on various monitored key performance indicators (KPIs). Subsequently, it is necessary to analyze and localize the detected anomalies to promptly take further actions such as repairs and loss prevention. After collecting the KPI from various service nodes, anomaly detection needs to be conducted. If any KPI shows an anomaly in any dimension, it is crucial to quickly identify which services are anomaly and which service is the root cause. Generally, the operational anomaly in microservices are as shown in Table 1.

This paper focuses on root cause based on service call chains. A call chain includes two types of information: one is the attributes of the nodes themselves, such as memory usage; the other is the metrics resulting from the call relationships between nodes, which can be considered as edge weights, such as request latency. Additionally, there are attributes of the call dependencies between nodes, which can be represented as an adjacency matrix of the graph. From the cluster, we can extract call chain data that exhibits anomalies, forming graph-structured data.

We extract the data of call chains of a microservice system to construct its call graph and deployment diagram as shown in Figure 1. The graph contains n nodes, each with m features. All the features of the service nodes in the entire call graph can be represented by a feature matrix $\mathbb{S}^{m\times n}$.

$$\mathbb{S}^{m\times n} = \left\{\vec{S}_1, \vec{S}_2, \ldots, \vec{S}_n\right\} \tag{1}$$

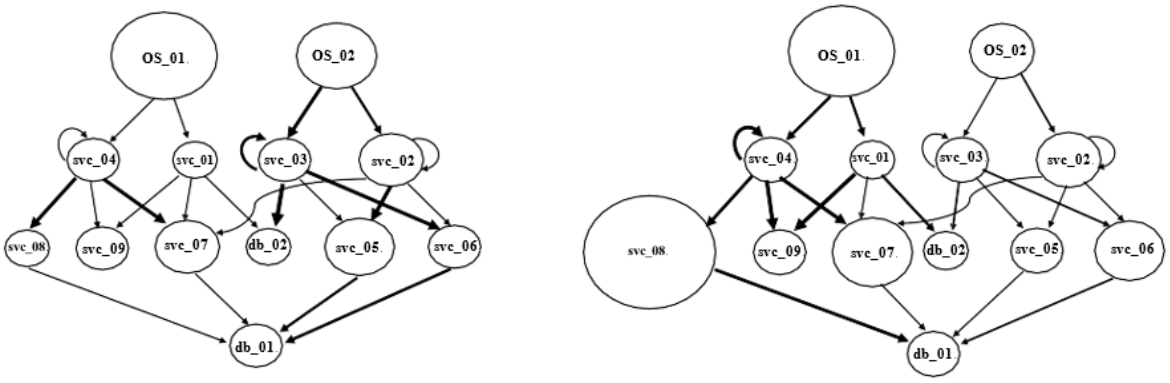

Figure 1. An example of microservice call graph

Each node in the figure represents a service, and the text within the node is the name of the service, e.g. os 01, svc 04, db 01. The edges represent the service call relationships between the services nodes. Bold lines indicate important call relationships, while thin lines represent regular calls.

Table 1. Common Operational Failures in Microservices

Anomaly	Description
F1	Network packet loss
F2	Network latency
F3	Packet duplication
F4	Memory leakage
F5	Insufficient CPU resources

The existence of a call relationship between nodes can be represented by the adjacency matrix A. Each element in A is either 1 or 0, representing the existence and non-existence of a call relationship.

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \tag{2}$$

For the invocation information between nodes (i.e., edge features in the graph), the edge feature matrix R, which has the same dimension as the adjacency matrix, is used for representation. If there is a call relationship between nodes i and j, then rij is its corresponding metric, otherwise it is 0.

$$R = \begin{bmatrix} r_{11} & \cdots & r_{1n} \\ \vdots & \ddots & \vdots \\ r_{n1} & \cdots & r_{nn} \end{bmatrix} \tag{3}$$

Since communication between services generates several metrics (e.g. number of network packets sent, response time, etc.), we represent them by a set R^D of length d, where d is the number of generated metrics.

$$R^D = \{R_1, R_2, \ldots, R_d\} \tag{4}$$

The major component of our model is a root cause classifier that can automatically infer the real root cause. The inputs of this classifier include the adjacency matrix A, the set of call relationship attributes R^D and the features matrix S of all nodes. At the time of fault occurrence, they are fed into the classifier to obtain a fault localization result vector $\vec{Z}$ of equal length to the number of service nodes. Each element zi represents the probability that node i becomes the root cause node of the fault.

