Engineering Applications of Artificial Intelligence 133 (2024) 108558

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

An effective failure detection method for microservice-based systems using distributed tracing data

Zahra Purfallah Mazraemolla, Abbas Rasoolzadegan *

Software Quality Laboratory, Computer Engineering Department, Ferdowsi University of Mashhad, Mashhad, Iran

ARTICLE INFO

Keywords:
Microservice architecture
Failure detection
Distributed tracing
Sequence-to-sequence multiclass classification

ABSTRACT

Microservice architecture is the latest trend in the design and development of software systems based on modularization. In a microservice-based system, each microservice acts as an independent system; whereas, processing a request may require communication between two or more microservices. Therefore, increasing the complexity and size of microservice-based systems and increasing the communication between microservices increase the probability of runtime failures in such systems. On the other hand, failure detection in microservice-based systems faces challenges due to complex communications, frequent updates, dynamicity at runtime, and complex log management. In a microservice-based system, distributed tracing data is used to overcome the challenge of complex communication and dynamicity at runtime. Distributed tracing data can be used to obtain call paths between microservices when responding to a request. The purpose of this study is to improve the performance of the process of failure detection in terms of increasing accuracy and reducing the false-positive rate to detect failure in the microservices' call paths using distributed tracing data. For this purpose, we selected a basic method among the existing failure detection methods, and according to the basic method, we designed a process in three phases. The process phases of the proposed method are 1) training and evaluating the failure detection model, 2) online failure detection, and 3) updating the failure detection model. The evaluation results of the proposed method show that with 98% Accuracy and a false-positive rate of 0.02, we achieved a significant improvement over the basic method.

1. Introduction

Microservice architecture provides the facility of decomposing an application into a set of small autonomous services within the scope of the business domain (Jamshidi et al., 2018). In a microservice-based system, each microservice has an independent and specific task to satisfy a functional or non-functional requirement (Thones, 2015). So, using a microservice-based system has the following advantages (Jamshidi et al., 2018; Balalaie et al., 2016).

- The possibility of deploying each microservice on different platforms independently
- The possibility of developing each microservice by independent teams
- The possibility of using different programming languages, libraries, and technologies in each microservice
- Increasing the speed and ease of development and adding new features to the system

However, the growth of the complexity, size, and communication between microservices in microservice-based systems increases the probability of runtime failures in such systems. So, a microservice may need to communicate with many other microservices to process a request. Therefore, failure detection in microservice-based systems manually is a difficult and time-consuming task (Brandón et al., 2020). Therefore, the characteristics of microservice-based systems should be considered to detect runtime failures. The characteristics of microservice-based systems that caused challenges in detecting failures are as follows:

Complex communications: The communication between microservices in microservice-based systems is much more complex than in monolithic systems due to the large number of microservices, and a failure in a microservice can widely propagate to related services (Wu et al., 2020; Nedelkoski et al., 2019a; Yu et al., 2021; Scheinert et al., 2021). Therefore, it is difficult to manually determine the microservices that caused the failure (Wu et al., 2020; Kim et al., 2013; Liu et al., 2020; Gan et al., 2019; Meng et al., 2021; Wang et al., 2018). Also, the call

* Corresponding author.
E-mail address: rasoolzadegan@um.ac.ir (A. Rasoolzadegan).

https://doi.org/10.1016/j.engappai.2024.108558
Received 18 February 2023; Received in revised form 28 March 2024; Accepted 2 May 2024
Available online 5 June 2024

paths of microservices are complex due to the large number of microservices and the complex communication between them. So, analyzing the call paths of microservices manually is a time-consuming task (Yu et al., 2021).

Frequent updates: Microservices are frequently updated by the development team. Frequent updates in microservice-based systems cause new features or new call paths between microservices to be added to the system (Yu et al., 2021). For example, according to a report published in 2019, millions of changes have been made in the code and configuration of the eBay ecosystem; so understanding these changes based on the design documentation or business scope is difficult (Guo et al., 2020).

Dynamicity at runtime: One of the dynamicity aspects of microservice-based systems is differences in the call paths of microservices depending on the users' requests (Kim et al., 2013) and the call paths of microservices may have different lengths for different requests (Wang et al., 2018; Liu et al., 2021; Mariani et al., 2018; Nedelkoski et al., 2019b). Therefore, the failure detection method should consider the differences in the microservices' call paths in response to different requests.

Complex log management: The internal runtime errors of each microservice in a microservice-based system are recorded in log files. The log structure is usually determined by the developer and is used to detect and fix the runtime errors of each microservice. Log management in microservice-based systems is more complex than in monolithic systems. In microservice-based systems, multiple independent services are running and communicating with each other to process requests and generate their own logs along the execution. So, there is a need to use a distributed logging infrastructure that can capture logs from multiple microservices, aggregate and centralize them, and then process logs to detect runtime errors. Also, each microservice in a microservice-based system can developed by an independent team and have different programming languages and runtime environments. This means that the various microservices in a microservice-based system may be using different logging frameworks or log formats. A lack of consistent logging formats between microservices makes it more complex, costly, and time-consuming to parse, normalize, and process the log data (Cinque et al., 2022).

According to the mentioned issues, there is a need for a failure detection method in microservice-based systems to overcome challenges resulting from the characteristics of such systems. In microservice-based systems, distributed tracing data is used to overcome the challenge of complex communication between microservices and dynamicity at runtime for online failure detection. Using distributed tracing data provides the possibility of understanding the system architecture and the relationship between microservices despite a large number of microservices and the complex communication between them without the need for domain knowledge and a system graph, and it reflects the way of communication between microservices to process different user requests. It is possible to detect the failure in the call path of microservices by using distributed tracing data. On the other hand, due to the dynamicity of the communication between microservices depending on the user's request, the call path of microservices may change (Kim et al., 2013) and the call paths of microservices for different requests may have different lengths that should not detect as failures; so, using distributed tracing data have some challenges. So that the review of the available studies, especially the studies based on the distributed tracing data analysis shows the presented methods still do not have the acceptable performance to detect the failure in microservice-based systems. The performance of a failure detection method is evaluated by two metrics: 1) Accuracy: a successful failure detection method should be able to detect failures with high accuracy and low false-positive rate, 2) Speed: a successful failure detection method should be able to detect the failure online and fast. Also, to face the challenge of complex log management, after detecting the failure using distributed tracing data; we can report to the development team the failed microservices. Then, the development team can manually check the cause of the failure and fix it using logs. So, the overhead of log processing will be reduced.

Therefore, the purpose of this study is to improve the process of failure detection in microservice-based systems using distributed tracing data in terms of increasing accuracy and reducing the false-positive rate. Where false-positive refers to the normal tracing data samples that are incorrectly detected as failures. Using distributed tracing data, it is possible to detect the failure in the call paths of microservices. To improve the failure detection process, we are trying to overcome the challenge of the differences in the call paths of microservices depending on the users' requests. Also, due to the frequent updates of microservice-based systems, new call paths may arise; so, in the proposed method, we are trying to update the model of the system's behavior to maintain the accuracy of the failure detection model.

Since, distributed tracing data consist of sequences of events, Recurrent Neural Networks such as LSTM, Bi-LSTM, and GRU enable us to model distributed tracing data patterns in the execution environment. RNNs have a wide range of applications across various fields due to their ability to model sequential and temporal data such as stock market prediction, machine translation, and text generation. A Recurrent Neural Network is an artificial neural network where the output from the previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other. Still, in cases when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. RNNs solved this issue using a Hidden Layer. The main and most important feature of RNNs is the hidden state, which remembers important information in the sequences (Skrobek et al., 2022; Pirani et al., 2022). The key intuition behind the design of the proposed method is derived from natural language processing. So, we consider the sequence of events in distributed tracing data like a sequence of words in the sentences, and model distributed tracing data patterns using RNNs.

For this purpose, among the available failure detection methods, we selected a method as the basic method (Nedelkoski et al., 2019b) and designed a process in three phases according to the basic method. The process phases of the proposed method are 1) training and evaluating the failure detection model, 2) online failure detection, and 3) updating the failure detection model. In the training and evaluating phase of the failure detection model, we model the failure detection problem as a sequence-to-sequence multiclass classification problem and use the tracing data set obtained from the normal execution of the system as a training data set. In the second phase, we detect the failure in new tracing data samples using the trained model. Finally, the third phase is performed intermittently to update the failure detection model to cover new or less frequent call paths using the new tracing data set. We used the VWR benchmark (Chen et al., 2020) to evaluate the proposed method. Using the VWR benchmark, tracing data from the execution of a microservice-based system can be obtained. The evaluation results of the proposed method show that with 98% Accuracy, 90% Precision, 100% Recall, 98% Specificity, 95% F-score, 99% G-mean, 99% Balance, MCC of 0.94, a false-negative rate of 0, and a false-positive rate of 0.02, we achieved a significant improvement over the basic method.

The rest of the paper is organized as follows: In section 2, some basic definitions for the concepts used in this study are introduced. In section 3, the existing literature on failure detection in microservice-based systems has been reviewed. In section 4, the proposed method is described in more detail. In section 5, the evaluation has been discussed. Ultimately, section 6 focuses on the conclusion and presents suggestions for future research in this area.

2. Background

In this section, we introduce some basic definitions for the concepts used in this study as follows:

Distributed tracing data: A microservice-based system consists of a large number of autonomous microservices. In the execution

environment, each microservice has one or more instances and each instance runs on a physical or virtual host. Considering that in a microservice-based system to respond to a request, a set of microservices communicate with each other; distributed tracing data is used to understand the call path of microservices while processing a request (Liu et al., 2020; Gan et al., 2019; Meng et al., 2021).

In distributed tracing data, a unique ID is assigned to each request and passed to each of the microservices involved in the request processing. Each microservice uses this ID to report errors and the timestamps of receiving and completing the processing of the desired request. In this way, it is possible to trace the call path of microservices to process each of the requests independently (Meng et al., 2021; Zhou et al., 2019).

Each tracing data sample contains a set of events; So that each event shows a call between two microservice instances as caller → callee (Zhou et al., 2019). Each event contains information such as the type of request, the timestamps of receipt and completion of request processing, and the name of the microservice. Also, each event includes an identifier called "traceID", which indicates the desired event belongs to which tracing data sample (Yu et al., 2021). In Fig. 1, the relationship between microservices, tracing data samples, and events is shown in the form of an entity diagram.

An example of the sequence of events in a distributed tracing data sample for processing a user's request to add a product to the shopping cart is shown in Fig. 2. Each event includes information such as the microservice's name, operation type, and timestamp. Also, the "Parent Event" identifier indicates the parent-child relationship of the events.

Failure in microservice-based systems: In general, failure is the inability of a system to perform its required function due to unhandled errors. In microservice-based systems, failure may occur when one or more services fail or lead to degradation in function, or cascading errors cause the entire system to become unavailable (Soldani et al., 2022). The microservice-based systems are composed of a large number of microservices and each microservice acts as an independent system so that processing a request may require communication between two or more microservices. So, it is possible to detect failures in the functionality of microservices such as the failure down of one or more microservices using call paths of microservices while processing a request. A call path may be faulty from two perspectives: 1) The microservices' call sequence in the call path may be incorrect, or 2) The response time of a microservice in a call path may be out of expectation (Liu et al., 2020; Meng et al., 2021). Distributed tracing data illustrate microservices' call paths and can be used to detect failure in microservices' call paths. In this way using distributed tracing data, it is possible to get a general overview of the functionality of a microservice-based system during the execution and to detect failed microservices.

Long Short-Term Memory neural network: Long Short-Term Memory (LSTM) is a type of recurrent neural network capable of learning the order of sequences in sequence prediction problems and has solved the problem of long-term dependency in recurrent neural networks. LSTM tries to estimate the probability of each event next to a set of events according to the sequences in the training data set. In other words, it obtains the probability distribution of each event next to the sequence of events as pr (e_t |e_1, …, e_{t-2}, e_{t-1}) (Du et al., 2017; Hochreiter et al., 1997).

The structure of the LSTM network is shown in Fig. 3. So that the input of the network is a sequence of events and its output is a sequence of events detected by the network according to the input.

Bidirectional Long Short-Term Memory neural network: Bidirectional Long Short-Term Memory (Bi-LSTM) consists of two layers of LSTM in two opposite directions. So, the forward LSTM layer processes the sequence of events from the beginning to the end and obtains the probability of each event according to its previous events. Meanwhile, the backward LSTM layer processes the sequence of events from the end to the beginning and obtains the probability of each event according to its next events. Therefore, the probability of each event in the sequence

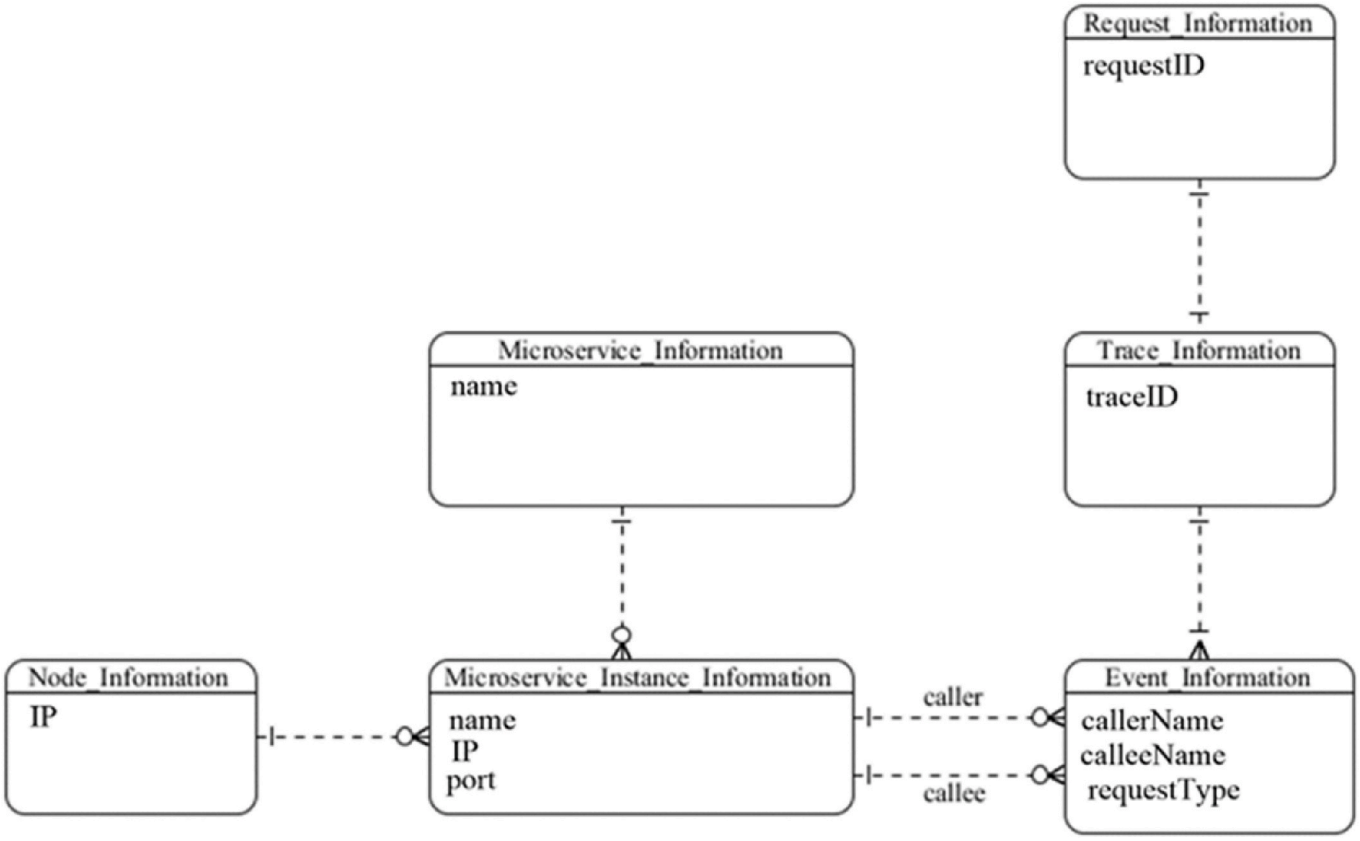

Fig. 1. Entity diagram of distributed tracing data sample and its relationship with system components (Zhou et al., 2019; Li et al., 2021).

