# Approach to Anomaly Detection in Microservice System with Multi-Source Data Streams

ZHANG Qixun[1], HAN Jing[2], CHENG Li[2], ZHANG Baisheng[2], GONG Zican[2]
(1. Peking University, Beijing 100091, China;
2. ZTE Corporation, Shenzhen 518057, China)

DOI: 10.12142/ZTECOM.202203011
https://kns.cnki.net/kcms/detail/34.1294.TN.20220728.1549.002.html, published online July 28, 2022
Manuscript received: 2022-01-24

**Abstract:** Microservices have become popular in enterprises because of their excellent scalability and timely update capabilities. However, while fine-grained modularity and service-orientation decrease the complexity of system development, the complexity of system operation and maintenance has been greatly increased, on the contrary. Multiple types of system failures occur frequently, and it is hard to detect and diagnose failures in time. Furthermore, microservices are updated frequently. Existing anomaly detection models depend on offline training and cannot adapt to the frequent updates of microservices. This paper proposes an anomaly detection approach for microservice systems with multi-source data streams. This approach realizes online model construction and online anomaly detection, and is capable of self-updating and self-adapting. Experimental results show that this approach can correctly identify 78.85% of faults of different types.

**Keywords:** anomaly detection; data stream; microservice; monitored indicator; system log

**Citation** (IEEE Format): Q. X. Zhang, J. Han, L. Cheng, et al., "Approach to anomaly detection in microservice system with multi-source data streams," *ZTE Communications*, vol. 20, no. 3, pp. 85–92, Sept. 2022. doi: 10.12142/ZTECOM.202203011.

## 1 Introduction

In recent years, the microservice architecture has been widely used in enterprises. Its core ideas are fine-grained module division, service-oriented interface encapsulation, and lightweight communication interaction. The architecture splits a tightly coupled application into several independent services that have their own functions and run in independent development and deployment processes. The services coordinate and cooperate with each other based on a lightweight communication mechanism. Compared with traditional software systems, microservice systems are characterized by finer granularity towards the division of service, more flexible expansion, more frequent program update iterations, etc. At the same time, in order to improve resource utilization, services are often deployed in a lightweight containerized manner. In the microservice system, besides the defects in an application itself, system failures may often be caused by configuration errors and resource contention problems. When a failure inside or outside the system is activated, it may cause errors and failures, which will further spread between services to produce a chain reaction, affecting the service performance or even making it impossible to run the service normally.

Existing microservice anomaly detection approaches often acquire the behavior features of the system through analyzing system runtime data such as monitored system indicator data or log data, identifying the abnormal behavior of the system, diagnosing the type of system failure, and locating the root cause of the failure. Some methods have key limitations and shortcomings. Firstly, these methods often use offline training and online detection methods, which are not efficient and cannot adapt to system updates or data changes, resulting in poor anomaly detection results. Secondly, the data are often output as a stream when the system is running. The existing methods usually apply batch processing for data analysis, which cannot adapt to the real-time characteristics of streaming data, leading to a high degree of lag in anomaly detection. Therefore, how to process and analyze these data streams and how to construct an online anomaly detection model have become important issues. This paper will focus on how to use both log data and monitored system indicator data for anomaly detection and simultaneously to improve the model's capabilities of self-updating and self-adapting for streaming data.

1) An anomaly detection method with multiple data streams

**This work was supported by ZTE Industry-University-Institute Cooperation Funds under Grant No. HF-CN-202008200001.**

is proposed. Based on the data flow of the runtime system, the microservice anomaly features in the data stream are mined, and online model construction and online anomaly detection are realized with the capability of self-updating and self-adapting.

2) A rule-based fault identification method is proposed, which can synthesize abnormal information online, filter noise and identify faults.

Experiments are conducted in Sock-Shop, an open source microservice application system, to verify the effectiveness of the method in this paper through fault injection. The experimental results show that the proposed method can identify different types of faults with a correctness of over 81%.