$$\vec{Z} = \{z_1, z_2, \ldots, z_n\} \tag{5}$$

The optimization objective of the classifier is shown in the Equation (6), where $\vec{Y}$ is the true case of whether the node is the root cause, and the element in $\vec{Y}$ takes the value of 1 (or 0), representing that the current node is (or is not) the root cause of the failure, respectively.

$$\vec{Z}^* = \arg\min_{\vec{Z}} \| \vec{Z} - \vec{Y} \|_2^2 \tag{6}$$

In this work, the input and output of the algorithm are illustrated in Figure 2. In this model, the classifier produces a root cause localization result vector Z, and the actual anomaly and normal conditions are represented by vector Y. The elements of Ycan take on only two values: 1 indicates that the current node is the root cause of the fault, and 0 indicates that it is not. Both are vectors of length n. Thus, the training objective of the classifier can be expressed as minimizing the error between the output values and the true values.

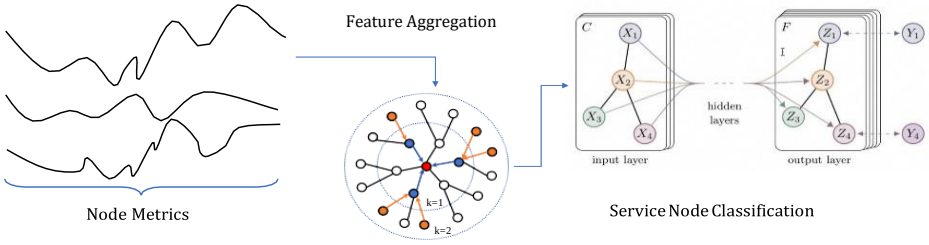

Figure 2. Illustration of the input and output of the root cause localization algorithm

GraphSAGE with Attention

Building on GraphSAGE, this research incorporates ideas from Graph Attention Networks (GAT) and adapts them to the specific scenario of fault service root cause localization on microservice platforms, introducing the GraphSAGE with Attention algorithm. The core idea of this algorithm is to integrate GAT's attention mechanism with GraphSAGE's neighbor sampling technique, using the call relationship data among microservices themselves as the attention mechanism. This approach not only ensures the timeliness of detection but also achieves excellent detection results.

In the dataset, there are two types of operational data related to microservices. The first type includes the node's own metric data, such as container memory, which serves as the node's individual indicators. The second type involves call-related data, such as request latency and TCP data transmission volumes. Since the call relationships are directional, the call graph is a directed graph, allowing edge features to be aggregated in two ways as the following equation:

$$\begin{aligned} s_{ib} &= \sum_{j \in N(i)} h_{ji} \\ s_{ib} &= \sum_{j \in N(i)} h_{ji} \end{aligned} \tag{7}$$

In this study, s_{ib} and s_{ie} respectively represent the aggregated features of node i acting as the source and the destination. $N(i)$ denotes the set of neighbors for node i, and hij and hji are the edge features connecting nodes i and j, where ij and ji represent the edges starting from node i to j and from j to i, respectively. The original node feature is represented as:

$$\vec{s}_i = (s_{i1}, s_{i2}, \ldots, s_{im}) \tag{8}$$

After aggregating edge information into node information, the amount of node information increases. If there are k types of edge information, the node information increases as follows:

$$\vec{s}_i = (s_{i1}, s_{i2}, \ldots, s_{im}, s_{ib}^1, s_{ib}^2, \ldots, s_{ib}^k, s_{ie}^1, s_{ie}^2, \ldots, s_{ie}^k) \tag{9}$$

Although this allows for each edge feature to be converted into two node features, the aggregation inevitably results in the loss of edge features. Also, in the sampling stage following aggregation, it is not possible to effectively utilize the varying edge weights to conduct differentiated sampling, which fails to highlight key information. Although GraphSAGE's model satisfies the needs of microservice root cause localization scenarios

in terms of neighbor node sampling, its initial model cannot handle graphs with edge weights. During the neighbor sampling stage, all of a current node's neighbors are sampled with equal probability and importance, which does not reflect the actual conditions in anomaly scenarios. If an anomaly occurs in a call, and the root cause node has multiple neighbors, its anomaly features will be diluted by indiscriminate sampling and aggregation of neighboring nodes, leading to inaccurate localization results. To address this limitation, GraphSAGE with Attention incorporates the attention mechanism from graph attention networks, applying weighted sampling to the neighbors of the sampled node. The attention computation method in graph attention networks is shown in the formula:

$$e_{ij} = Attention\left(W\vec{h}_i, W\vec{h}_j\right) \tag{10}$$

$$a_{ij} = \mathrm{Softmax}\left(e_{ij}\right) = \frac{\exp(e_{ij})}{\sum_{m\in N_i}\exp(e_{im})} \tag{11}$$

where, e_{ij} represents the importance of node j to node i. In this formula, j can be any node in the graph, not necessarily a neighbor of i. However, the final attention coefficient results focus on the structure of the graph: it only calculates e_{ij} for $j \in N_i$, where N_i represents all first-order neighbor nodes of i. To facilitate comparison of attention coefficients between different nodes and ensure that the sum of attentioncoefficients from the central node to all its neighbors equals 1, the softmax function is used to normalize all e_{ij} for $j \in N_i$. The attention calculation formula indicates that the importance of node j to node i is determined only by the surrounding neighbors, unrelated to the overall graph structure and other nodes in the graph. The traditional attention function calculates attention coefficients based on the weighted node features, specifically $W\vec{h}_i$ and $W\vec{h}_j$. Therefore, these coefficients are primarily determined by the features of the two nodes involved in the computation and vary with the weights.

To effectively utilize the edge weight information available in root cause localization scenarios, the proposed GraphSAGE with Attention uses the collected inter-node related information directly as attention values for neighbor node sampling. These attention values do not need to change with each training session like in the GAT model. Using inter-node related information directly as attention, the new attention calculation formula a'_{ij} is:

$$e'_{ij} = \sum_{h_{ij}^m \in L(h_{ij})} k_m h_{ij}^m \tag{12}$$

$$a'_{ij} = \mathrm{Softmax}\left(e'_{ij}\right) = \frac{\exp\left(e'_{ij}\right)}{\sum_{m\in N_i}\exp(e'_{im})} \tag{13}$$

where $L\left(h_{ij}\right)$represents all call relation data generated between nodes i and j, such as call latency and network packet counts, h_{ij}^m represents a certain type of call relation value, andk_m is a predefined parameter weight coefficient for a certain call relation, set according to the current call relation's impact on anomaly values, with the results normalized using the softmax function. After adding attention based on the inter-node call information, the output feature of the nodes is given by:

$$\vec{h}'_i = \sigma\left(\sum_{j\in N_i} a'_{ij} W\vec{h}_j\right) \tag{14}$$

where σ is a nonlinear activation function, N_i is the set of neighbor nodes for node i, $\vec{h}_j$ represents the feature of node j, and W is the learnable attention coefficient in the neural network, $W \in R^{F'*F}$.

Although the attention coefficients derived from call relation data are fixed and lack flexibility, it is possible to enhance the flexibility of the entire model by adding a trainable layer of attention on top of the unchangeable attention. It is well-known that the original graph attention networks used a single-layer feed-forward neural network with LeakyReLU as the activation function for the attention layer. In the model for root cause localization algorithms, the formula for calculating attention in the attention layer can be modified to:

$$a_{ij} = \frac{\exp\left(LeakyReLU\left(\vec{\boldsymbol{a}}^{\boldsymbol{T}}[W\vec{h}_i||W\vec{h}_j]\right)\right)}{\sum_{m\in N_i}\exp\left(LeakyReLU\left(\vec{\boldsymbol{a}}^{\boldsymbol{T}}[W\vec{h}_i||W\vec{h}_m]\right)\right)} + a'_{ij} \tag{15}$$

where a'_{ij} is the attention coefficient obtained from call relation indicator data as mentioned earlier, N_i represents the set of all first-order neighbor nodes of node i, $\vec{a}^T$ is the weight matrix of the feed-forward neural network, $\vec{a} \in R^{2F'}$, where F' is the feature dimension of nodes entering the next layer of the neural network, and W is the weight matrix of the graph convolution layer. This formula divides attention into two parts: one based on call relation indicator data and the other calculated by the GAT attention layer. By directly combining these two parts, attention coefficients that incorporate both call information and learning from the GAT layer are obtained. Based on this attention coefficient, the aggregated node feature representation at sampling depth k is expressed as:

$$S_i^k \leftarrow \sigma\left(W \cdot \left(\left\{S_i^{k-1}\right\} \| \operatorname{avg}\left\{a_{ij} \cdot S_j^{k-1}, \forall j \in N(i)\right\}\right)\right) \tag{16}$$

where S_i^k represents the aggregation result of node i at the k-th layer. Specifically, S_i^{k-1} is the feature representation of node i at the previous layer (i.e., the (k−1)-th layer). The symbol σ denotes the activation function, which is Sigmoid in this research. The symbol W represents a learnable weight matrix. The operator $\|$ indicates the concatenation of vectors, which combines the features of node i at the previous layer with the aggregated features from its neighboring nodes. The term avg takes the average of the neighboring node features, weighted by the attention coefficient a_{ij}, where a_{ij} is the attention coefficient that quantifies the importance of node j's feature in the aggregation of node i. Finally, N(i) denotes the set of neighboring nodes of node i.

In this paper, the selected key call relation metrics include the rate of change in request latency and the rate of change in TCP byte transmission over a short period. Different weight parameters are assigned to these metrics. These weight parameters can be used as hyperparameters in the neural network and can be adjusted according to different system requirements or to incorporate additional call information.

EXPERIMENTAL RESULTS

Datasets

This study uses two datasets: Sock shop and Mocker. We use Chaos Mesh to injuect different faults into these two datasets. Chaos Mesh is an open-source, cloud-native chaos engineering platform for conducting chaos tests on Kubernetes by intentionally injecting faults into clusters to assess their fault handling and recovery capabilities. Compared to other chaos engineering platforms, Chaos Mesh stands out due to its ease of use through YAML configuration and a userfriendly dashboard, and its ability to inject a wide range of faults into network, disk, filesystems, and operating systems.

(1) Sock Shop. Sock Shop is an open-source microservice application, which deployed on a locally established Kubernetes cluster. We injected three types of failures into five main service nodes of Sock Shop using the open-source chaos engineering tool Chaos Mesh: network latency, network packet loss, and memory leakage. We also collected fault data for each service using Istio and Prometheus. Approximately 100 call chain samplings were conducted per failure on each node (with a margin of error of ±10), completing a total of 42 experimental groups and gathering 23,100 node information records and 23,015 call chain records. Notably, the Frontend, which only serves the user interface, was excluded from the memory leak tests due to minimal benefit from such testing.

(2) Mocker. Mocker simulates the microservices data used in the business processes of the national power. It consists of 7 service nodes, making it quite large in scale with numerous business operations, complex invocation relationships, and closely resembling the real-world application scenarios and system operations of the national power grid.

Experimental Setup

To verify the performance and effectiveness of the algorithm based on graph neural networks with attention, we designed comparative experiments. These experiments compare our designed algorithm with random selection of fault points (Random) and contrast networks with attention (GraphSAGE with Attention) against those without attention (GraphSAGE). Since in practice these algorithms all achieve localization within seconds, there is little difference in meeting real-time requirements; therefore, our experiments focus primarily on comparing the accuracy of these algorithms.

Specifically, during the experiments, different anomaly were injected into different nodes over a period of time, and node and call chain information was collected during the fault periods for training. Each call chain will have only one type of fault occurring at a single node at any time.

This study employs Precision at top K(P@K) as the evaluation metric, where P@K represents the probability that the top K results provided by the algorithm are the root causes of the given fault. We plan to examine cases where K=1, 2, and 3, i.e., the frequency with which the top 1, 2, or 3 nodes listed by the classifier are the actual root causes of the anomaly. The formula is as follows:

$$P@K = \frac{1}{A}\sum_{a\in A}\frac{\sum_{i:r_a(i)\le K} R_a(i)}{\min(K,\sum_i R_a(i))} \quad (17)$$

Among these, A represents the set of all types of anomalies tested. ra(i) denotes the ranking position of each metric i, in the fault ranking given by the algorithm for anomaly a, Ra(i) = 1 indicates that metric i is the root cause of anomaly a, and Ra(i) = 0 indicates that metric i is not the root cause of anomaly a. Given K, the higher the (P@K) value, the more precise the localization. The parameters for the graph neural network model used for root cause localization, after multiple rounds of training and adjustment, were finally set as follows: a learning rate of 0.001, Adam optimizer, and CrossEntropy as the loss function. On the Sock Shop dataset, 70% of the data was used as the training set, and 30% as the test set. For the Mocker dataset, the model trained on the Sock Shop dataset was used directly for testing without a separate training set.