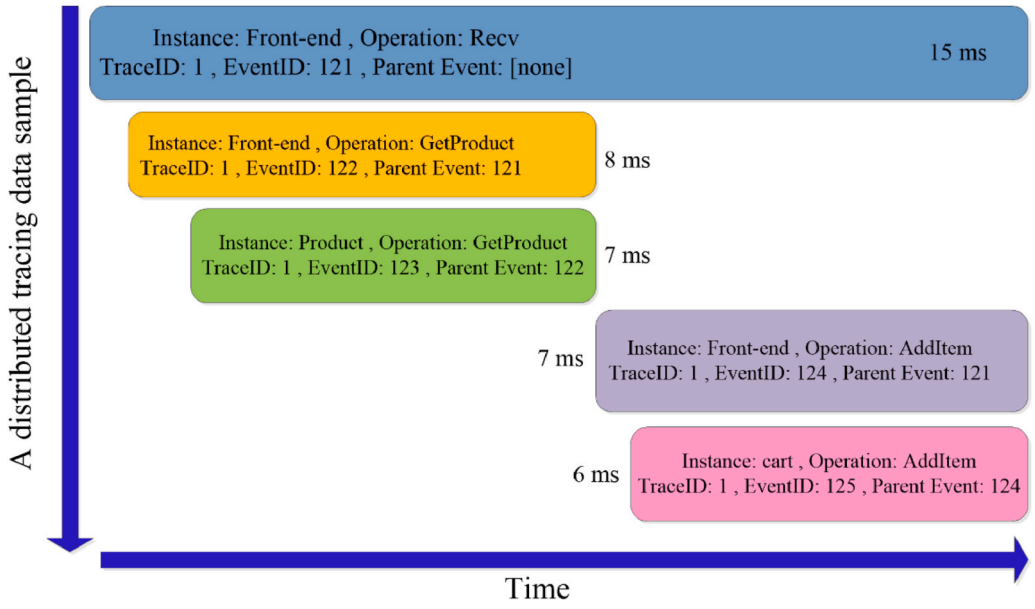

Fig. 2. A distributed tracing data sample to add a product to the shopping cart (Yu et al., 2021).

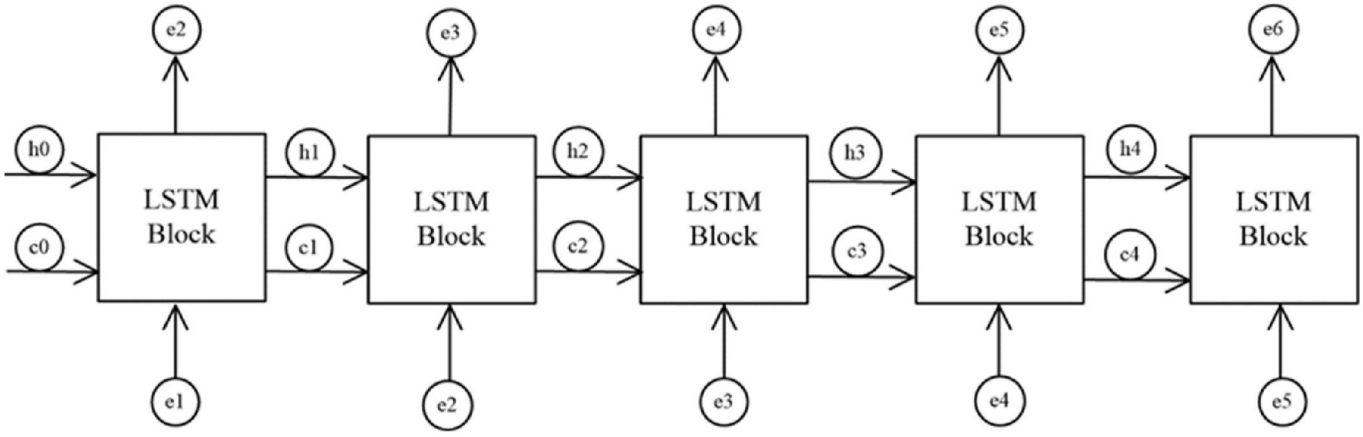

Fig. 3. The structure of the LSTM network.

is estimated depending on its previous and next events (Schuster et al., 1997). The structure of the Bi-LSTM network is shown in Fig. 4. So that the input of the network is a sequence of events and its output is a sequence of events detected by the network according to the input.

Gated Recurrent Unit neural network: Gated Recurrent Unit (GRU) is a type of recurrent neural network that was introduced in 2014 as a simpler alternative to the LSTM network. The GRU is like an LSTM with a forget gate, but has fewer parameters than LSTM, as it lacks an output gate. So, the trained model using GRU is simpler and faster than LSTM. GRU's performance on sequential data processing such as text, speech, and time-series data was found to be similar to LSTM (Chung et al., 2014). The structure of the GRU network is shown in Fig. 5. So that the input of the network is a sequence of events and its output is a sequence of events detected by the network according to the input.

Dropout: Dropout is a regularization technique for reducing overfitting in neural networks by preventing complex co-adaptations on training data. It is an efficient way of performing model averaging with neural networks. Dropout refers to randomly "dropping out" or omitting units during the training process of a neural network. So that during training, some layer outputs are ignored or dropped at random. This makes the layer regarded as having a different number of nodes and connectedness to the preceding layer. This prevents the network from relying too much on single neurons and forces all neurons to learn to generalize better (Srivastava et al., 2014).

Cross-Entropy loss function: Cross-entropy loss measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverge from the actual label. Cross-entropy loss decreases as the predicted probability is close to the actual label. Therefore, during training, the neural network tries to update the weights to reduce the value of the loss function in each iteration. For a multi-class classification, the value of the loss function of the model is equal to the sum of the value of the loss function for each observed value and each class, which is obtained through formula 1:

$$Cross - entropy\ loss = -\sum_{c=1}^{M} y_{o,c} \log\left(p_{o,c}\right) \tag{1}$$

In formula 1, M is equal to the number of classes and y is a binary value. So that if the detected value by the model is equal to the observed value, the value of y is one, and otherwise the value of y is zero. P is the probability calculated by the model that determines how much an observed value belongs to a class (Wang et al., 2022).

Softmax function: In multi-class classifications, the softmax

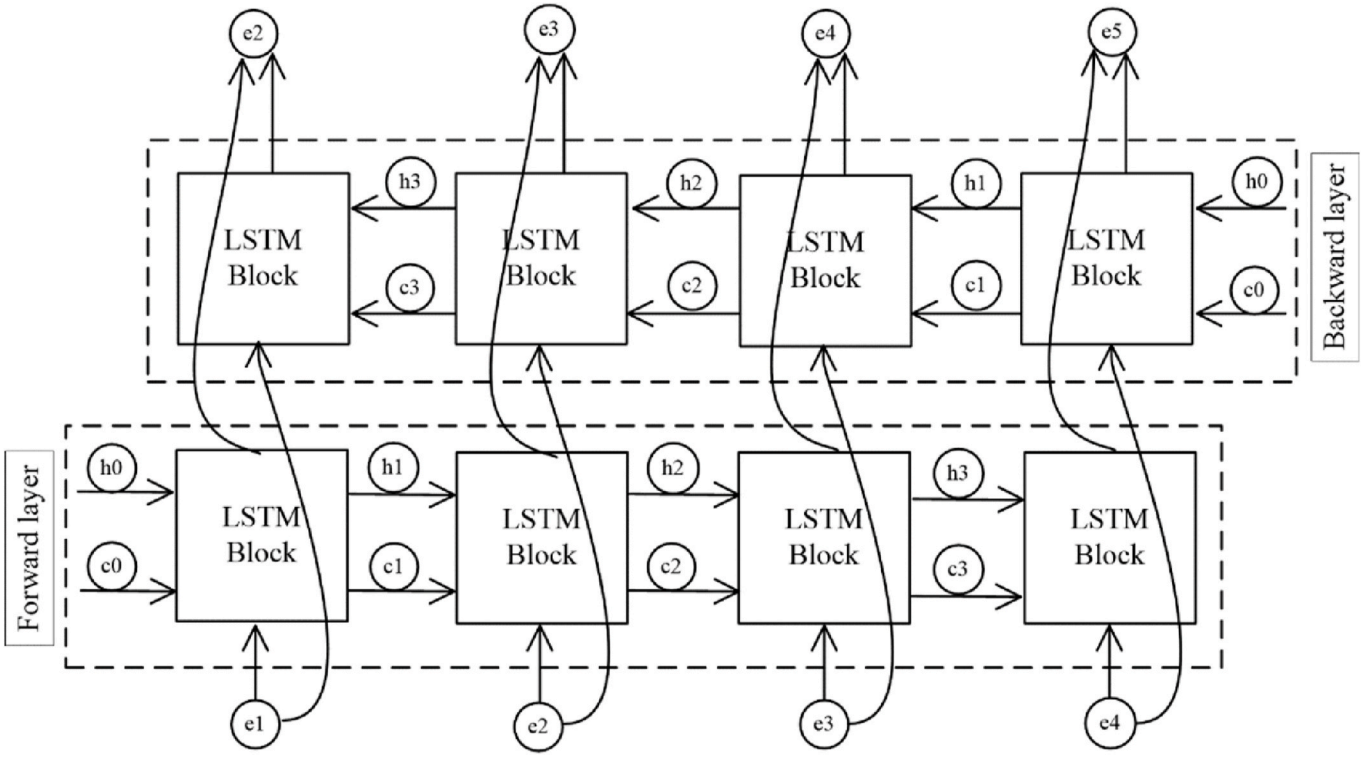

Fig. 4. The structure of the Bi-LSTM network.

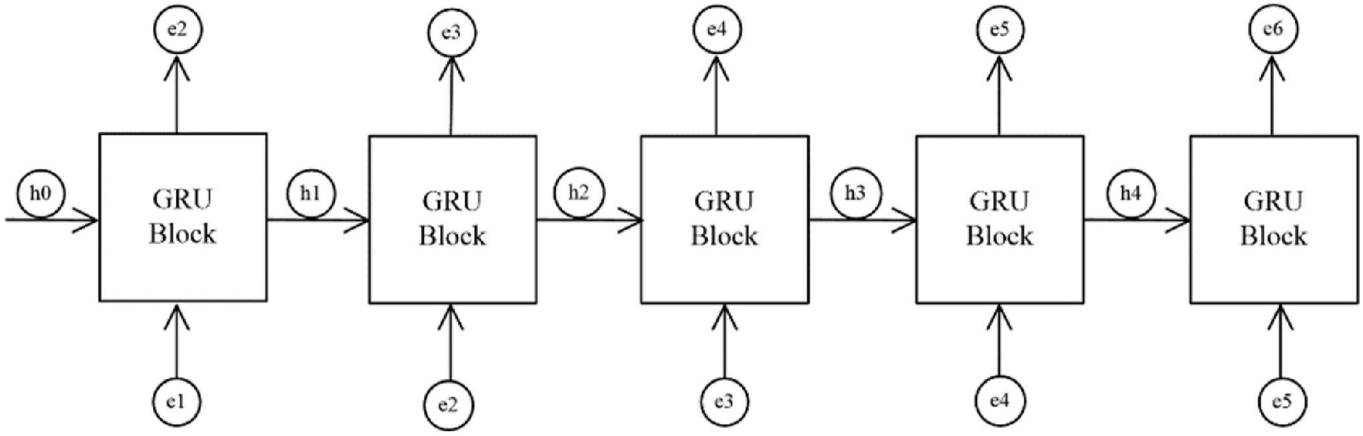

Fig. 5. The structure of the GRU network.

function is used as an activator in the output layer. The output of the softmax function is the probability distribution of different classes. The softmax function converts a vector of real values into a vector of values between zero and one. Therefore, the output of the softmax function can be used as a probability distribution. The softmax function is given in formula 2.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \tag{2}$$

In formula 2, Z is the input vector of the softmax function and each z_i can take a real value. The exponential function transforms the input vector into a positive value. Finally, the expression inside the denominator is the normalization expression and guarantees that all the output values of the softmax function take a value between 0 and 1 and the sum of all of them is one. Therefore, the output of the softmax function can be considered a probability distribution. The value of k is equal to the number of classification classes (Banerjee et al., 2020).

3. Related work

In general, the failure detection methods in microservice-based systems based on the data used to detect failures are divided into three approaches: 1) based on distributed tracing data analysis, 2) based on log analysis, and 3) based on monitoring metrics analysis (Soldani et al., 2022). Considering that, the purpose of this study is to improve the process of failure detection in microservice-based systems using distributed tracing data. In the following, we review the studies based on distributed tracing data analysis. It should be noted that we selected the studies in a non-systematic way and tried to consider state-of-the-art studies on distributed tracing data analysis.

The methods based on distributed tracing data analysis used supervised, unsupervised machine learning algorithms or tracing data comparison for failure detection (Soldani et al., 2022). Distributed tracing

data are used to obtain call paths between microservices while responding to a request. A call path may be faulty from two perspectives: 1) The microservices' call sequence in the call path may be incorrect, or 2) The response time of a microservice in a call path may be out of expectation (Liu et al., 2020; Meng et al., 2021). The studies based on distributed tracing data analysis are focused on one of these two types of failure. Also, the methods based on distributed tracing data analysis provide the possibility of locating the root cause of the failure due to the use of the call path of the microservices. In Table 1, the characteristics of studies based on distributed tracing data analysis are given in terms of the goal, output, and algorithm.

According to Table 1, studies based on the distributed tracing data analysis are divided into two categories based on the purpose. The first category is studies whose main purpose is failure detection; so, they have used three methods of unsupervised learning (Nedelkoski et al., 2019a, 2019b; Liu et al., 2020; Chen et al., 2020, 2023; Jin et al., 2020; Bogatinovski et al., 2020; Meng et al., 2020; Gan et al., 2021; Zhang et al., 2022), supervised learning (Gan et al., 2019; Zhou et al., 2019), or tracing data comparison (Meng et al., 2021; Guo et al., 2020). The second category is studies whose main goal is failure localization (Yu et al., 2021; Kim et al., 2013; Liu et al., 2021; Li et al., 2021). So that if they detect the failure they locate the failure manually according to the system graph.

Considering that using distributed tracing data provides the possibility of understanding the system behavior without the need for domain knowledge and a system graph. So, studies based on distributed tracing data analysis can detect failures without the need for domain knowledge. However, each study has advantages and disadvantages according to the use of one of the three methods, unsupervised learning, supervised learning, or tracing data comparison which we will explain below.

In unsupervised learning methods, there is no need to label tracing data, and assuming that the system works normally mostly, its normal behavior is obtained in the training phase. Therefore, unsupervised learning methods are suitable for use in monitoring systems and are compatible with the characteristics of dynamicity and frequent updates in microservice-based systems (Liu et al., 2020; Mariani et al., 2018). In supervised learning methods, there is a need to label the data to train the model. Therefore, labeling data is time-consuming and is in contradiction with the characteristics of dynamicity and frequent updates in microservice-based systems and making it difficult to update the trained model (Gan et al., 2019; Mariani et al., 2018; Zhou et al., 2019). In methods based on tracing data comparison, there should be a predetermined tracing data set to compare. Therefore, the time complexity of the comparison-based methods depends on the size of the predetermined tracing data set; so, they are not suitable for online failure detection and use in large-scale microservice-based systems (Meng et al., 2021; Guo et al., 2020). Considering the advantages of unsupervised learning methods compared to the other methods and the purpose of this study to detect the failure in the sequence of events of distributed tracing data, in the following, we review studies that used unsupervised learning methods.