## 2 Related Work

Formerly, anomaly detection is mostly achieved by monitoring indicator data or learning the features of system behavior. The related work can be classified as anomaly detection approaches based on monitored indicators or based on system log analysis. The anomaly detection based on monitored indicators include approaches based on rules, statistical methods, or machine learning. Rule-based approaches usually define rules by analyzing historical data and expert experience, which helps to accurately detect anomalies that meet the rules. However, limited to the fixed rules, it requires the in-time updating of rules. Otherwise, an anomaly belonging to the new cluster would not be able to be detected. Statistical methods-based approaches assume the data obeys a certain distribution, and then use statistical data to estimate, which heavily relies on the assumption. The approaches based on machine learning are usually classified by supervised learning or unsupervised learning. Supervised learning uses plenty of sample data with labels to train a classifier. Unsupervised learning detects the anomaly using mathematical approaches such as distance, density and clustering. To detect anomalies on multiple dimension with causality, an indicator dependency graph can be depicted to discover the abnormal indicator. The graph-based approach generally consists of two steps, graph representation and abnormal indicator detection. Based on observed performance indicators, CloudRanger[1] uses the PC algorithm to construct an influence diagram, then uses Pearson Correlation Function to calculate the correlation between services, and finally uses Customized Second-Order Random Walk Heuristic Survey Algorithm in the influence diagram to detect anomalies. Through causal analysis, MS-Rank[2] extracts the impact diagram between services from various indicators, and then uses Customized Random Walk Algorithm in the impact diagram based on the confidence of service indicators to obtain the abnormal service level to achieve the result.

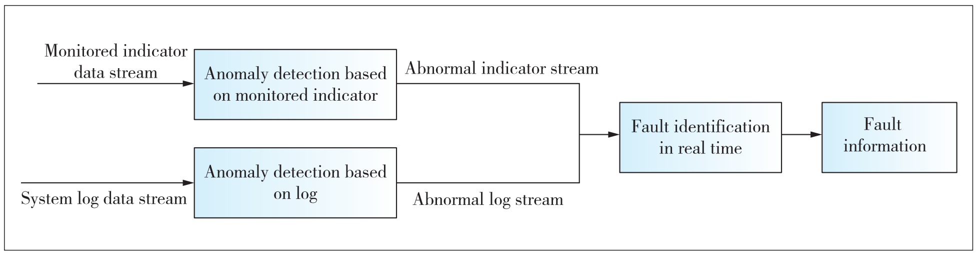
Log-based anomaly detection includes anomaly detection based on graph models, probability distributions, and machine learning[3]. A graph-based anomaly detection technique constructs a model for log sequence relationship, association relationship, and log text content. The anomaly detection based on the probability distribution calculates the correlation probability between the log and the anomaly. The approach based on machine learning is to extract the features of the log, and use machine learning algorithms such as clustering for feature correlation. CHUAH et al.[4] proposed a log diagnosis tool, which extracts log information through a structured template and calculates the similarity of the log to detect the anomaly log. CHEN et al.[5] proposed a log analysis approach that analyzes the trace log of a large-scale system. It aims to calculate and analyze the frequency of the log template in each time window. The time window in which the frequency suddenly changes is a fault window and the corresponding log is an anomaly log. ZHOU et al.[6] proposed an anomaly detection approach for microservice applications called Microservice Error Prediction and Fault Localization (MEPFL), which trains the model through supervised learning, uses the features and injected faults on the tracking log in the system as the training set, and then uses the model in the production environment to capture potential anomalies.

## 3 Anomaly Detection Based on Multi-Source Data Streams

This section provides a detailed description of the approach proposed in this paper. As mentioned earlier, this approach performs real-time analysis on the multi-source data streams to find anomalies and diagnose the root cause of indicators that characterize anomalies. This approach includes three key steps: the anomaly detection based on monitored indicators, anomaly detection based on system logs, and real-time fault identification (Fig. 1). Anomaly detection based on monitored indicators, which integrates multiple time series models, is responsible for analyzing monitored data streams. The model captures a variety of different features of the monitored indicators, finds abnormal points in the indicator data in an all-direction way, and outputs the abnormal indicator data stream. Log-based anomaly detection is responsible for analyzing the system log stream, constructing a real-time time-weighted control flow, finding different types of anomalies in the log, and outputting the abnormal log data stream. Online fault identification is responsible for integrating the abnormal indicator data stream and the abnormal log data stream, filtering the abnormal noise, and finally identifying the fault and providing feedback on the fault information in real time.

### 3.1 Anomaly Detection Based on Monitored Indicator Data Stream

Monitored indicators can be formalized as time series streams in the form of $\{x_t\}$, where $x$ is a specific indicator type and $t$ is the corresponding time for collecting. With the most recent $T$ indicator values (that is, selecting a sliding time window with length $T$), the anomaly detection problem of moni-


Monitored indicator data stream
Anomaly detection based on monitored indicator
Abnormal indicator stream
Fault identification in real time
Fault information
Anomaly detection based on log
System log data stream
Abnormal log stream

▲**Figure 1. Overview of the proposed approach**

tored indicators can be regarded as a historical time series $\{x_{t-T}, x_{t-T+1}, \cdots, x_{t-1}\}$ with length $T$ to determine whether the current indicator value $x_t$ is abnormal or not.