Regarding the parameters used in the model, the learning rate is set to 0.001, the optimizer is the Adam optimizer, and the loss function is the CrossEntropy function. On the Sock Shop dataset, 70% of the data is used as the training set and 30% as the test set. On the Mocker dataset, the model trained on the Sock Shop dataset is directly tested, without a training set.

Results and Analysis

The experimental results are shown in Table 2 to 5. Table 2 presents a comparative analysis of the performance of different methods across several service nodes, evaluated using the P@K (Precision at K) metric. The methods analyzed include Random, PageRank, GraphSAGE, and GraphSAGE with Attention. GraphSAGE with Attention consistently provides superior performance across all services, particularly at lower K values, which are critical for immediate and accurate anomaly detection.

Table 2. Comparative results of the root cause algorithm on the Sock Shop dataset (Average across nodes)

Service Nodes	P@K	Root Cause Method			
		Random	Page Rank	GraphSAGE	GraphSAGE with Attention
Frontend	P@1	0.06	0.85	0.95	0.94
	P@2	0.14	0.96	0.98	0.99
	P@3	0.19	0.99	0.99	1.00
Orders	P@1	0.07	0.9	0.88	0.99
	P@2	0.14	0.97	0.98	1.00
	P@3	0.19	1	0.99	1.00
Payment	P@1	0.06	0.95	0.57	0.78
	P@2	0.15	0.97	0.80	0.90
	P@3	0.20	0.99	0.96	0.96
Catalogue	P@1	0.08	0.83	0.33	0.56
	P@2	0.13	0.93	0.53	0.96
	P@3	0.20	0.98	0.71	0.99
Shipping	P@1	0.06	0.74	0.96	0.98
	P@2	0.15	0.91	1.00	1.00
	P@3	0.16	0.98	1.00	1.00

Table 3. Comparative results of the root cause algorithm on the Sock Shop dataset (Different Anomaly types)

Anomaly Forms	P@K	Root Cause Methods		
		Random	GraphSAGE	GraphSAGE with Attention
Network latency	P@1	0.06	0.78	0.88
	P@2	0.15	0.89	0.99
	P@3	0.19	0.92	0.99
Network packet loss	P@1	0.07	0.72	0.79
	P@2	0.14	0.84	0.96
	P@3	0.20	0.91	0.97
Network packet loss	P@1	0.08	0.65	0.86
	P@2	0.13	0.81	0.96
	P@3	0.17	0.95	1.00

Table 3 summarizes the experimental results based on different anomalies. From the experimental results in Table 2 and Table 3, it is evident that the GraphSAGE with Attention algorithm demonstrates excellent accuracy in localizing root causes on the Sock Shop dataset. In most cases, when only one root cause is allowed to be output, the localization accuracy is over 80%. However, accuracy decreases when the current fault node is closer to the root of the call chain and has a broad impact. For instance, in the scenario of network packet loss in the Catalogue service, the accuracy is only 12%. In cases of network latency and memory leaks, one node's localization accuracy is around 50%. Nonetheless, these nodes with low accuracy under single output conditions are almost always localized when the top three possible root causes are allowed to be output. The accuracy of GraphSAGE with Attention nearly reaches 100% at the P@3 metric, whereas the standard GraphSAGE without attention mechanisms still shows several cases of unsatisfactory accuracy, although both graph neural network algorithms perform significantly better than random selection across all scenarios. Additionally, although GraphSAGE with Attention generally outperforms GraphSAGE in experimental results, there are individual nodes and faults where this is not the case, such as in the network latency root cause localization at the Frontend node, where GraphSAGE achieves an accuracy of 96%, compared to 93% for GraphSAGE with Attention. When averaging the accuracy across all nodes and observing the average localization accuracy of GraphSAGE with Attention across each type of fault, its P@1 accuracy for network latency, network packet loss, and memory leaks is respectively 10, 7, and 21 percentage points higher than that of GraphSAGE, with a P@3 accuracy near 100%, surpassing GraphSAGE.

Furthermore, we set up ablation experiments to verify the effectiveness of the attention mechanism based on call chain relationships. The results are shown in Figure 3.