A distributed tracing data sample is a sequence of events; so, unsupervised learning methods used neural networks to detect failures in the sequence of events. On the other hand, due to the dynamicity of microservices at runtime and different users' requests, the call paths of microservices and their lengths are different (Wang et al., 2018; Liu et al., 2021; Mariani et al., 2018; Nedelkoski et al., 2019b). Six studies (Liu et al., 2020; Nedelkoski et al., 2019b; Bogatinovski et al., 2020; Meng et al., 2020; Zhang et al., 2022; Chen et al., 2023) that detect the failure in the sequence of events of distributed tracing data samples, have paid attention to different call paths for different users' requests. However, each one still has disadvantages that lead to the low performance of the failure detection model in terms of accuracy and speed. The vector proposed by TraceAnomaly (Liu et al., 2020) to combine response time and sequence of events has high dimensions, and model training and online failure detection overhead. The Midiag (Meng et al., 2020) and TraceGra (Chen et al., 2023) train a distinct model for each pattern of microservices' call path. Therefore, due to the large number of

Table 1
Characteristics of studies based on the distributed tracing data analysis.

Work	Characteristic				
	Year	Name	Purpose	Algorithm	Output
Kim et al. (2013)	2013	MonitorRank	Detection of the root cause of failure	Random Walk	Sorted list of candidate services as the root of failure
Jin et al. (2020)	2020	–	Failure detection and root cause analysis	PCA, Random Forest, SVM, and LOF	Sorted list of candidate services as the root of failure
Liu et al. (2020)	2020	TraceAnomaly	Failure detection	Deep Bayesian Network	Failed distributed tracing data samples
Nedelkoski et al. (2019b)	2019	–	Failure detection	Multimodal LSTM	Failed distributed tracing data samples
Gan et al. (2019)	2019	Seer	Failure detection	Convolutional Neural Network	Failed microservices
Nedelkoski et al. (2019a)	2019	–	Failure detection	Convolutional Neural Network	Failed microservices
Bogatinovski et al. (2020)	2020	–	Failure detection	Encoder-Decoder Neural Network	Failed distributed tracing data samples
Zhou et al. (2019)	2019	MEPFL	Failure detection and localization	Supervised Learning	Failed microservices
Meng et al. (2021)	2021	–	Failure detection	Trace Comparison	Failed distributed tracing data samples
Meng et al. (2020)	2020	Midiag	Failure detection	GRU	Failed distributed tracing data samples
Liu et al. (2021)	2021	MicroHECL	Failure detection and localization	Breadth-First Search	Sorted list of candidate services as the root of failure
Li et al. (2021)	2021	TraceRCA	Failure detection and localization	Unsupervised Multi-Metric Trace Anomaly Detection	Sorted list of candidate services as the root of failure
Yu et al. (2021)	2021	MicroRank	Latency error detection and localization	Comparison with SLO - PageRank	Sorted list of candidate services as the root of failure
Guo et al. (2020)	2020	GMTA	Failure detection and localization	Trace Comparison	Failed microservices
Gan et al. (2021)	2021	Sega	Failure detection	Causal Bayesian Networks	Failed microservices
Chen et al. (2020)	2020	VWR	Failure detection	Matrix Sketch-Based Streaming Anomaly Detection	Failed microservices
Zhang et al. (2022)	2022	DeepTraLog	Failure detection	Gated Graph Neural Network based Deep SVDD	Failed distributed tracing data samples
Chen et al. (2023)	2023	TraceGra	Failure detection	Encoder–Decoder using Graph Neural Network and LSTM	Failed distributed tracing data samples

microservices, the complexity of the communication between them, and the call paths with different patterns, many failure detection models should be trained; this causes overhead during training and detecting failures. In another study (Bogatinovski et al., 2020), only events in the neighborhood of an event are considered to train the failure detection model. Therefore, the failure detection model has low performance in terms of accuracy. In the study presented by Nedelkoski et al. (2019b), to train the failure detection model, only the previous events of each event in the tracing data are considered. Therefore, the failure detection model has low performance in terms of accuracy. Also, due to the frequent updates in microservice-based systems and the creation of new call paths, the five mentioned studies (Nedelkoski et al., 2019b; Bogatinovski et al., 2020; Meng et al., 2020; Zhang et al., 2022; Chen et al., 2023) do not update the trained model to learn the sequence of events in new call paths. Therefore, the performance of the trained model will decrease over time.

According to the disadvantages of the reviewed studies, there are still challenges to detecting failures in the sequence of events in distributed tracing data samples using unsupervised learning. In the following, we explain these challenges.

1) **Increasing the accuracy and reducing the false-positive rate of the failure detection model:** As mentioned before, according to the dynamicity of microservice-based systems at runtime, the call paths of microservices are different depending on the users' requests. So, the sequence of events in the tracing data samples which form the call paths, have different patterns and variable lengths. The review of studies shows that the existing methods still do not have acceptable performance in terms of accuracy and false-positive rate to detect failures in the sequence of events of distributed tracing data samples according to different types of call paths. Therefore, there is a need to improve the performance of failure detection in microservice-based systems in terms of increasing the accuracy and reducing the false-positive rate.
2) **Maintaining the performance of the failure detection model according to frequent updates of microservice-based systems:** Considering that, microservices are frequently updated by the development team and frequent updates in microservice-based systems cause new features to be added and new call paths to be created. Therefore, to maintain the performance of the failure detection model, the model should be periodically updated.

Considering the mentioned challenges, the purpose of this study is to improve the process of failure detection in microservice-based systems using distributed tracing data in terms of increasing accuracy and reducing the false-positive rate. In the following, we explain the proposed method to overcome the mentioned challenges.

4. The proposed failure detection method

Considering that, the purpose of this study is to improve the process of failure detection in microservice-based systems using distributed tracing data in terms of increasing accuracy and reducing the false-positive rate. According to the studies based on the distributed tracing data analysis in Section 3, we selected the proposed method by Nedelkoski et al. (2019b) as the basic method and improved it. The reasons for selecting the mentioned method as the basic method are 1) It has a comprehensive process to detect failures, and 2) The tracing data format used in it is similar to the Zipkin and the benchmark used in this study. In this section, we describe the proposed method; then, we explain the assumptions and limitations of the proposed method.

4.1. Overview of the proposed method

The activity diagram of the proposed method is shown in Fig. 6. According to Fig. 6, the process of the proposed method consists of three phases. In the first phase, the failure detection model is trained using the tracing data set obtained from the normal execution of the system as training data. The first phase is done once at the beginning of the process. Then, the second phase is performed for each new tracing data sample to detect failure. Thus, the status of the tracing data sample is checked using the trained model. If the tracing data sample is detected as normal, it is stored in a set to update the model in the third phase, but if the tracing data sample is detected as a failure, its status is checked manually again. So, if the tracing data sample is incorrectly detected as a failure by the model but detected as normal manually, it will be stored in the data set to update the model. Finally, the third phase is periodically performed to update the failure detection model to cover new or less frequent call paths. Thus, in the third phase, the failure detection model is updated using the tracing data set obtained from the second phase. In the following, we describe the process phases of the proposed method and the improvement points of each phase compared to the basic method according to Fig. 6.

4.1.1. Phase 1: training and evaluating the failure detection model

As shown in Fig. 6; the first phase consists of three steps: preprocessing, training, and evaluating the failure detection model. Considering that, each tracing data sample is composed of some events as $T = \{e_1, e_2, e_3, \ldots, e_t\}$ which are sorted based on timestamp; so we need a failure detection model that detects the most probable event in each position of the tracing data sample according to its previous and next events and identifies the correct order of events. Therefore, considering the basic method, we model the failure detection problem in the sequence of events of the tracing data samples as a sequence-to-sequence multiclass classification problem. The input of the sequence-to-sequence multiclass classification should be in the form of numerical vectors; so, in sequence-to-sequence multiclass classification, each unique event is considered a class and the value of each position in the vector corresponds to the unique value of each event in the tracing data sample. In the preprocessing step, we convert the tracing data samples into numerical vectors that can be injected into the model. Then, in the second and third steps, we train and evaluate the model using the preprocessed data set. In the following, we will describe the steps of the first phase.

Step 1. Preprocessing: By running a microservice-based system, we have a set of tracing data samples. Each tracing data sample consists of a number of events with the same "traceID" value. As previously mentioned, tracing data samples should be converted into numerical vectors with the same length to inject as input of the sequence-to-sequence multiclass classification. In the following, we describe the preprocessing step for a tracing data sample; so, the preprocessing step will be done similarly for all tracing data samples. The activity diagram of the preprocessing step is shown in Fig. 7.

In the sequence-to-sequence multiclass classification, each event represents a class and should be mapped to a unique numerical value that represents its class. For this purpose, the basic method uses one-hot encoding. In the one-hot encoding, a binary vector with the length of the number of unique events in the tracing data samples is considered to map the unique value to each event; so, the position of the vector corresponding to the desired event will have a value of one and the rest of the positions will have a value of zero. As a result, if we consider the maximum length of the tracing data samples to be m and the number of unique events to be n; to map a tracing data sample to a numerical vector, we will have a sparse vector with a length of m*n. Therefore, if the number of classes in a data set is large, the one-hot encoding has high spatial complexity and is not suitable (Dahouda et al., 2021). Considering that, in the sequence-to-sequence multi-class classification model to detect failures in the sequence of events, each unique event is considered as a class. Also, in a microservice-based system, each pair of microservice and request form an event; so, the number of unique events depends on the number of microservices, and using the one-hot

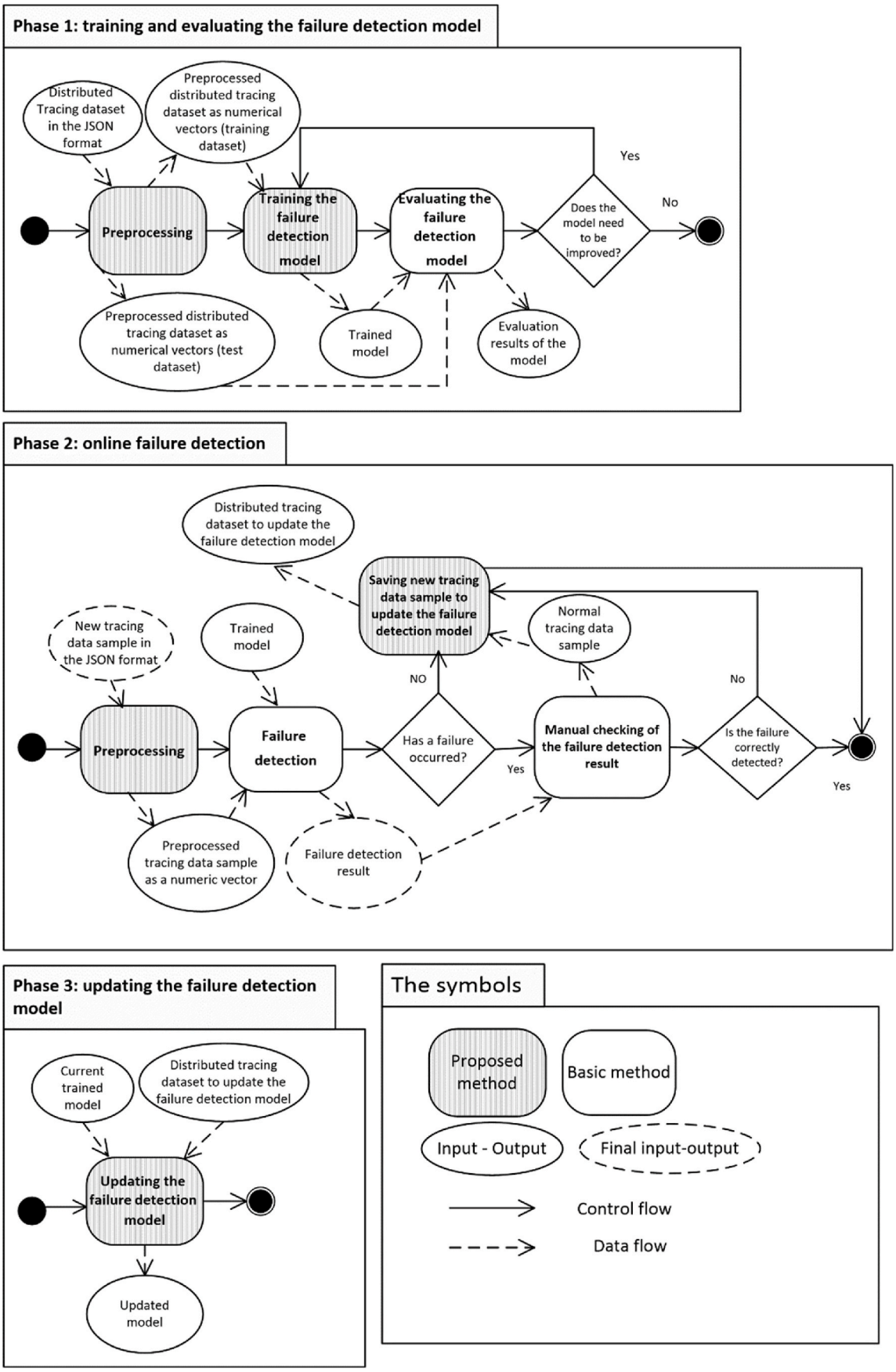

Fig. 6. The failure detection process of the proposed method as an activity diagram.