This paper proposes an anomaly diagnosis approach based on monitored indicators. Specifically, the kernel density estimation and weighted moving average approaches are selected, and the anomaly detection results obtained are quantified and normalized. The final anomaly score is obtained by integration and used for subsequent root cause diagnosis.

Kernel density estimation[7] is a non-parametric test approach, mainly used to estimate the unknown probability distribution of a sample. The probability density function based on the sample frequency is smoothed by the kernel density estimation to obtain the derivable density function. A major advantage of the kernel density estimation is that there is no need to make any assumptions about the distribution of the sample data. In the scenario of monitored indicator anomaly detection, we aim to establish the distribution function model of each indicator through the kernel density estimation of historical data. When a new monitored data point is received, the quantified degree of abnormality can be measured by using the idea of hypothesis testing and verifying the probability that the new data point conforms to the existing distribution function. Specifically, we can estimate a probability distribution $f_X(x)$ from the historical data.

$$f_X(x) = \frac{1}{T-1}\sum_{i=t-T}^{t-1} G(x; x_i). \tag{1}$$

Based on this distribution, we can use hypothesis testing to calculate the degree of anomaly parameter $p$ quantified by the latest indicator value $x_t$. This value will be used to calculate the overall degree of anomaly of the data point.

Another effective lightweight unsupervised time series predicting approach is the weighted moving average[8]. The main idea is to assign higher weights to the nodes that are closer to the current moment and perform a weighted average, thereby obtaining the predictive value of the current indicator.

$$\widehat{x_t} = \alpha x_{t-1} + \alpha(1-\alpha)x_{t-2} + \alpha(1-\alpha)^2 x_{t-3} + \cdots. \tag{2}$$

We use the difference between the predicted value and the real value at the current moment as an evaluation of the degree of the indicator's anomaly.

The overall degree of the indicator's anomaly (denoted as $A$) is the weighted integration of the above two statistical values. Before weighting, the above statistical values must be normalized in advance (mapped to the interval of 0 – 1). Then, statistical values are assigned with different weights to obtain the overall anomaly score.

$$A = \omega_1 p_{\text{value}} + \omega_2 \left| x_t - \widehat{x_t} \right|. \tag{3}$$

Indicators with an overall anomaly score higher than the threshold are considered anomaly indicators. These indicators will be performed with subsequent fault identification.

## 3.2 Anomaly Detection Based on Log Data Stream

This approach converts the log stream into a log template stream, uses a network inference algorithm to construct and update the control flow graph model in real time, and finally detects anomalies in real time based on the control flow graph model.

### 3.2.1 Time-Weighted Control Flow Graph

The time-weighted control flow graph (TCFG) is a directed graph composed of edges, nodes and time weights. The nodes represent log templates, the edges represent the transfer relationship between log templates, and the time-weight records the transfer time between log templates. The time-weight is calculated by the difference between the timestamps of two adjacent logs of the log sequence belonging to the same request.

The formalized definition of TCFG is as follows:

$$\text{TCFG} = (V, E, W), \tag{4}$$

where $V=\{v_1, v_2, \cdots, v_n\}$ represents the nodes (log templates) in the graph model, and the total number is n. $E=\{e_{ij}|1\leqslant i, j\leqslant n\}$ represents the edge from $v_i$ to $v_j$ in the graph model. $W=(w_{ij}|e_{ij}\in E)$ represents the time weight of each edge in the graph model.

The TCFG model describes the request execution logic of the healthy system and is the basis for fault diagnosis. When the system fails, the requested log sequence will show a difference from the TCFG model. For example, a request outputs an ERROR-level log that is not recorded in the TCFG model, which indicates the system has a fault and this fault-sensitive log is accurately located. Furthermore, a TCFG model can diagnose system request latency exceptions at a fine-grained level. When a request latency exception occurs in the system, the execution time between adjacent logs in the same request log sequence increases. By comparing with the time weight in the TCFG model, we can accurately locate the log with high latency where the request latency exception occurs, and even the program fragment.