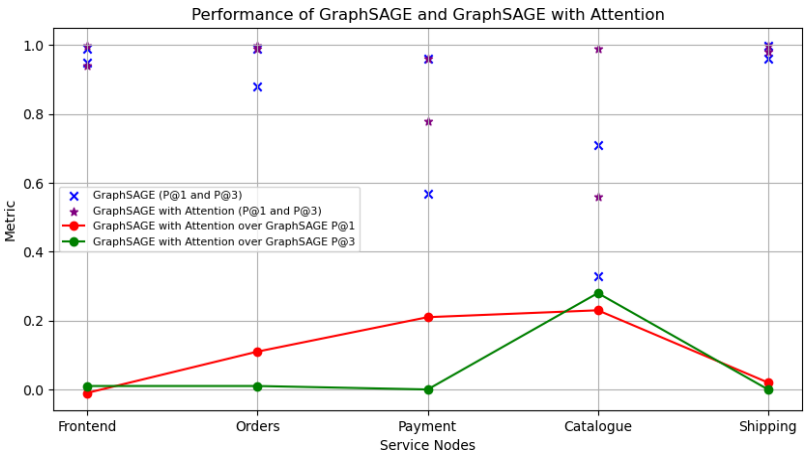

Figure 3. Comparison of results for GraphSAGE with Attention and GraphSAGE on the Sock Shop dataset.

This figure compares the performance of GraphSAGE with and without attention on the Sock Shop dataset, showing P@1 and P@3 results for different service nodes (Frontend, Orders, Payment, Catalogue, Shipping). Different symbols represent the comparison between P@1 and P@3 for each method.

Figure 3 displays the comparative results of GraphSAGE and GraphSAGE with Attention on the Sock Shop dataset for the P@1 and P@3 metrics, illustrating that, overall, the addition of the attention mechanism significantly enhances accuracy. In sections where there is no improvement, it is because the accuracies of both methods are nearly 100%, making it difficult to demonstrate a distinction.

From the Table 4 and Table 5, it is evident that on the Mocker dataset, which is closer to real-world application scenarios, the accuracy of the GraphSAGE with Attention fault localization algorithm is comprehensively higher than other methods. It only matches GraphSAGE in scenarios where the accuracy levels are already high, such as 1.00 and 0.99, indicating relatively simple fault detection tasks. As the complexity of faults increases, the advantages of GraphSAGE with Attention become more pronounced due to its ability to allocate sampling weights to critical information through its attention mechanism. For example, it exceeds GraphSAGE by 10 percentage points on the guard node for network latency faults, with an accuracy of 0.83; it is 13 percentage points higher for network packet loss on the kronos node, with an accuracy of 0.99; and it is 32 percentage points higher for memory leakage on the transaction node, with an accuracy of 0.90. On average, across various nodes, it consistently outperforms GraphSAGE by a margin of 7 to 26 percentage points. The GraphSAGE with Attention fault localization algorithm achieves an average accuracy of around 0.90 for different faults on the Mocker dataset, which is 6 to 18 percentage points higher than GraphSAGE.

Table 4. Comparative experimental results of fault localization algorithms on mocker (average across nodes)

Service Nodes	Root Cause Methods		
	Random	GraphSAGE	GraphSAGE GraphSAGE with Attention
biz	0.02	0.78	0.93
guard	0.01	0.69	0.76
kronos	0.03	0.91	0.99
ogs	0.02	0.62	0.62
gz	0.02	0.87	0.99
pay	0.02	0.88	0.98
transaction	0.02	0.69	0.95

This table shows the P@K (Precision at K) values for different nodes (such as biz, guard and so on) using three different methods: Random, GraphSAGE, and GraphSAGE with Attention.

Table 5. Comparative experimental results of fault localization algorithms on mocker (various faults)

Anomaly Forms	Random	GraphSAGE	GraphSAGE with Attention
Network latency	0.02	0.85	0.91
Network packet loss	0.02	0.75	0.91
Memory leakage	0.02	0.72	0.90

CONCLUSION

This paper primarily explores a method for root cause localization of operational faults in microservices architectures, introducing and implementing the GraphSAGE with Attention algorithm, which integrates features of GraphSAGE and graph attention neural networks. Compared to traditional root cause methods, GraphSAGE with Attention creatively applies graph neural network approaches to root cause analysis, where the entire localization process requires only the periodic collection of operational data that reflects fault characteristics without the need for human intervention. Under the premise of meeting real-time requirements for root cause localization, the GraphSAGE with Attention algorithm consumes fewer resources and improves the accuracy of root cause determination. Experimental results validate that the GraphSAGE with Attention algorithm achieves higher accuracy over multiple projects and fault types than GraphSAGE, proving its effectiveness on microservice platforms.