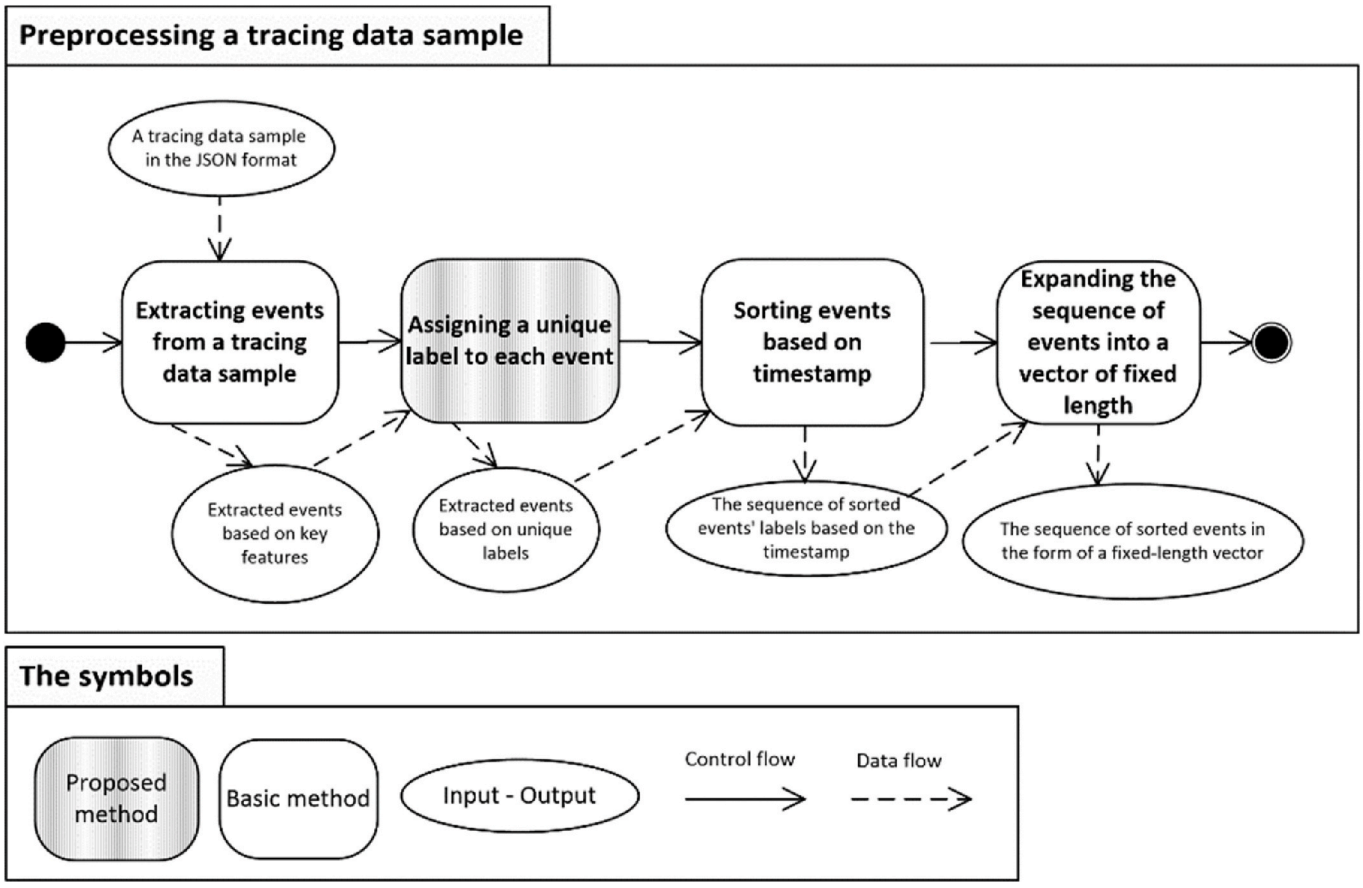

Fig. 7. The preprocessing step of the first phase of the proposed method as an activity diagram.

encoding to map events into numerical vectors is not suitable. For this purpose, we examined other methods that are used to map the input of the sequence-to-sequence multiclass classification into numerical vectors; in the following, we introduce these methods and explain the reason for selecting the desired method.

One of the new methods used to convert non-numerical values into numerical vectors to inject as input of machine learning models is vector embedding. For example, the vector representation of each event in tracing data samples is learned using vector embedding. Considering that, the vector embedding learns the vector representation of each class; it requires a large amount of data to learn and has more time complexity than other methods. Another method used to convert non-numeric values into numerical vectors to inject as input of machine learning models is integer encoding. In the integer encoding method, each class is mapped to a unique integer. Therefore, the integer encoding method has less time and spatial complexity than the other two methods (Dahouda et al., 2021).

Considering the advantages and disadvantages of the three encoding methods and the fact that only the order of the events in the sequence is important to detect failure, in this study, we selected the integer encoding method. In the following, we describe the preprocessing step of the first phase of the proposed method to convert a tracing data sample into a numerical vector according to Fig. 7.

We extract the events from a tracing data sample based on the key

{"traceId":"0000000000006616","name":"GetRequest","id":"0000000000006616","annotations":[{"endpoint":{"serviceName":"www.denominator","ipv4":"54.198.0.1","port":8080},"timestamp":1533807096489216,"value":"cs"},{"endpoint":{"serviceName":"www-elb.elb","ipv4":"54.198.0.2","port":8080},"timestamp":1533807096489261,"value":"sr"},{"endpoint":{"serviceName":"www-elb.elb","ipv4":"54.198.0.2","port":8080},"timestamp":1533807096489582,"value":"ss"},{"endpoint":{"serviceName":"www.denominator","ipv4":"54.198.0.1","port":8080},"timestamp":1533807096489594,"value":"cr"}]},
{"traceId":"0000000000006616","name":"GetRequest","id":"0000000000006617","parentId":"0000000000006616","annotations":[{"endpoint":{"serviceName":"www-elb.elb","ipv4":"54.198.0.2","port":8080},"timestamp":1533807096489335,"value":"cs"},{"endpoint":{"serviceName":"wwwproxy.zuul","ipv4":"50.19.0.6","port":8080},"timestamp":1533807096489367,"value":"sr"},{"endpoint":{"serviceName":"wwwproxy.zuul","ipv4":"50.19.0.6","port":8080},"timestamp":1533807096489556,"value":"ss"},{"endpoint":{"serviceName":"www-elb.elb","ipv4":"54.198.0.2","port":8080},"timestamp":1533807096489567,"value":"cr"}]},
{"traceId":"0000000000006616","name":"GetRequest","id":"0000000000006618","parentId":"0000000000006617","annotations":[{"endpoint":{"serviceName":"wwwproxy.zuul","ipv4":"50.19.0.6","port":8080},"timestamp":1533807096489389,"value":"cs"},{"endpoint":{"serviceName":"homepage.karyon","ipv4":"50.19.0.7","port":8080},"timestamp":1533807096489406,"value":"sr"},{"endpoint":{"serviceName":"homepage.karyon","ipv4":"50.19.0.7","port":8080},"timestamp":1533807096489532,"value":"ss"},{"endpoint":{"serviceName":"wwwproxy.zuul","ipv4":"50.19.0.6","port":8080},"timestamp":1533807096489544,"value":"cr"}]},
{"traceId":"0000000000006616","name":"GetRequest","id":"0000000000006619","parentId":"0000000000006618","annotations":[{"endpoint":{"serviceName":"homepage.karyon","ipv4":"50.19.0.7","port":8080},"timestamp":1533807096489421,"value":"cs"},{"endpoint":{"serviceName":"subscriber.staash","ipv4":"50.19.0.3","port":8080},"timestamp":1533807096489435,"value":"sr"},{"endpoint":{"serviceName":"subscriber.staash","ipv4":"50.19.0.3","port":8080},"timestamp":1533807096489511,"value":"ss"},{"endpoint":{"serviceName":"homepage.karyon","ipv4":"50.19.0.7","port":8080},"timestamp":1533807096489523,"value":"cr"}]},
{"traceId":"0000000000006616","name":"GetRequest","id":"0000000000006620","parentId":"0000000000006619","annotations":[{"endpoint":{"serviceName":"subscriber.staash","ipv4":"50.19.0.3","port":8080},"timestamp":1533807096489455,"value":"cs"},{"endpoint":{"serviceName":"cassSubscriber.priamCassandra","ipv4":"50.19.0.4","port":8080},"timestamp":1533807096489470,"value":"sr"},{"endpoint":{"serviceName":"cassSubscriber.priamCassandra","ipv4":"50.19.0.4","port":8080},"timestamp":1533807096489483,"value":"ss"},{"endpoint":{"serviceName":"subscriber.staash","ipv4":"50.19.0.3","port":8080},"timestamp":1533807096489492,"value":"cr"}]},

Fig. 8. A tracing data sample obtained from running the VWR benchmark.

features of the events. The key features of events are a set of features that uniquely specify the type of the event. For this purpose, we select the key features based on the format defined in the Zipkin distributed tracing data collection tool and extract the unique events based on these features. A tracing data sample obtained from running the VWR benchmark is shown in Fig. 8. The format of the tracing data sample in Fig. 8 is according to the format defined in Zipkin. In the following according to Fig. 8, we describe the events' features and the key features that uniquely specify the events.

As shown in Fig. 8, the features of the tracing data sample are specified as name-value pairs. In Table 2, we describe each feature.

According to the definition of an event in Section 2, an event represents the call between two microservice instances in the form of caller → callee. As shown in Fig. 9, each event consists of some subevents that determine the way of sending or receiving messages between two microservice instances. The way of communication between three microservices, their events, and subevents are shown in Fig. 9.

In Fig. 9, the order of subevents is specified with numbers 1 to 8 based on the timestamp. As shown in Fig. 9, the order of calling microservices to respond to a request based on the timestamp is determined by subevents. Therefore, we use subevents and sort them based on the timestamp to detect failures in the call path of microservices. In the continuation of this study, the term "event" refers to the subevents.

To train the failure detection model in the call path of microservices, unique events should be extracted and each unique event should be mapped to an integer that represents its class. For this purpose, we consider the features that uniquely specify the events. Among the features in Table 2, the set of {Name, ServiceName, IPv4, Port, Value} specify an event; But due to the two features {IPv4, Port} specify the microservice instance that sends or receives the request in an event, and the instances of microservices in a microservice-based system are transient and have a short lifetime. Therefore, to learn the call path between microservices to detect failure, we consider only three features {Name, ServiceName, Value} that uniquely specify events. Then, we use {IPv4, Port} to locate failures at the level of microservice instances. In Fig. 10, the events extracted from a tracing data sample using three features {Name, ServiceName, Value} are shown.

Table 2
Description of the features of a tracing data sample.

Feature name	Description
TraceID	A unique identifier that identifies each tracing data sample, so that all events belonging to a tracing data sample have the same "traceID".
ID	A unique identifier that identifies each event of each tracing data sample. Each event represents the call between two microservice instances in the form of caller → callee.
Name	The name specifies the type of request; The HTTP requests are supported in the tracing data. So, two types of requests named GetRequest and Put are used to simulate the microservice-based system in the VWR benchmark.
Annotations	The features array of the endpoints; endpoints are two microservices that form an event.
ServiceName	The name of an endpoint
IPv4	The IP of the host of a microservice instance
Port	The port of a microservice instance
Timestamp	The time of sending, receiving, or responding to a request
Value	This feature can take 4 values cs, cr, ss, or sr. So that these 4 values specify how to send or receive requests between two microservices as a client-server.

After extracting the events from the tracing data sample to convert the sequence of events into a numerical vector according to the integer encoding method, we assign a unique label to each unique event. For this purpose, we use the "HashMap" data structure and incremental identifier. The way of assigning a unique label to each unique event is given in Pseudocode 1.

```
if(!HashMap.containsKey(event)){
   HashMap.put(event,label);
   label++;
}
thisLabel= HashMap.get(event);
```

Pseudocode 1. Assigning a unique label to each unique event in the trace data sample

After determining the label of each event of the tracing data sample, the labels should be sorted according to the timestamp of the events. For this purpose, we use the quick sort algorithm, which is one of the efficient sorting algorithms and has an average time complexity of O(nlogn) (Mishra et al., n.d.).

As mentioned before, we model the failure detection problem in the events sequence of tracing data samples as a sequence-to-sequence multiclass classification problem. The input of the sequence-to-sequence multiclass classification model is numerical vectors with a fixed length. On the other hand, according to the dynamicity of microservice-based systems at runtime, the call paths of microservices have different lengths to respond to different requests. Therefore, we convert the numerical vectors of events' labels obtained from the previous sub-step into vectors with a fixed length. The conventional method in machine learning (Yoon et al., 2020), which is also used in the basic method is padding to expand the vectors to a desired maximum length. We determine the maximum length equal to the maximum length of the tracing data samples in the training data set. To expand the vectors, we fill the end of each vector with zero digits up to the maximum length. If the length of the vector is greater than the maximum length, we ignore the extra indices. Considering that the maximum length of the tracing data samples obtained from the running of the VWR benchmark is 30; we consider the maximum length for padding equal to 30. The final output of the preprocessing step for the tracing data sample of Fig. 8 is as follows:

[1 2 5 6 9 10 13 14 17 18 19 20 15 16 11 12 7 8 3 4 0 0 0 0 0 0 0 0 0 0]

The preprocessing step for all tracing data samples will be done similarly. Therefore, after performing the preprocessing step on the tracing data samples obtained from the running of the VWR benchmark, we divide them into two sets of training and testing. For this purpose, we consider 80% of the data for training and 20% for testing. In the following, we explain the failure detection model training step.

Step 2. Training the failure detection model: Considering that each tracing data sample consists of some events as $T = \{e_1, e_2, e_3, \ldots, e_t\}$, which are sorted based on the timestamp. Therefore, we need a failure detection model to reconstruct the sequence of events and detect the correctness of the order of events in the sequence. The basic method has modeled the failure detection problem in the sequence of events of the tracing data samples as a sequence-to-sequence multiclass classification problem. The input of the sequence-to-sequence multiclass classification problem for the tracing data sample $T = \{e_1, e_2, e_3, \ldots, e_t\}$ is the numerical vector

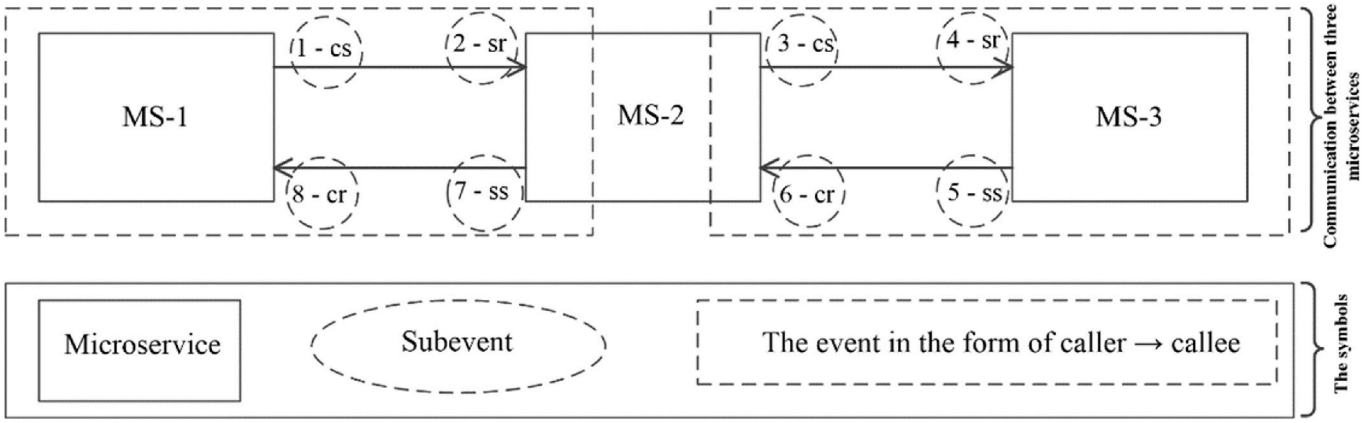

Fig. 9. The way of communication between the three microservices, their events, and subevents.

```
www.denominator,GetRequest,cs | www-elb.elb,GetRequest,sr | www-elb.elb,GetRequest,ss | www.denominator,GetRequest,cr
www-elb.elb,GetRequest,cs | wwwproxy.zuul,GetRequest,sr | wwwproxy.zuul,GetRequest,ss | www-elb.elb,GetRequest,cr
wwwproxy.zuul,GetRequest,cs | homepage.karyon,GetRequest,sr | homepage.karyon,GetRequest,ss | wwwproxy.zuul,GetRequest,cr
homepage.karyon,GetRequest,cs | subscriber.staash,GetRequest,sr | subscriber.staash,GetRequest,ss | homepage.karyon,GetRequest,cr
subscriber.staash,GetRequest,cs | cassSubscriber.priamCassandra,GetRequest,sr | cassSubscriber.priamCassandra,GetRequest,ss | subscriber.staash,GetRequest,cr
```

Fig. 10. Events extracted from a tracing data sample.

corresponding to $T_{input} = \{e_1,e_2,e_3, \ldots, e_{t-1}\}$, and the output is the most probable event in each position of the tracing data sample in the form of a numerical vector corresponding to $T_{output} = \{e_2,e_3,e_4, \ldots, e_t\}$. Therefore, we detect the failure in the sequence of events of a tracing data sample by reconstructing the output based on the input. The basic method has used the LSTM layer to train the sequence-to-sequence multiclass classification model. By using the LSTM layer, it is possible to determine the most probable event in each position of the tracing data sample according to its previous event and without considering its next event. But in microservice-based systems, there are different call paths depending on the users' requests. There are four types of basic call paths in microservice-based systems as shown in Fig. 11. The four types of basic call paths are (Bellur, 2007):

- **Sequential call path:** In the sequential call path, a microservice only calls another specific microservice.
- **Alternate call path:** In the alternate call path, a microservice calls a different microservice depending on the user's request.
- **Concurrent call path:** In the concurrent call path, a microservice calls two or more other microservices in parallel.