### 3.2.2 Anomaly Detection Model Construction

In this paper, the log template mining algorithm, Drain[9], is used to convert the log stream into a log template stream $p$. The core idea is to use a transition probability function parameter $\alpha_{j,i}$ to model the transition probability-time distribution from template $j$ to template $i$. The transition probability function is formalized as $f\left(t_i|t_j, \alpha_{j,i}\right)$, representing the probabilities of log template $j$ to log template $i$ appearing at the time $t_j$ and $t_i$ respectively. Through the analysis of the real log data in this paper, a power law distribution is used to fit the function, which is

$$f\left(t_i|t_j,\alpha_{j,i}\right) = \begin{cases} \dfrac{\alpha_{j,i}}{\delta}\left(\dfrac{t_i - t_j}{\delta}\right)^{-1-\alpha_{j,i}} & \text{if } t_j + \delta < t_i \\ 0 & \text{otherwise,} \end{cases} \tag{5}$$

where $\delta$ represents the minimum transition time from template $j$ to template $i$. Based on this function, the occurrence probability of the entire log template stream is calculated. By adjusting the parameters to maximize the occurrence probability of the real log template stream, the log stream is fitted.

In the log template stream $p$, the occurrence probability of any log template $i$ at time $t_i$ is the sum of the transition probabilities of all previous log templates at time $(t_1,\cdots,t_N| t_k \leqslant t_i)$. For any log template transfer $j \rightarrow i$, the probability that the transfer does not occur is $S\left(t_i|t_k, \alpha_{k,i}\right)$ (non-transfer probability).

$$S\left(t_i|t_k,\alpha_{k,i}\right) = 1 - F\left(t_i|t_k,\alpha_{k,i}\right), \tag{6}$$

where $F\left(t_i|t_k,\alpha_{k,i}\right) = \int_{t_j}^{t_i} f\left(t|t_k,\alpha_{k,i}\right)\mathrm{d}t$. The transfer probability of the log template transfer $j \rightarrow i$ is multiplied by the transfer probability of $j \rightarrow i$ and non-transfer probability towards other log templates $k \rightarrow i$, where $k \in \{1,\cdots,N\}, k \neq j, t_k < t_i$ and $A = \{\alpha_{i,i}|i, j = 1,\cdots,N, i \neq j\}$.

$$f\left(t_i|t_j,A\right) = f\left(t_i|t_j,\alpha_{j,i}\right) \times \prod_{k:k\neq j,t_k<t_i} S\left(t_i|t_k,\alpha_{k,i}\right). \tag{7}$$

The occurrence probability of the entire log template stream $p$ is

$$f\left(t^{\leqslant T},A\right) = \prod_{t_i \leqslant T} f\left(t_i|t_1,\cdots,t_{N\backslash t_i},A\right), \tag{8}$$

which is

$$f\left(t^{\leqslant T},A\right) = \prod_{t_i\leqslant T}\left(\prod_{t_k<t_i} S\left(t_i|t_k,\alpha_{k,i}\right) \times \sum_{j:t_j<t_i} \frac{f\left(t_i|t_j,\alpha_{j,i}\right)}{S\left(t_i|t_j,\alpha_{j,i}\right)}\right). \tag{9}$$

More specific simplification steps can be found in Ref. [10].

Finally, the TCFG model construction is transformed into inferring the most likely graph structure so that the graph structure can fit the log template flow $\boldsymbol{p}$ with the greatest probability. Given a TCFG, the matrix composed of transition probability function parameters between any two log templates in the graph is $A$. The problem can be formalized as

$$\begin{aligned} &\text{maximize}_A && \log f\left(t,A\right) \\ &\text{subject to} && \alpha_{j,i} \geqslant 0, i,j = 1,\cdots,N, i \neq j, \end{aligned} \tag{10}$$

where $A = \left\{\alpha_{j,i}|I,j = 1,\cdots,N, i \neq j\right\}$.

This approach uses the random gradient descending for training. In each iteration during the training process, $A$ is updated. The updating calculation is as follows.

$$\alpha_{j,i}^{k}(t) = \left(\alpha_{j,i}^{k-1}(t) - \gamma\nabla_{\alpha_{j,i}} L_c\left(A^{k-1}(t)\right)\right)^{+}, \tag{11}$$

where $k$ is the number of iterations and $\nabla_{\alpha_{j,i}} L_c(\cdot)$ is the gradient of $L_c(\cdot)$. In each iteration, only the TCFG subgraph related to the log template that appears in the current time period is updated. Finally, if the transition probability of the two log templates is high enough, a corresponding edge is added to TCFG.