Although the algorithm designed in this paper has achieved impressive results, there are still some shortcomings that require further study:

1. While validating the effectiveness of the fault localization algorithm, the test data only involved call chains with a fault at a single node and did not test the algorithm's effectiveness in scenarios with multiple simultaneous faults. Future research could explore root cause localization algorithms for situations where multiple faults occur simultaneously.

2. Although the dataset in this study includes simulated data from a real scenario, there is still some discrepancy from actual user call data in real-world settings. If real data can be obtained in the future, further research could be conducted to explore the practicality of this algorithm.

ACKNOWLEDGMENTS

This work was supported by the Science and Technology Project Research and Application of Cloud Basic Environment Adaptation Technology for State Grid in the Context of XinChuang, under the project number 5700-202318307A-1-1-ZN.

REFERENCES

[1] Dragoni N, Lanese I, Larsen S T, et al., Microservices: How to make your application scale, International Andrei Ershov Memorial Conference on Perspectives of System Informatics, Springer, Cham, 2017, pp. 95-104.

[2] Khazaei H, Barna C, Beigi-Mohammadi N, et al., Efficiency analysis of provisioning microservices, 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2016, pp. 261-268.

[3] Di Francesco P, Malavolta I, Lago P., Research on architecting microservices: Trends, focus, and potential for industrial adoption, 2017 IEEE International Conference on Software Architecture (ICSA), IEEE, 2017, pp. 21-30.

[4] Brand´on ´A, Sol´e M, Hu´elamo A, et al., Graph-based root cause analysis for service-oriented and microservice architectures, Journal of Systems and Software, 159 (2020), 110432.

[5] Hamilton, Will, Ying, Zhitao, and Leskovec, Jure, Inductive representation learning on large graphs, Advances in neural information processing systems, 2017, pp. 1-11.

[6] Veli˘ckovi´c P, Cucurull G, Casanova A, et al., Graph attention networks, arXiv preprint arXiv:1710.10903, 2017.

[7] Yu G., Chen P., Li Y., et al., Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data, Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 553–565.

[8] Zimmermann O, Microservices tenets, Computing Research Repository, 32(3-4) 2017, pp. 301–310, https://doi.org/10.1007/s00450-016-0337-0.

[9] Merkel D, Docker: Lightweight Linux containers for consistent development and deployment, Linux Journal, 239(2) 2014, pp. 2.

[10] Gertler J., Fault detection and diagnosis in engineering systems, CRC press, 2017.

[11] Prewett J E., Analyzing cluster log files using logsurfer, Proceedings of the 4th Annual Conference on Linux Clusters, Citeseer, 2003.

[12] Fonseca R, Porter G, Katz R H, et al., X-Trace: A Pervasive Network Tracing Framework, 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07), 2007.

[13] Bernstein D., Containers and cloud: From lxc to docker to kubernetes, IEEE Cloud Computing, 1(3) (2014), pp. 81-84.

[14] Balalaie A, Heydarnoori A, Jamshidi P., Microservices architecture enables devops: Migration to a cloud-native architecture, IEEE Software, 33(3) (2016), pp. 42-52.

[15] Pham C, Wang L, Tak B C, et al., Failure diagnosis for distributed systems using targeted fault injection, IEEE Transactions on Parallel and Distributed Systems, 28(2) (2016), pp. 503-516.

[16] Chen Y, Yang X, Lin Q, et al., Outage prediction and diagnosis for cloud service systems, The World Wide Web Conference, 2019, pp. 2659-2665.

[17] Gan Y, Zhang Y, Hu K, et al., Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices, Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems, 2019, pp. 19-33.
[18] Mariani L, Monni C, Pezz´e M, et al., Localizing faults in cloud systems, 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2018, pp. 262-273.
[19] Wang L., Zhang C., Ding R., et al., Root cause analysis for microservice systems via hierarchical reinforcement learning from human feedback, Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2023, pp. 5116–5125.
[20] Zhou J, Cui G, Hu S, et al., Graph neural networks: A review of methods and applications, AI Open, 1 (2020), pp. 57-81.
[21] Kipf T N, Welling M., Semi-supervised classification with graph convolutional networks, arXiv preprint arXiv:1609.02907, 2016.