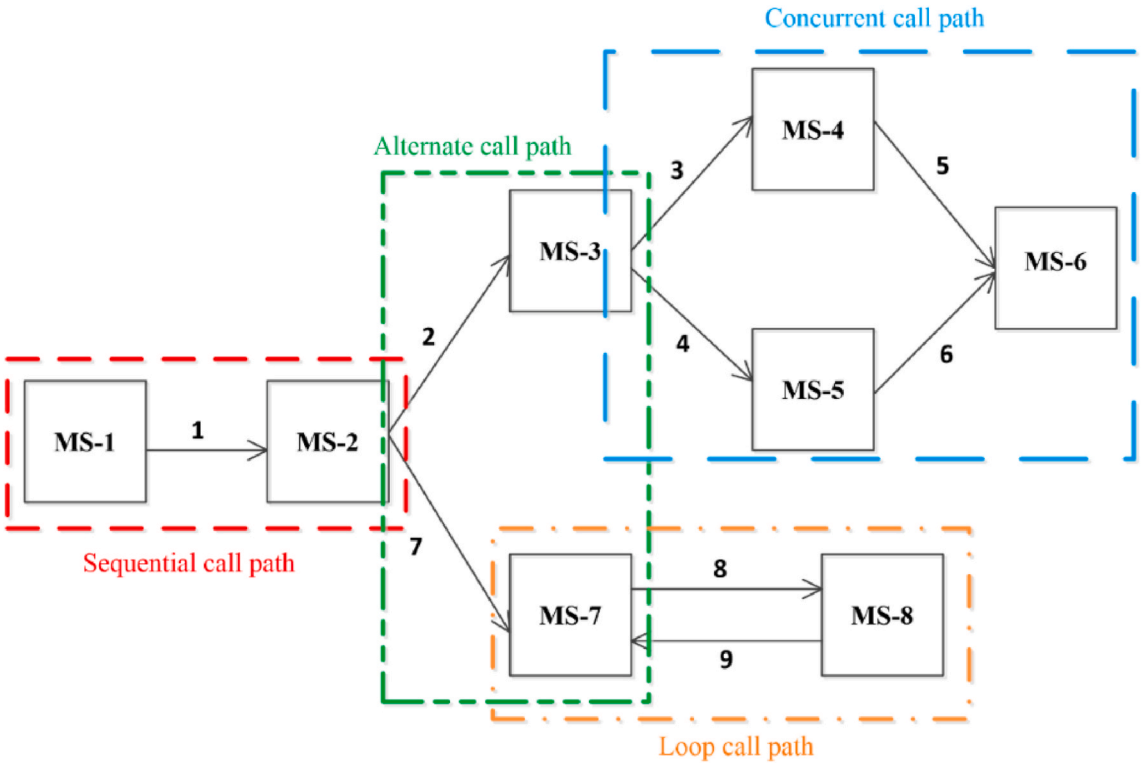

Fig. 11. Types of call paths in microservice-based systems.

- **Loop call path:** There is a loop in the call path of microservices and the loop path can be treated as a single sequential path.

In microservice-based systems, call paths are a combination of one or more basic call paths. Using a sequence-to-sequence multiclass classification model with an LSTM layer is suitable for reconstructing events in sequential call paths; because in the sequential call paths, each event is only dependent on its previous event. We can consider the loop call path as a single sequential call path, which is repeated. But using a sequence-to-sequence multi-class classification model with an LSTM layer to reconstruct events in alternate and concurrent call paths does not have an acceptable performance; in the following, we explain the reason with an example.

A simple example of call paths of microservices in an online shop is given in Fig. 12. As shown in Fig. 12, depending on the user's request, the front-end service calls one of the search or add-to-cart microservices. Therefore, there is an alternate call path in the front-end microservice. So, if the search request rate is higher than add-to-cart; Using the LSTM layer, {search, GetRequest, sr} is detected as the most probable event as the next event of the {front-end, GetRequest, cs} event. So that if the user's request is add-to-cart; the trace data sample will be incorrectly detected as a failure and the false-positive rate will increase. In concurrent call paths, due to the parallel calling of two or more microservices at the same time, the sequence of events in the tracing data samples may be different at runtime. Therefore, only paying attention to the previous event of each event in the concurrent call path will increase the false-positive rate.

Considering that, the output of the sequence-to-sequence multiclass classification model for each tracing data sample is a matrix with the dimensions of the number of unique events multiplied by the length of the input vector; so, in the mentioned matrix, the probability of occurrence of each unique event in each position of the tracing data sample is determined. To reduce the false-positive rate, the basic method considers K probable events for each position of the tracing data sample, and if all observed events in each position of the tracing data sample will be in the set of K probable events detected by the failure detection model, the tracing data sample will be considered as a normal sequence. The performance of the failure detection model depends on the value of K. If the value of K is considered too large, it increases the false-negative rate, and if it is considered too small, the false-positive rate increases. Therefore, in the proposed method, we are trying to increase the accuracy of the failure detection model in $K = 1$ by improving the basic method. So, we detect the failure in the tracing data sample by comparing the event observed in each position of the tracing data sample with the most probable event detected by the model. In the following, we explain the proposed sequence-to-sequence multiclass classification model.

In the proposed method, to increase the accuracy and reduce the false-positive rate, we calculate the probability of occurrence of each event in each position of the tracing data sample according to its previous and next events. For this purpose, we use the Bi-LSTM layer instead of the LSTM layer in the sequence-to-sequence multiclass classification model. The layers of the proposed sequence-to-sequence multiclass classification are shown in Pseudocode 2. As shown in Pseudocode 2, the mentioned classification consists of 6 layers. In the following, we explain each layer separately.

Input layer: This layer injects the numerical vector corresponding to each tracing data sample obtained in the preprocessing step into the model. The length of the input vector should be constant for all tracing data samples. For example, the input for tracing data sample $T = \{e_1, e_2, e_3, \ldots, e_t\}$ is the numerical vector corresponding to $T_{input} = \{e_1, e_2, e_3, \ldots, e_{t-1}\}$.

Bi-LSTM layer: According to Section 2, the Bi-LSTM layer consists of two LSTM layers in two forward and backward directions. So, the numerical vector corresponding to the sequence of events of each tracing data sample is processed once from the beginning to the end and once from the end to the beginning. Therefore, the Bi-LSTM layer learns the order of each event of each tracing data sample according to its previous and next events.

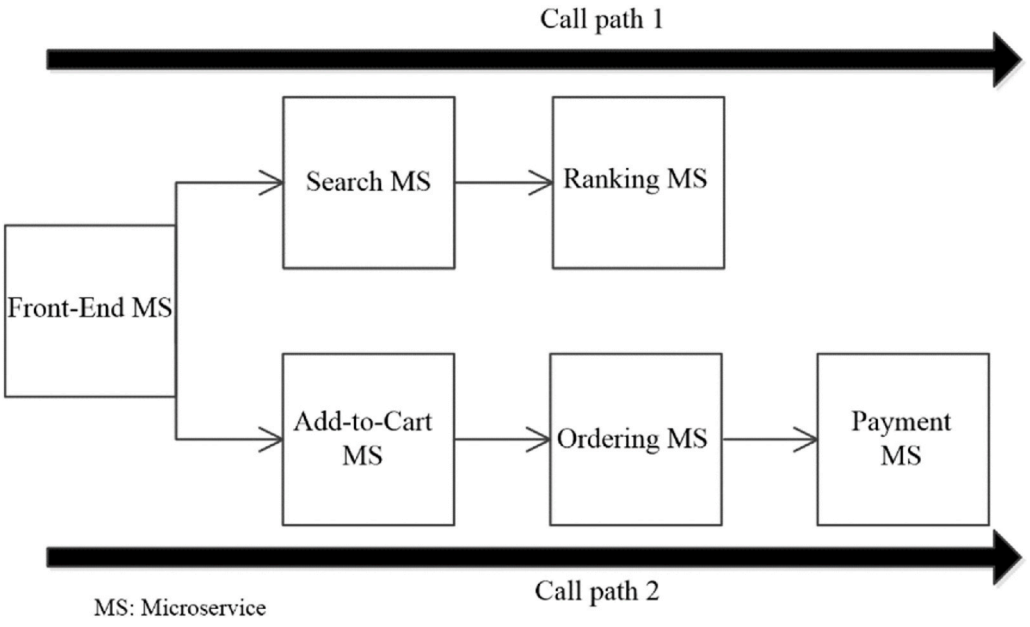

Fig. 12. A simple example of the alternate call path of microservices in an online shop.

```
layers = [ ...
    sequenceInputLayer(inputSize)
    bilstmLayer(numHiddenUnits,'OutputMode','sequence')
    dropoutLayer(0.1)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Pseudocode 2. The layers of the proposed sequence-to-sequence multiclass classification model

Dropout layer: Using the dropout layer is a way to prevent overfitting in neural networks. So, using the dropout layer previous to the other neural network layers causes some of the input neurons of the desired layer to be ignored with the probability of P, and more useful features to be learned. In the proposed model, we use the dropout layer next to the Bi-LSTM layer and previous the fully connected layer. Because using the dropout layer previous to the Bi-LSTM layer causes some order of events in the sequence to be forgotten while they should be learned. So, we use a dropout layer next to the Bi-LSTM layer and previous to the fully connected layer, and this causes the weights of the fully connected layer to be calculated considering all events of tracing data samples (Srivastava et al., 2014). Therefore, the occurrence probability of each unique event in each position of the tracing data sample is obtained according to all tracing data samples.

Fully connected layer: All the outputs of the Bi-LSTM layer are connected to each neuron of the fully connected layer. The number of neurons in the fully connected layer is equal to the number of sequence-to-sequence multiclass classification classes. In the problem of failure detection in the sequence of events of tracing data samples, the number of classes is equal to the number of unique events. The output of the fully connected layer will be a matrix with the dimensions of the number of unique events in the length of the input vector (the length of the tracing

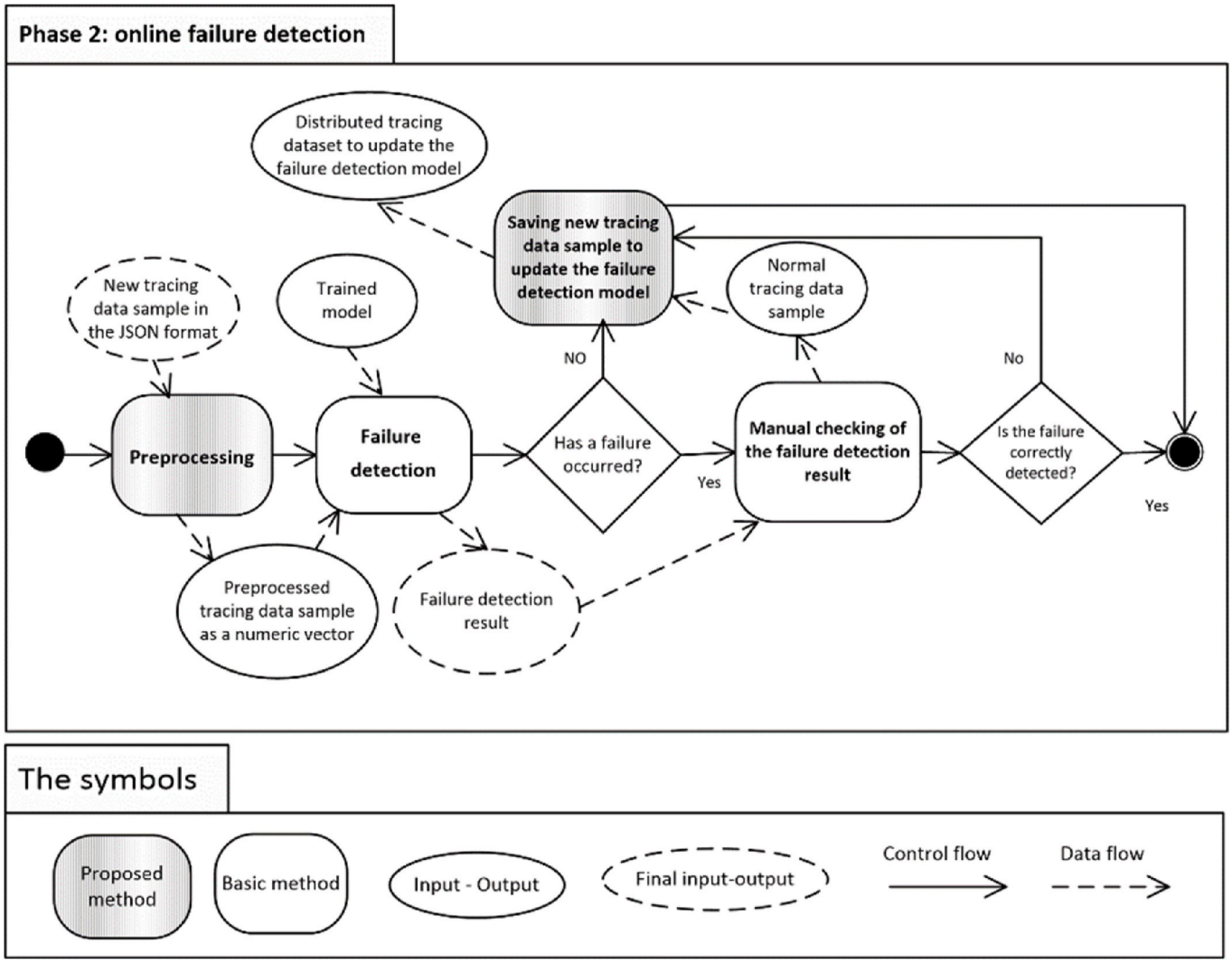

Fig. 13. The second phase of the proposed method as an activity diagram.

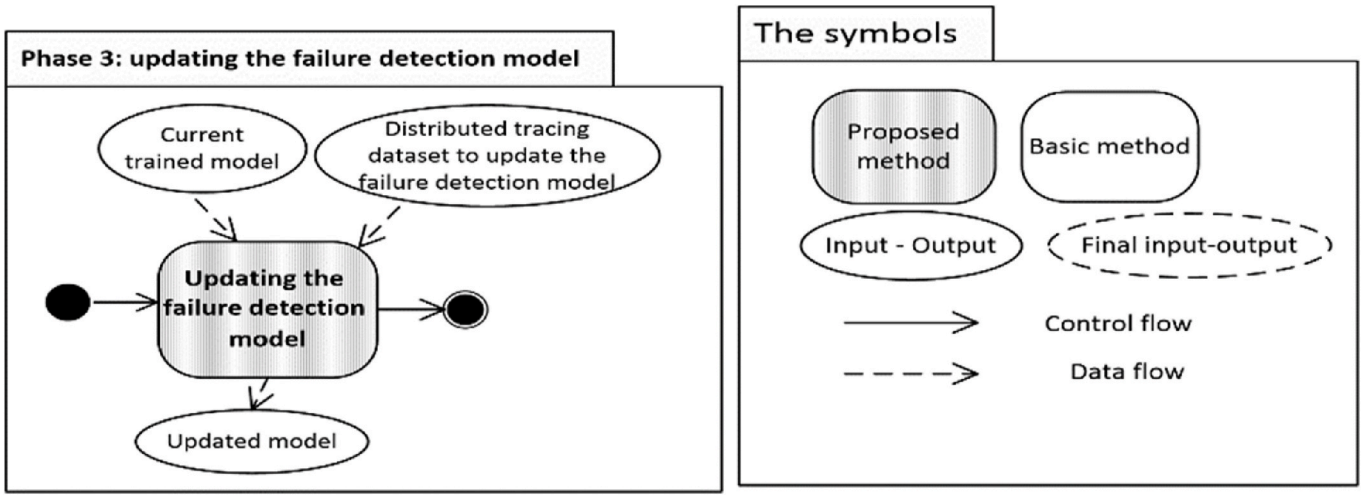

Fig. 14. The third phase of the proposed method as an activity diagram.

data sample). Therefore, by using the fully connected layer, it is possible to obtain the occurrence probability of each unique event in each position of the tracing data sample according to the samples in the training data set.

Softmax layer: The softmax layer receives the output of the fully connected layer as input and applies the softmax function on its input to transform input values between zero and one and maintains the probability distribution. Therefore, in the multiclass classification model, the softmax function is applied to each column and turns the occurrence probability of each unique event in each position of the tracing data sample into a probability distribution.

Classification layer: The classification layer calculates the value of the cross-entropy loss function for the classifier. This layer infers the number of classes from its previous layer. Therefore, the fully connected layer and the softmax layer are used to determine the number of unique events in the failure detection model previous to the classification layer.

To train the failure detection model, for each tracing data sample $T = \{e_1,e_2,e_3, \ldots, e_t\}$ in the training set; The input-output pair is injected into the model as a numerical vector corresponding to $T_{input} = \{e_1,e_2,e_3, \ldots, e_{t-1}\}$ and a numerical vector corresponding to $T_{output} = \{e_2,e_3,e_4, \ldots, e_t\}$. In the training step, the weights of each layer are obtained to minimize the loss function according to each input-output pair of the tracing data samples of the training data set. The loss function specifies the distance between the observed output value for each input and the value detected by the model (Du et al., 2017; Hochreiter et al., 1997). It is worth mentioning, the tracing data set used to train the model includes normal tracing data samples. Therefore, the trained model is based on the normal execution of the desired system.

There are various optimization methods to minimize the value of the loss function. The basic method used stochastic gradient descent to minimize the value of the loss function. In this study, we use the Adam optimizer to minimize the value of the loss function. Adam optimizer converges to the minimum value faster than the stochastic gradient descent and has a better performance (Kingma et al., 2015). Also, to train the failure detection model, the hyperparameters of the model should be set in such a way that the model has the best performance. There are different ways to set the hyperparameters of the model. One of the common ways to tune hyperparameters is through manual search (Bischl et al., 2023). Thus, the results of the model are compared using different values of hyperparameters and the best ones are selected. In this study, we tune the values of the hyperparameters by manual search and according to the values of the hyperparameters in the basic method.

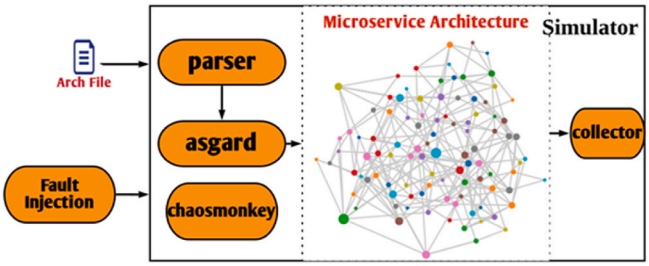

Fig. 15. Framework of VWR simulator.

Step 3. Evaluating the failure detection model: After training the failure detection model, we evaluate the performance of the model using the test data set. If the performance of the model is acceptable according to the evaluation results, the training of the model ends; otherwise, we continue to train the model and tune the hyperparameters until the trained model reaches an acceptable performance according to the training and test data set.

4.1.2. *Phase 2: online failure detection*

While running a microservice-based system, tracing data samples are generated online. So, each tracing data sample shows the call path of microservices to respond to a request. Considering that, each tracing data sample is given online to the failure detection model; in the following, we explain the steps of the online failure detection phase according to Fig. 13.

First, we perform the preprocessing step for each new tracing data sample according to the preprocessing step of the first phase. The output of the preprocessing step is a numerical vector. For the new tracing data sample $T = \{e_1,e_2,e_3, \ldots, e_t\}$, the numerical vector corresponding to $T_{input} = \{e_1,e_2,e_3, \ldots, e_{t-1}\}$ is given to the trained model to detect the failure. The trained model reconstructs the output vector corresponding to $T_{output} = \{e_2,e_3,e_4, \ldots, e_t\}$ considering the input vector and determines the label of each event in each position of the trace data sample considering the most probable event.

To detect the failure in the proposed method, we consider the output vector reconstructed by the trained model and compare it with the vector of observed events in the tracing data sample. So, if all events of both vectors are equal, we consider the tracing data sample normal; otherwise, we consider the tracing data sample as a failed sample. The online failure detection is given in Pseudocode 3.