### 3.2.3 Anomaly Detection Based on TCFG

The anomaly detection based on the control flow graph identifies the difference between the control flow graph and the log sequence. There are three types of anomalies that serve as the basis for fault diagnosis, including sequence anomalies, redundancy anomalies, and latency anomalies. The sequence anomaly refers to any child node of the log template $T$ that does not appear in the log template sequence in the expected time window $t$ after $T$. The redundancy anomaly is defined as a new log template that has never appeared in the expected time window after $T$. The latency anomaly means that the time interval between $T$ and the most recent child node in the log template sequence is greater than that recorded in TCFG.

### 3.2.4 Real-Time Fault Identification

Our anomaly detection approach, which is based on metrics data and log data, outputs anomaly information in real time. However, due to data noise and the inference accuracy of the algorithm, not all anomalies are system failures. Therefore, a rule-based fault identification approach that combines characteristics including anomaly density and anomaly duration is proposed to determine system failures. The calculation function is expressed as follows

$$f\left(\text{density, time}\right) = \begin{cases} 1 \\ 0, \end{cases} \tag{12}$$

where 1 signifies a fault has occurred and 0 signifies no fault has occurred. The anomaly density refers to the number of anomalies output in real time based on monitored data and log data over a period of time. It has been verified in Ref. [11] that anomalies with a higher frequency are more likely to characterize a failure. Therefore, higher anomaly density leads to a greater possibility of system failure. The duration of an anomaly is an important factor in determining system faults. Generally, if there is no intervention from external factors, such as manual processing, critical faults will hardly become weaker or disappear over time. On the contrary, some system states often fluctuate instantaneously and these instantaneous fluctuations will produce anomalies that are not system failures.

For these characteristics, two parameter thresholds $\{\gamma, \varepsilon\}$ are set for fault determination. For anomaly density, the number of anomalies per minute is used as the determination parameter. When the parameter value exceeds the threshold, it is determined as a fault. The anomaly distribution is evaluated by the standard deviation of the number of anomalies per minute for each service. If the number of anomalies in the duration exceeds the threshold, it will be determined as a fault:

$$f(\text{density}, \text{distribution}, \text{time}) = \begin{cases} 1, \text{desity} > \gamma \vee \text{time} > \varepsilon \\ 0, \text{otherwise} \end{cases} . \tag{13}$$

### 3.2.5 Experiment Environment

In order to verify the effectiveness of the proposed approach, we built a microservice system based on Kubernetes as an experimental environment. The hardware platform used in the experiment is 2 Dell R740 Server, configured with 2 Intel Xeon Gold 5220R processors (2.2 GHz, 48 core, and 96 threads), 128 G physical memory, a 4 TB SSD hard disk, and a Gigabit Ethernet card. For each sever , we installed the Ubuntu LTS operating system, created a virtual machine through Kernel-Based Virtual Machine (KVM), and then built the Kubernetes cluster on the virtual machine. The cluster contains 2 master nodes and 3 worker nodes, with the Istio Service Grid System installed. We also deployed supportive software for analysis such as Jaeger, Kiali, Node-exporter, Filebeat, ELK (Elasticsearch, Logstash, and Kibana), Zabbix, and Prometheus for log and monitoring data collection in the Kubernetes cluster. The resource configuration information of the virtual machine used in the experimental environment is shown in Table 1.

We selected the open source microservice application system Sock-Shop as the experimental object. Sock-Shop is an electronic business system that simulates selling socks. The development environment includes Java, Golang and NodeJS. The system is divided into eight application services, including Front-end (user interaction interface), Users (user registration and login), Catalogue (product classification), Carts (shopping cart), Orders (submitting orders), Queue Master (processing order queue), Payment (payment), and Shipping (delivery), besides the database service MongoDB and message middleware service RabbitMQ. Each service mainly communicates and interacts using the HTTP protocol. Thus, the coupling between services is low and the development and deployment are convenient. Sock-Shop has been widely used in Refs. [12 – 13] as a typical representation.

The resource configuration of each application container in the microservice application system Sock-Shop deployed in the cluster is set according to the official reference. The specific configuration information is listed in Table 2.

### 3.2.6 Fault Injection

In an actual production environment, the failure probability of a running system is extremely low, and the failures are often uncertain. A common approach is to inject specified types of faults into the system to verify its ability of the microservice system to handle failures and observe the operating status of the system. We used a series of tools (Stress-ng, traffic control, etc.) to inject faults into the Sock-Shop system and a load testing tool (Locust) to simulate multiple users sending a series of requests to the system at the same time, real user login, query, order and other operations, and collected logs, metrics and service KPI data generated during the running period.

By investigating the faults that often occur in the microservice system, two representative faults are identified: application faults and system resource faults. The fault description is shown in Table 3.

**▼Table 1. Virtual machine resource configuration**

| Virtual Machine Number | Virtual Machine Function | Resource Configuration | Virtual Machine Location |
|---|---|---|---|
| 1 | Master 1 | 8 Core CPU, 16 G Memory, 200 G Disk Space | Server_1 |
| 2 | Master 2 | 8 Core CPU, 32 G Memory, 200 G Disk Space | Server_2 |
| 3 | Worker 1 | 8 Core CPU, 32 G Memory, 200 G Disk Space | Server_1 |
| 4 | Worker 2 | 8 Core CPU, 16 G Memory, 200 G Disk Space | Server_1 |
| 5 | Worker 3 | 4 Core CPU, 16 G Memory, 200 G Disk Space | Server_2 |
| 6 | ELK | 8 Core CPU, 32 G Memory, 1 T Disk Space | Server_2 |
| 7 | Others | 4 Core CPU, 8 G Memory, 200 G Disk Space | Server_2 |

ELK: Elasticsearch, Logstash, and Kibana

**▼Table 2. Container resource configuration**

| | Front-End | User | Catalogue | Carts | Orders | Queue-Master | Payment | Shipping |
|---|---|---|---|---|---|---|---|---|
| CPU/m | 300 | 300 | 200 | 300 | 500 | 300 | 200 | 300 |
| Memory/Mi | 1000 | 200 | 200 | 500 | 500 | 500 | 200 | 500 |

▼Table 3. Two representative fault types in microservice system

| Fault Type | Fault Description |
|---|---|
| Application fault | Caused by null value errors in the code, short-circuit of exception statements, condition reversal, default values missing in switch statements, etc. |
| System resource fault | Caused by high node CPU utilization, insufficient memory, network delay or packet loss, disk I/O blocking, etc. |

### 3.2.7 Application Faults

Application faults mainly refer to software bugs introduced during the software development process by developers, such as the direct use of uninitialized objects in the code and the incorrect boundary of a conditional statement. When the bugs within the application are activated during the system running, exceptions or even service failures might appear. For application faults, we directly modify the application source code, inject faults into the source code, and trigger them by sending a request to the microservice system. The application faults involve the null value, unexpected value, short-circuit in the exception statement, condition reversed, switch statement lacking a default value, exception uncaught, requested memory unreleased, and middleware upgrade.

• Null value: When the program encounters an uninitialized object during the running period, the error log will be printed if there is a null value judgment statement block; an exception error will be thrown if there is no null value judgment. Therefore, abnormal characteristics will appear in the application log. The corresponding service outputs a failing request.

• Unexpected value: The variable value in the process of a program is of the wrong type. If there is a corresponding type judgment statement block, the error log will be printed; otherwise, an abnormal error will be thrown. Therefore, the abnormal characteristics will be shown in the application log. The corresponding service outputs a failing request.

• Short-circuit in the exception statement: The exception statement in the program is directly triggered and the corresponding exception is thrown. If it is not caught, there will be an error output in the log. If the exception is caught, the program logic will change. The service outputs a request failure or an incorrectly return result.

• Condition reversal: The judgment condition of the conditional judgment statement used in the program running process is reversed. In some cases, it will cause the wrong variable value or even an exception thrown directly. The service outputs a request failure.

• Switch statement lacking a default value: The switch statement used in the running program lacks the default branch and the existing branch cannot cover the current situation. In some cases, it will cause variable value initialization errors, null value errors, etc. The service outputs a request failure.

• Exception uncaught: An undeclared exception is thrown when the program is running, and there is no corresponding capturing and processing statement block in the code. The service outputs a request failure.

• Requested memory unreleased: When the requested memory resources fail to release in the program code, a memory leak occurs. When the memory occupation reaches its upper limit, the process will be killed and the pod restarted. The service will output time-outs or request failures.

• Middleware upgrade: The upgrade of middleware which the application is relied on causes compatibility issues. This further causes system failures or request failures.