```
T_input = {e_1, e_2, ..., e_t-1};
T_obsereved = {e_2, e_3, ..., e_t};
 [T_output,scores] = classify(TrainedModel,T_input);
 for i =1: T_obsereved.length {
  if(T_obsereved[i]!=T_output[i]){
    print(Trace is anomaly);
    return;
   }
 }
    print(Trace is normal);
```

Pseudocode 3. Online failure detection in a trace data sample using the trained model

By performing the failure detection step, if the tracing data sample is detected as normal, there is no need to re-check the tracing data sample. Because the number of generated tracing data samples is large and checking all the tracing data samples manually is time-consuming. Therefore, assuming that the tracing data samples that are detected as normal by the failure detection model are really normal; they are stored in a set to update the failure detection model. Therefore, the failed samples detected by the failure detection model should be checked manually.

If the tracing data sample is detected as a failed sample by the failure detection model; the result of the failure detection and the corresponding tracing data sample are re-checked manually. If the failure detection model has incorrectly detected a normal tracing data sample as a failure, the tracing data sample is stored in the set of new normal tracing data samples to update the failure detection model. If the failure detection model has correctly detected the tracing data sample as a failure, it is possible to locate the root cause of failure using the feature set {Name, ServiceName, IPv4, Port, Value} related to each event; in such a way that we find the microservice that caused the failure using the "ServiceName" and find its instance using the two features {IPv4, Port}. After determining the location of the failure, a report is given to the development team responsible for the failed microservice. Then, the development team can determine the cause of the failure and fix it by manual checking and log analysis.

4.1.3. Phase 3: updating the failure detection model

Considering that all call paths between microservices may not be covered while training the failure detection model. Also, new call paths may be created by updating microservice-based systems. To cover new or less frequent call paths, we periodically update the trained model using the tracing data set obtained from the second phase. The model is updated offline using the weights of the current model. After updating the model, we replace the current trained model with the updated model. The activity diagram of the third phase is shown in Fig. 14.

4.2. Assumptions and limitations

There are some assumptions and limitations for the proposed method according to the purpose of this study to improve the basic method (Nedelkoski et al., 2019b), the assumptions and limitations of the basic method, reviewing the related works, and the existing evaluation platform. The assumptions and limitations of the proposed method are defined as follows.

- The tracing data produced by the benchmark (Chen et al., 2020) used to evaluate the proposed method are similar to the format defined in the Zipkin distributed tracing data collection tool (Data Model;Zipkin API); therefore, we perform and explain the preprocessing step and extracting the key features of distributed tracing data events based on the format defined in Zipkin. According to the definition of events and distributed tracing data in Section 2, the preprocessing step can be generalized to other data formats defined by other distributed tracing data collection tools.
- Considering that in a microservice-based system, the development of each microservice is done by an independent team, and frequent updates cause the creation of new call paths. We assume that we do not have access to the source code, its changes, and microservices relationships (Scheinert et al., 2021; Zhou et al., 2019). So, we perform failure detection without prior knowledge of the microservices' call graph and only use distributed tracing data at runtime (Wang et al., 2018; Mariani et al., 2018).
- To train and update the failure detection model, we assume that the system works normally mostly and the failure rate in the system is much lower than the normal execution of the system. So, this assumption is according to the real systems (Guo et al., 2020) that formulates the failure detection problem as an anomaly detection problem. Anomaly detection is the identification of rare items, events, or observations that deviate significantly from the majority of the data. The general idea of anomaly detection is that there is one reference class, and everything that deviates clearly from the reference class is classified as an anomaly (Chandola et al., 2009). Therefore, we train and update the failure detection model only using normal tracing data. Then, we consider any tracing data that does not match the sequence detected by the failure detection model as a failure.
- Considering that the purpose of this study is to detect failures in the events' sequence of call paths of microservices using distributed tracing data. To detect failure, we focus on two aspects of the dynamicity of communication between microservices when running a microservice-based system: 1) Depending on the users' requests, the call paths between microservices are different, and 2) The call paths of microservices have different lengths. Therefore, we assume a maximum length for the call paths. So that the length of the call paths will vary from one to the considered maximum value (Nedelkoski et al., 2019b; Bogatinovski et al., 2020).
- Considering that the frequent updates in the microservice-based systems cause new features, including a new microservice or new call paths between microservices (Yu et al., 2021). In the model updating phase, we assume that all call paths may not be covered during training or new call paths may be created by updating the system. So, we update the model according to the new call paths (Nedelkoski et al., 2019b; Du et al., 2017).

5. Evaluation

In this section, we evaluate the proposed method as follows.

5.1. Assumptions and limitations

In this section, we describe the assumptions and limitations to evaluate the proposed method as follows.

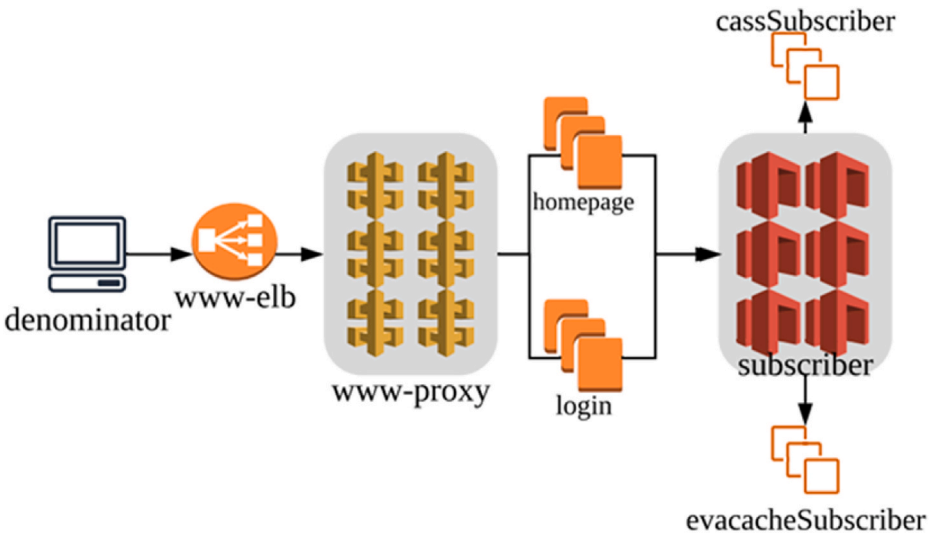

Fig. 16. Netflix architecture to simulate the behavior of a microservice-based system in the VWR benchmark.

- We use the VWR benchmark (Chen et al., 2020) to simulate the behavior of a microservice-based system due to the lack of access to distributed tracing data obtained from running a real system. In VWR, tracing data is generated under the normal execution of the desired system. So, obtained tracing data are suitable to train the failure detection model. Also, it is possible to obtain tracing data while injecting failure and use it to test the model.
- To make a reliable and fair evaluation, we implement both the proposed and basic method and use a similar evaluation platform. Then, we compare both methods using the same experiments. Therefore, we assume that the evaluation results are fair and show the improvement of the proposed method compared to the basic method.

5.2. Context

The goal of the evaluation is to assess the ability of the proposed method to detect the failure in the call paths of microservices using distributed tracing data. In this regard, we need tracing data obtained from the execution of microservices in a microservice-based system. So that the tracing data includes normal and failed sequences. For this purpose, we selected the VWR benchmark to simulate the behavior of a microservice-based system and collected distributed tracing data obtained from the execution of the desired system. In the following, we explain the VWR benchmark, the features of the simulated system in the VWR benchmark, and the features of the obtained tracing data.

The VWR benchmark is used to evaluate anomaly detection methods in microservice-based systems. So, it simulates the behavior of a microservice-based system by emulating the behavior of users and records distributed tracing data to respond to users' requests. In the VWR benchmark, it is possible to define the architecture of the desired microservice-based system and runtime parameters including simulation time and request rate. The simulator of the VWR benchmark has some key components. The design of the simulator is shown in Fig. 15. Four key components are parser, asgard, collector, and chaosmonkey. After uploading the architecture definition file to the simulator, the parser will parse the architecture and send the architecture configuration to asgard. In the definition file, the name of the services, the number of instances per service, the type of each service, and the architecture topology should be defined. According to the received architecture

Table 3
Characteristics of the tracing data used to evaluate the proposed method.

Total number of tracing data samples	1036	**The maximum length for normal tracing data samples**	30
The number of normal tracing data samples	997	**The minimum length for normal tracing data samples**	5
The number of failed tracing data samples	39	**The average length for normal tracing data samples**	20
Total number of events	21181	**The maximum length for failed tracing data samples**	24
The number of unique events	49	**The minimum length for failed tracing data samples**	3
-	–	**The average length for failed tracing data samples**	16