### 3.2.8 System Resource Faults

System resource faults refer to system faults in the actual production environment due to resource contention or incorrect configurations. When this type of fault appears, service response time becomes longer, leading to service instability or unavailability. We use third-party tools to simulate service resource faults in the experiment. To simulate CPU, memory and disk I/O exceptions, we use the open-source tool stress-ng under the Linux operating system to seize the system's CPU, memory, and disk I/O resources. For network anomalies, we use the traffic control command in the Linux system to control network traffic to simulate network delays and network packet loss. The system resource faults include the high node CPU load, high container CPU load, insufficient node memory, insufficient container memory, node disk I/O obstructed, and network delay in the node/packet loss.

• High node CPU load: As other pods on the node seize CPU resources, resource competition is caused, and the response time of some services is affected. Faults can be found through the memory resource monitored data on the node.

• High container CPU load: If the deployment is configured improperly, the container memory is insufficient and the service cannot run. As the dynamic expansion strategy is not set, CPU resource load is too high under high concurrent requests. The service request response is therefore abnormal or the service request fails.

• Insufficient node memory: Due to insufficient node memory, the node fails and all services on the node are unavailable.

• Insufficient container memory: Pod start-up failure or continuous restart of pod under high load occurs due to insufficient memory resource allocation. The service will be unavailable or unstable.

• Node disk I/O obstructed: Due to a large number of disk I/O requests from other pods on the node, disk read and write competition occurs, which leads to a prolonged service request time.

• Network delay in the node/package loss: Packet loss or network latency occurs on the network due to switch failure or server network card failure, which affects the request time of the service on the node.

### 3.3 Analysis of Anomaly Detection Results

We conducted a large number of random fault injection experiments on the system. The injected applications include the front-end (user interaction interface), users (user registration and login), catalogue (product classification), carts (shopping carts), orders (order submit), payment (payment) and shipping (delivery). At least 20 successful activation cases were randomly selected for each failure, and indicator data, log data and service KPI data were collected to serve as the experimental data set. Recall was used to evaluate the fault diagnosis results of injected faults. The calculation of recall is shown in Eq. (14), where TP is the number of correctly identified faults and FN is the number of unrecognized faults.

$$\text{Recall} = \frac{TP}{TP + FN} . \tag{14}$$

The anomaly detection approach based on monitored indicators and the anomaly detection approach based on logs proposed in this paper were individually performed on the experimental data set to detect the anomalies and identify the faults. The results are shown in Table 4.

▼Table 4. Recall rate of anomaly detection and fault recognition (recall)

| | Indicator-Based Anomaly Detection | Log-Based Anomaly Detection | Fault Identification |
|---|---|---|---|
| Null value | 1.00 | 1.00 | 1.00 |
| Unexpected value | 1.00 | 1.00 | 1.00 |
| Short circuit of the exception statement | 0.35 | 0.45 | 0.45 |
| Condition reversed | 0.10 | 0.55 | 0.55 |
| Switch statement lacking a default value | 0.15 | 0.75 | 0.45 |
| Exception uncaught | 1.00 | 1.00 | 1.00 |
| Requested memory unreleased | 1.00 | 1.00 | 1.00 |
| High node CPU load | 1.00 | 0.00 | 0.70 |
| High container CPU load | 1.00 | 0.00 | 0.75 |
| Insufficient node memory | 1.00 | 1.00 | 1.00 |
| Insufficient container memory | 1.00 | 1.00 | 1.00 |
| Node disk I/O obstructed | 0.65 | 0.00 | 0.65 |
| Network delay in the node/package loss | 0.90 | 0.00 | 0.70 |

According to the experimental results in Table 4, the method proposed in this paper can correctly identify about 78.85% of the fault types. Although the proposed method can accurately detect and locate most anomalies, there are still some faults that cannot be detected effectively. Through manual analysis, the key reasons include:

1) The sampling time interval of indicator monitoring data is too long and some instantaneous peak data cannot be collected, thus some faults do not output abnormal values and cannot be detected by the algorithm.

2) Some detected anomalies are false alarms, because noises are hidden in system running data.

3) There are many kinds of application logic faults. These faults only output abnormal values of calculation results, however, the degree of anomaly in the running data is not obvious. Therefore, the algorithm proposed in this paper cannot solve these problems.

## 4 Conclusions

This paper proposes an anomaly detection method based on streaming runtime data for microservices. First, an anomaly detection method based on monitored indicators is used to analyze the streaming system running data monitored. By analyzing a variety of different features of the monitored indicators, the abnormal points in the indicator data are found. Then the log-based anomaly detection method is used to analyze the system log stream, and the time-weighted control flow graph is built online to detect anomalies in log data. Finally, the results of the previous two anomaly detection methods are integrated and a filtering method is applied to these results to output the final anomalies.