```
{
    "arch": "netflixoss",
    "description":"A very simple Netflix service. See http://netflix.github.io/ to decode the package names",
    "version": "arch-0.0",
    "victim": "homepage",
    "services": [
        { "name": "cassSubscriber",   "package": "priamCassandra", "count": 6,  "dependencies": ["cassSubscriber"]},
        { "name": "evcacheSubscriber","package": "store",          "count": 3,  "dependencies": []},
        { "name": "subscriber",       "package": "staash",         "count": 6,  "dependencies": ["cassSubscriber", "evcacheSubscriber"]},
        { "name": "login",            "package": "karyon",         "count": 18, "dependencies": ["subscriber"]},
        { "name": "homepage",         "package": "karyon",         "count": 24, "dependencies": ["subscriber"]},
        { "name": "wwwproxy",         "package": "zuul",           "count": 6,  "dependencies": ["login", "homepage"]},
        { "name": "www-elb",          "package": "elb",            "count": 1,  "dependencies": ["wwwproxy"]},
        { "name": "www",              "package": "denominator",    "count": 1,  "dependencies": ["www-elb"]}
    ]
}
```


Fig. 17. The architecture file of the simulated Netflix system.

configuration, asgard spawns every needed service instance and builds its dependency lists. Then during the simulation, the collector collects all the produced communication data and converts them into the form of tracing data. Finally, VWR uses chaosmonkey to inject the fault into the system during simulation. So, it is possible to delete a microservice during execution to simulate the failure in the VWR benchmark. To simulate the behavior of the microservice-based systems, VWR supports the key services of a microservice-based system in the simulator including database, load balancer, service registry, and API gateway. By default, three popular microservice-based systems named WS, Netflix, and LAMP are proposed in the VWR benchmark.

To evaluate the proposed method, we simulated the behavior of the Netflix system in the VWR benchmark. The architecture of the simulated Netflix system is shown in Fig. 16. We selected Netflix architecture because it includes sequential, alternate, concurrent, and loop call paths as shown in Fig. 16. So that the call path between www-proxy, homepage, and login is alternate. The call path between the subscriber, cassSubscriber, and evacacheSubscriber is concurrent. Also, the call path between the denominator, www-elb, and www-proxy is sequential, and cassSubscriber has a loop call path with itself.

The architecture file for the simulated Netflix system is given in Fig. 17. In the architecture file, the microservices of the simulated architecture are determined by the “services" parameter. The “name" parameter determines each microservice’s name, the “package" parameter determines the behavior each microservice takes during simulation and the “count" parameter specifies the number of each microservice. The connection between microservices is configured by the “dependencies" parameter.

After defining the architecture of the Netflix system, we configured runtime parameters including simulation time and request rate. We executed the defined Netflix system in the VWR benchmark for 1800 s to obtain the distributed tracing data. Also, we determined the maximum request rate to 100. The VWR benchmark, create different and random load to maximum request rate during execution. By default, executing a system on the VWR benchmark is done without failure. Therefore, first, we collected normal tracing data. Then, to collect the tracing data including failed call paths, we executed the system in different executions and injected the deletion fault to each of the microservices defined in the Netflix system, except for the denominator microservice, which is the front-end microservice. Thus, we obtained 14 unique call paths. Also, to evaluate the performance of the proposed failure detection method for different failed call paths, we randomly selected 25 normal call paths and one of their event and replaced the considered event with another event according to the evaluation done in the basic method. The characteristics of the tracing data used to evaluate the proposed method are given in Table 3.

It is worth mentioning that the VWR benchmark does not report on the coverage of call paths between microservices. But the manual checking of the distributed tracing data obtained from the execution of the Netflix system shows that all the microservices and most call paths are covered.

5.3. Research questions

To address the goal of the study, we defined the following research questions.

RQ1: Can the proposed method detect failures in microservices’ call paths with a low false-positive rate considering different types of call paths compared to the basic method?
RQ2: Can the proposed method maintain the performance of the failure detection model according to frequent updates of microservice-based systems to cover less or new call paths?

RQ1 allows us to assess the ability of the proposed method to detect the failure in microservices’ call paths considering different types of call paths while RQ2 is defined to illustrate the ability of the proposed method to maintain the performance of the failure detection model over time. To respond to the research questions, we performed experiments on distributed tracing data generated by the VWR benchmark including different types of call paths and compared the results to the basic method.

5.4. Evaluation metrics

To evaluate the performance of the proposed method, we used Accuracy, False-Positive Rate (FPR), False-Negative Rate (FNR), Precision, Recall (Sensitivity), Specificity, F-score, G-mean, Balance, Matthews Correlation Coefficient (MCC), and Speed which are the most important metrics used to evaluate the failure detection methods. The metrics’ formulas are defined as follows:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{3}$$

$$FPR = \frac{FP}{FP + TN} \tag{4}$$

$$FNR = \frac{FN}{FN + TP} \tag{5}$$

$$Precision = \frac{TP}{TP + FP} \tag{6}$$

$$Recall = \frac{TP}{TP + FN} \tag{7}$$

$$\text{Specificity} = \frac{TN}{TN + FP} \tag{8}$$

$$\text{F} - \text{score} = \frac{2 \times Precision \times Recall}{Precison + Recall} \tag{9}$$

$$\text{G} - \text{mean} = \sqrt{Recall \times \text{Specificity}} \tag{10}$$

$$\text{Balance} = \frac{Recall + Specificity}{2} \tag{11}$$

$$MCC = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP + FP) \times (FN + TN) \times (FP + TN) \times (TP + FN)}} \tag{12}$$

Where TP refers to the number of failed tracing data samples that are correctly detected as a failure, FP refers to the number of normal tracing data samples that are incorrectly detected as a failure, FN refers to the number of failed tracing data samples that are incorrectly not detected as a failure, and TN refers to the number of normal tracing data samples that are correctly detected as normal.

It is worth mentioning that due to a large number of tracing data samples, reducing the false-positive rate is very important to avoid wasting time to re-check the tracing data samples manually. Also, reducing the false-negative rate and detecting most failed tracing data samples is important to maintain the reliability of the system. On the other hand, in a real system, the rate of failed tracing data samples is much lower than normal tracing data samples. So that the number of normal and failed tracing data samples is imbalanced. Therefore, we use the MCC metric to evaluate the failure detection model more accurately.

Table 4
The number of tracing data samples for training and testing the failure detection model.

The number of training data samples	The number of normal test data samples	The number of failed test data samples
800	197	39

Table 5
The values of the hyperparameters used to train the failure detection model.

The number of epochs	Batch Size	Learning Rate	Dropout probability
500	64	0.001	0.1

The MCC metric is suitable for evaluating two-class classifications with imbalanced data. In this way, it indicates the dependence between the observed values of a binary class and its detected values. The value of the MCC metric varies between −1 and 1. Thus, the value of +1 indicates an accurate detection of the detection method, the value of zero indicates a random detection, and the value of −1 indicates a complete mismatch between the detected and the real values. Therefore, the closer the value of the MCC metric to one indicates the better performance of the detection model. Also, we use F-score, G-mean, and Balance metrics to overcome the imbalance between the false positives and the false negatives.

Speed is another metric that we used to evaluate the failure detection method. Considering that some steps of the failure detection process are done online, they should have an acceptable speed. We calculated the speed metric for each step of the failure detection process in the form of the execution time of the desired step or its time complexity.

5.5. Evaluation procedure

In this section, according to the process of the proposed method, we describe the procedure of evaluation and conducting experiments. We implemented all three phases of the proposed method on a system with 12 GB RAM, Intel Core i7-6700HQ 2.60 GHz processor, and Windows 10 64-bit OS.

According to section 4, we proposed the Bi-LSTM layer and Adam optimizer to train the failure detection model. But, in the basic method, the use of an LSTM layer and stochastic gradient descent optimizer is proposed. On the other hand, the third type of recurrent neural network is GRU. Therefore, to evaluate the proposed method and compare it with the basic method and other type of recurrent neural network, we separately considered 6 different modes for training 6 failure detection models, which are: 1) Bi-LSTM layer and stochastic gradient descent optimizer, 2) Bi-LSTM layer and Adam optimizer, 3) LSTM layer and stochastic gradient descent optimizer, 4) LSTM layer and Adam optimizer, 5) GRU layer and stochastic gradient descent optimizer, and 6) GRU layer and Adam optimizer. Also, to train and test all 6 models, we used the same tracing data set. The number of tracing data samples for training and testing is given in Table 4.

To train the failure detection model with acceptable performance, we tuned the values of the hyperparameters manually and according to the values of the hyperparameters used in the basic method. The values of the hyperparameters used to train the failure detection model are given in Table 5.

To evaluate the third phase of the proposed method, according to the experiments performed in the DeepLog (Du et al., 2017), we first trained the initial model with the first 25% of the training data. Then, we updated the model in 3 steps using the next 25% of the training data in each step and compared the performance of the model before and after the update. To evaluate the third phase, we used the proposed model including the Bi-LSTM layer and Adam optimizer. The values of the hyperparameters used to perform the third phase are according to Table 5.

Table 6
The time complexity of each sub-step of the preprocessing step.

Extracting events from tracing data sample	Assigning a unique label to each event	Sorting of events according to timestamp	Events sequence padding to fix the length
O(n)	O(1)	O(nlogn)	O(n)

5.6. Results and comparison

In the following, we report the results of the evaluation and answer the research questions according to the results.

5.6.1. Evaluation of the preprocessing step

The preprocessing step is performed in phases 1 and 2. The purpose of the preprocessing step is to map a tracing data sample to its corresponding numerical vector as the input of the sequence-to-sequence multiclass classification model. Therefore, it does not affect the accuracy of the failure detection model. But due to that the failure detection phase is done online; its steps should have an acceptable speed.

The sub-steps of the preprocessing step are performed sequentially according to Fig. 7. Therefore, the time complexity of the preprocessing step is equal to the maximum execution time of its sub-steps. The time complexity of each sub-step of the preprocessing step is given in Table 6.