We simulated a microservice system based on Kubernetes as our lab environment. Fault injection is utilized to simulate multiple faults including system changes, applications faults and system resources faults. Logs, monitored indicators, and service PKI data are collected as datasets for evaluation. The experimental results show that the proposed method can identify different types of faults with over 78% accuracy. In the future, we consider to design more sophisticated models to capture features of multi-source data such as logs, monitoring data, KPI and tracing data.

## References

[1] WANG P, XU J M, MA M, et al. CloudRanger: root cause identification for cloud native systems [C]//18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). IEEE, 2018: 492 – 502. DOI: 10.1109/CCGRID.2018.00076

[2] MA M, LIN W L, PAN D S, et al. MS-rank: multi-metric and self-adaptive root cause diagnosis for microservice applications [C]//IEEE International Conference on Web Services. IEEE, 2019: 60 – 67. DOI: 10.1109/ICWS.2019.00022

[3] JIA T, LI Y, WU Z H. Survey of state-of-the-art log-based failure diagnosis (in Chinese) [J]. Journal of software, 2020, 31(7): 1997 – 2018. DOI: 10.13328/j.cnki.jos.006045

[4] CHUAH E, KUO S H, HIEW P, et al. Diagnosing the root-causes of failures from cluster log files [C]//International Conference on High Performance Computing. IEEE, 2010: 1 – 10. DOI: 10.1109/HIPC.2010.5713159

[5] CHEN C, SINGH N, YAJNIK S. Log analytics for dependable enterprise telephony [C]//Ninth European Dependable Computing Conference. IEEE, 2012: 94 – 101. DOI: 10.1109/EDCC.2012.14

[6] ZHOU X, PENG X, XIE T, et al. Latent error prediction and fault localization

for microservice applications by learning from system trace logs [C]//27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2019: 683 – 694

[7] SILVERMAN B. Density estimation for statistics and data analysis [M]. London, UK: Routledge, 2018

[8] PARZEN E, BROWN R G. Smoothing, forecasting and prediction of discrete time series [J]. Journal of the American statistical association, 1964, 59(307): 973. DOI: 10.2307/2283122

[9] HE P J, ZHU J M, ZHENG Z B, et al. Drain: an online log parsing approach with fixed depth tree [C]//IEEE International Conference on Web Services. IEEE, 2017: 33 – 40. DOI: 10.1109/ICWS.2017.13

[10] JIA T, WU Y F, HOU C J, et al. LogFlash: real-time streaming anomaly detection and diagnosis from system logs for large-scale software systems [C]//IEEE 32nd International Symposium on Software Reliability Engineering. IEEE, 2021: 80 – 90. DOI: 10.1109/ISSRE52982.2021.00021

[11] ZHAO N W, CHEN J J, WANG Z, et al. Real-time incident prediction for online service systems [C]//28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2020: 315 – 326. DOI: 10.1145/3368089.3409672

[12] WU L, TORDSSON J, BOGATINOVSKI J, et al. MicroDiag: fine-grained performance diagnosis for microservice systems [C]//IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence). IEEE, 2021: 31 – 36. DOI: 10.1109/CloudIntelligence52565.2021.00015

[13] YANG Y, LI Y, WU Z H. Survey of state-of-the-art distributed tracing technology (in Chinese) [J]. Journal of software, 2020, 31(7): 2019 – 2039

## Biographies

**ZHANG Qixun** is currently an assistant professor in School of Software and Microelectronics in Peking University, China. He received his PhD in 2022. His research interests include distributed systems, AIOps, etc.

**HAN Jing** (han.jing28@zte.com.cn) joined ZTE Corporation in 2000. She is an expert in AIOps. She has been putting effort into natural language process for over 10 years and has published several papers.

**CHENG Li** joined ZTE Corporation in 2006. He is an expert in AIOps and wireless communications. He has much experience in analyzing variety of types of data. He has a lot of experience in problem-solving and methodology.

**ZHANG Baisheng** joined ZTE Corporation in 2011. His work has been devoted to cell-phone terminal techniques for over 10 years. Besides, he is interested in the research of auto-driving technology.

**GONG Zican** joined ZTE Corporation in 2020. He received his master's degree in computing from Australian National University, Australia in 2019. His research interests include AIOps and natural language processing.