According to Table 6, the time complexity of the preprocessing step is equal to O(nlogn). Therefore, the preprocessing step is suitable for online failure detection in terms of time complexity.

5.6.2. Evaluating the performance of the trained models to detect failures

In this section, we first describe how to train failure detection models in minimizing the value of the loss function. Then, we compare the performance of 6 trained models in failure detection according to the metrics introduced in section 5.4. Figs. 18–23 show how to train all 6 failure detection models in minimizing the value of the loss function.

As shown in Figs. 18–23; the value of loss of all 6 failure detection models has decreased during training and has been minimized to a value of less than one. If the value of the cross-entropy loss function in the trained model is less than 0.3; the trained model will have acceptable performance in failure detection. If the value of the cross-entropy loss function is in the range (0.3, 1); the trained model will not work very well and, if the value of the cross-entropy loss function is greater than one, the trained model will not be suitable for failure detection (Wang et al., 2022). The minimum value of the loss function obtained for each of the 6 failure detection models in the training process is given in Table 7.

According to Table 7, the trained model using the proposed method and Adam optimizer has converged to the lowest value of the loss function equal to 0.23; therefore, it will have a better performance in failure detection than other trained models. On the other hand, the trained model using the basic method and stochastic gradient descent optimizer with the value of the loss function equal to 0.5 will have worse performance than other trained models. The evaluation results of the performance of each trained model in failure detection are given in Table 8.

As shown in Table 8; the recall metric for all 6 failure detection models is 100% and the false-negative rate is zero. Therefore, all 6 failure detection models correctly detect failed trace data samples. Because all 6 failure detection models are trained on normal trace data and can detect failed trace data samples.

According to Table 8, the trained model using the basic method and the stochastic gradient descent optimizer has the highest false-positive rate and the lowest accuracy. Also, the trained model using the basic method and Adam optimizer with a slight improvement compared to the trained model using the basic method and the stochastic gradient descent optimizer still has a high false-positive rate and low accuracy. Therefore, the use of the Adam optimizer can improve the performance of the failure detection model due to the convergence of the loss function to a lower value during training. But the main reason for the high false-positive rate and low accuracy of the trained model using the basic method is the use of the LSTM layer and the calculation of the probability of occurrence of each event in each position of the tracing data sample considering its previous events. Also, as shown in Table 8; the trained model using GRU has achieved results similar to LSTM because GRU calculates the probability of occurrence of each event in each

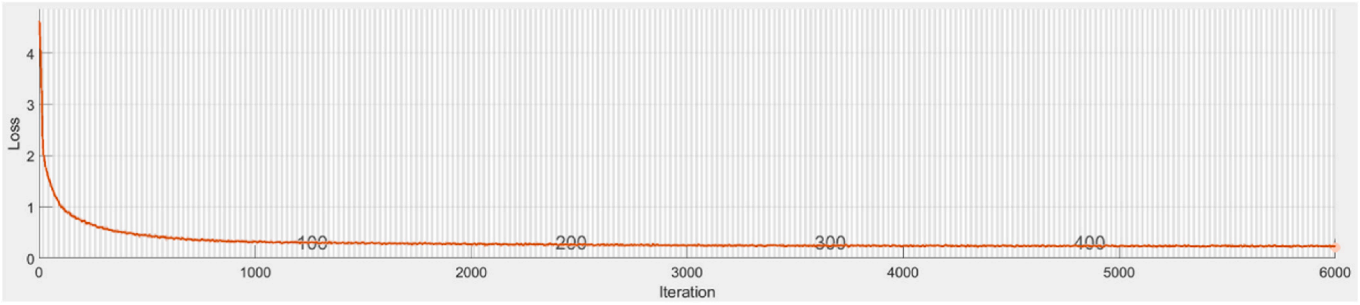

Fig. 18. Training the failure detection model of the proposed method using Adam optimizer to minimize the value of the loss function.

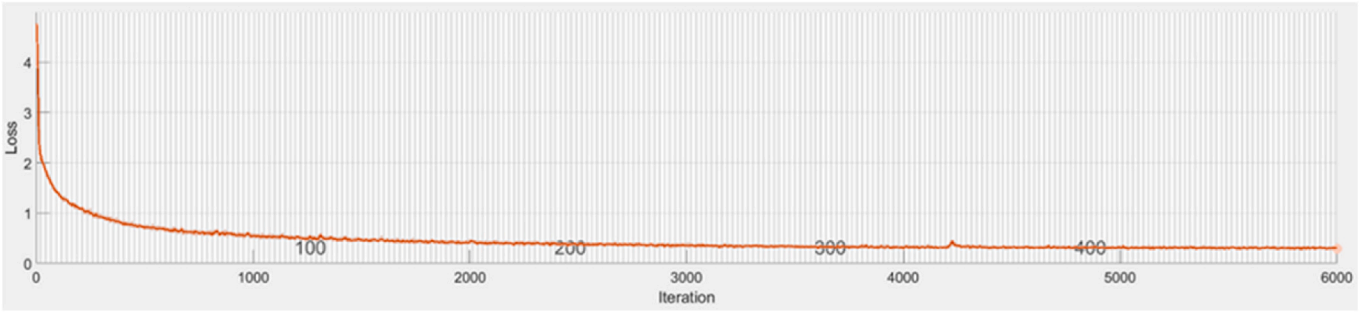

Fig. 19. Training the failure detection model of the proposed method using the stochastic gradient descent optimizer to minimize the value of the loss function.

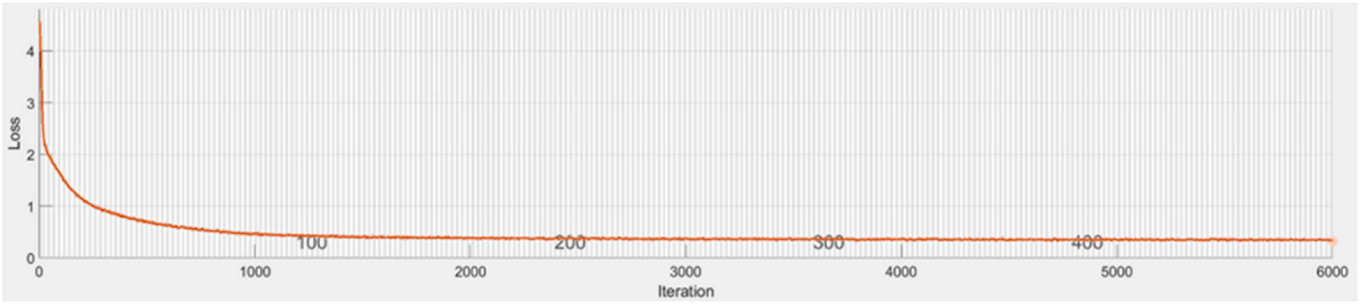

Fig. 20. Training the failure detection model of the basic method using Adam optimizer to minimize the value of the loss function.

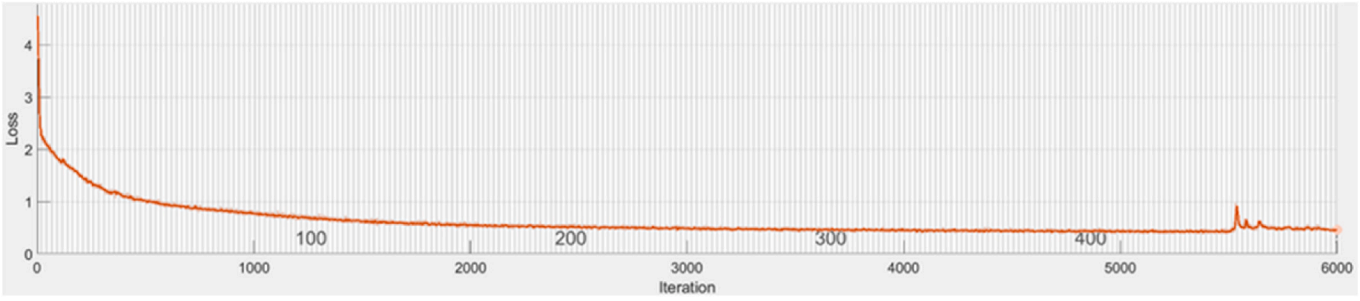

Fig. 21. Training the failure detection model of the basic method using the stochastic gradient descent optimizer to minimize the value of the loss function.

position of the tracing data sample considering its previous events like LSTM. Therefore, according to Figs. 11 and 12, a normal tracing data sample is incorrectly detected as a failure. As shown in Table 8; we improve the performance of the failure detection model in terms of decreasing the false-positive rate and increasing the accuracy using the Bi-LSTM layer and calculating the probability of occurrence of each event in each position of the tracing data sample according to its previous and next events. So, the trained model using the proposed method and Adam optimizer with a false-positive rate of 0.02 and an accuracy of 98% has the best performance among the 6 trained models.

Also, the trained model using the proposed method and Adam optimizer has achieved the best performance in terms of Precision, Specificity, F-score, G-mean, Balance, and MCC metrics among the 6 trained models. Therefore, the performance of the trained model using the proposed method has been significantly improved compared to the basic method in correctly distinguishing between normal and failed trace data samples.

Considering that failure detection is done online; the failure detection time is very important. The training time and average detection time of 6 trained models are given in Table 9.

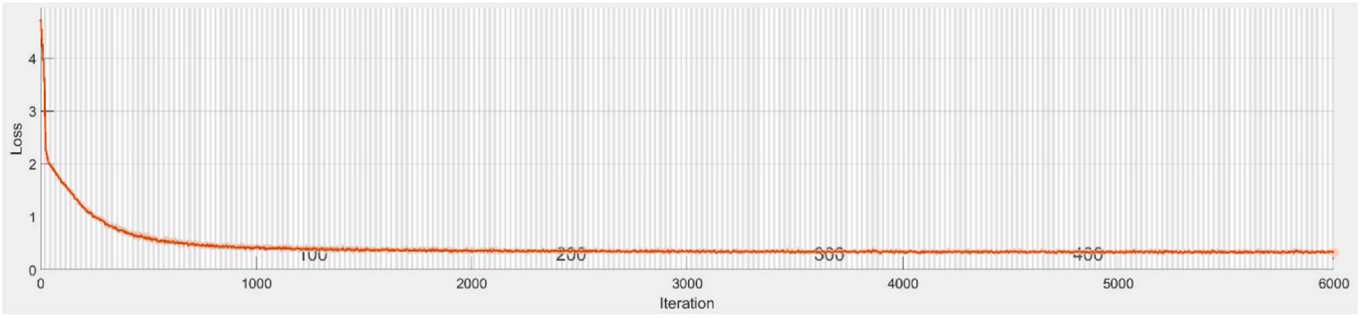

Fig. 22. Training the failure detection model using GRU and Adam optimizer to minimize the value of the loss function.

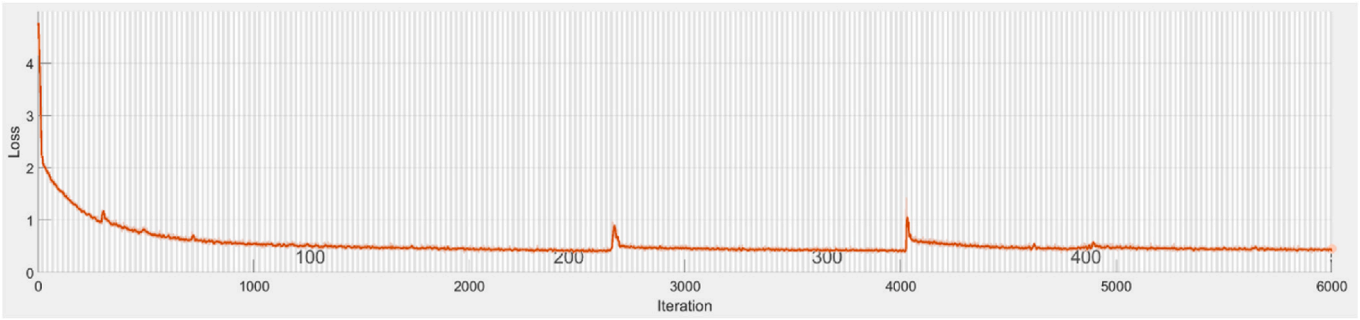

Fig. 23. Training the failure detection model using GRU and the stochastic gradient descent optimizer to minimize the value of the loss function.

Table 7
The minimum value of the obtained cross-entropy loss for 6 failure detection models in the training process.

Value	Failure detection model					
	The proposed method using Adam optimizer	The proposed method using the stochastic gradient descent optimizer	The basic method using Adam optimizer	The basic method using the stochastic gradient descent optimizer	GRU using Adam optimizer	GRU using the stochastic gradient descent optimizer
The obtained minimum loss value	0.23	0.31	0.37	0.5	0.35	0.36

Table 8
Evaluation results of all trained models in the failure detection.

The trained model	Metric									
	Accuracy	FPR	FNR	Precision	Recall	Specificity	F-score	G-mean	Balance	MCC
The proposed method using Adam optimizer	98%	0.02	0	90%	100%	98%	95%	99%	99%	0.94
The proposed method using the stochastic gradient descent optimizer	75%	0.29	0	40%	100%	70%	57%	83%	85%	0.53
The basic method using Adam optimizer	33%	0.8	0	20%	100%	19%	33%	43.5%	59.5%	0.19
The basic method using the stochastic gradient descent optimizer	25%	0.89	0	18%	100%	10%	32%	31.5%	55%	0.14
GRU using Adam optimizer	34%	0.78	0	20%	100%	21%	33%	45.8%	60.5%	0.21
GRU using the stochastic gradient descent optimizer	32%	0.8	0	19%	100%	19%	32%	43.5%	59.5	0.19

As shown in Table 9, the training time of the proposed method is almost 2 times the training time of the basic method. Because the proposed method uses the Bi-LSTM layer for training and the Bi-LSTM layer consists of two LSTM layers in two opposite directions. Therefore, the proposed method needs twice the time for training than the basic method. But due to that the training phase is done offline; there is no problem with the time of training. Also, the training time of the model using GRU is less than LSTM because GRU has fewer parameters than LSTM. The average detection time for all 6 failure detection models is less than 1 s. Therefore, all 6 failure detection models are suitable for online failure detection.

5.6.3. *Evaluation of the impact of updating the trained model*

In this section, we describe how to train and update the failure detection model using the proposed method and Adam optimizer in minimizing the value of the loss function. Then, we will compare the performance of the trained model in failure detection in different updating steps. Figs. 24–27 show how to train and update the failure detection model in minimizing the value of the loss function.

According to Fig. 24, the value of the loss function while training the failure detection model using the first 25% of the training set has converged to a value of 0.33. Therefore, the trained model will still not have an acceptable performance. According to Figs. 25–27, the failure detection model is updated using the weights obtained for the previous model. Therefore, the initial value of the loss function in each update is

Table 9
Training time and average detection time of 6 trained models.

The trained model	Time	
	Training time	Average detection time
The proposed method using Adam optimizer	32 min	19 ms
The proposed method using the stochastic gradient descent optimizer	33 min	23 ms
The basic method using Adam optimizer	18 min	12 ms
The basic method using the stochastic gradient descent optimizer	19 min	15 ms
GRU using Adam optimizer	14 min	18 ms
GRU using the stochastic gradient descent optimizer	15 min	19 ms

equal to the value obtained from the previous step. So, in the last update, the value of the loss function reached the minimum value of 0.23.

In Table 10, the evaluation results of each trained model during the update are given to evaluate the impact of updating the failure detection model using the new tracing data set.

As shown in Table 10; the recall metric in each update step is 100% and the false-negative rate is 0. Therefore, the failure detection model can correctly detect the failed trace data samples. The reason for the acceptable performance of the failure detection model according to the recall and the false-negative rate is that the failure detection model is trained on normal tracing data samples and can detect failed trace data samples. But other performance metrics such as Accuracy, False-positive rate, Precision, Specificity, F-score, G-mean, Balance, and MCC are improved in every update. Therefore, due to the frequent updates of a microservice-based system and the creation of new call paths between microservices, the training set will be completed over time. So, updating the trained model intermittently maintains the performance of the failure detection model.

5.6.4. Comparison

In this section, to demonstrate the improvement of the proposed method compared to other failure detection methods, we compare the proposed method with other methods in addition to comparing with the basic method and the GRU neural network in Section 5.6.2. Considering that the proposed method is an unsupervised learning method, we selected three methods (Meng et al., 2020; Zhang et al., 2022; Chen et al., 2023) among the available methods that used unsupervised learning according to Table 1. The comparison results are given in Table 11.

As shown in Table 11, we achieved 100% of Recall and False-Negative Rate of 0 in the proposed method. So, we correctly detect all

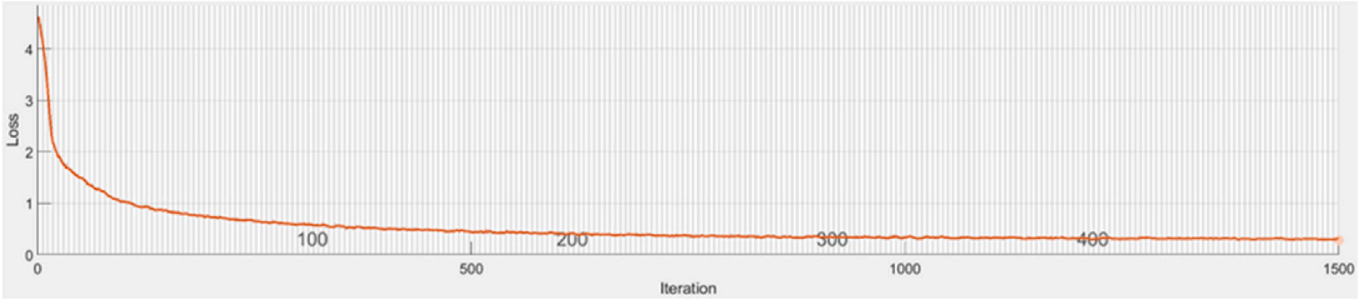

Fig. 24. Training the failure detection model in minimizing the value of the loss function using the first 25% of the training set.

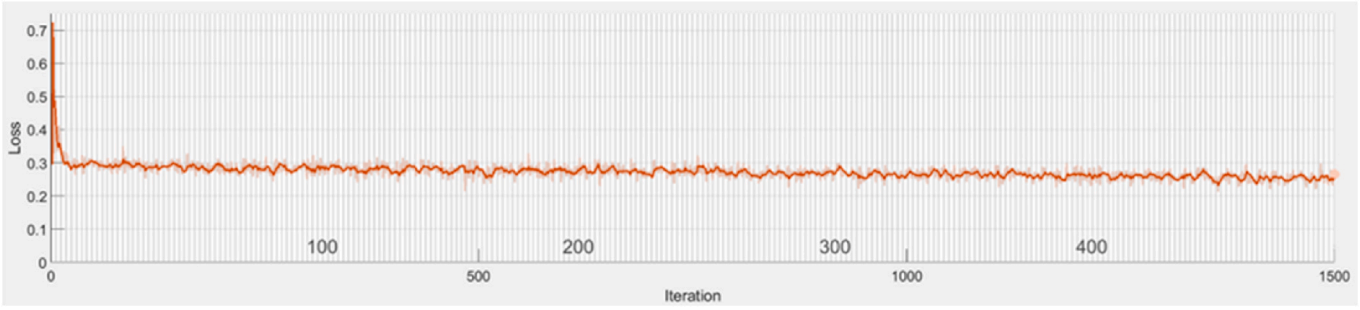

Fig. 25. First update of the failure detection model in minimizing the value of the loss function using the second 25% of the training set.

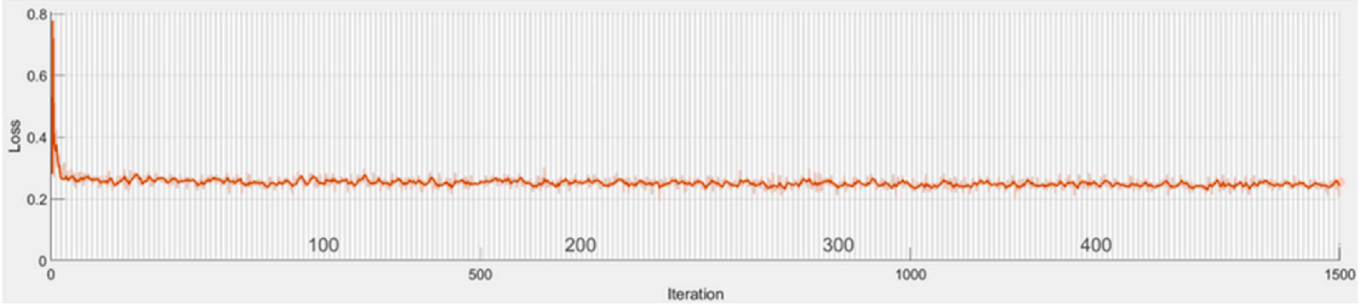

Fig. 26. The second update of the failure detection model in minimizing the value of the loss function using the third 25% of the training set.

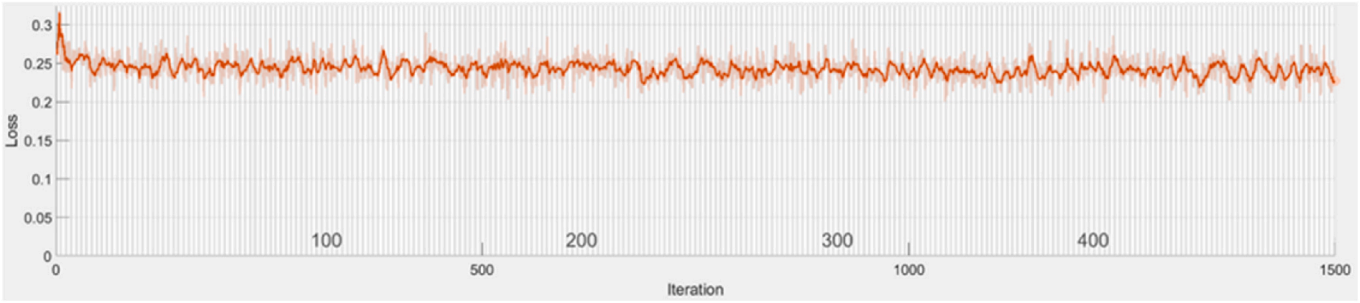

Fig. 27. The third update of the failure detection model in minimizing the value of the loss function using the fourth 25% of the training set.

Table 10
Comparing the performance of the trained model after each update.

Step	Metric									
	Accuracy	FPR	FNR	Precision	Recall	Specificity	F-score	G-mean	Balance	MCC
Initial training	80%	0.23	0	46%	100%	76%	63%	87%	88%	0.59
First update	89%	0.12	0	60%	100%	87%	76%	93%	93.5%	0.73
Second update	95%	0.05	0	78%	100%	94%	88%	97%	97%	0.86
Third update	98%	0.02	0	89%	100%	98%	94%	99%	99%	0.93

Table 11
The comparison results of the proposed method with the selected failure detection methods.

Method	Metric									
	Accuracy	FPR	FNR	Precision	Recall	Specificity	F-score	G-mean	Balance	MCC
The proposed method	98%	0.02	0	90%	100%	98%	95%	99%	99%	0.94
Midiag (Meng et al., 2020)	94%	0.019	0.1	91%	90%	98%	90.4%	94%	94%	0.91
DeepTraLog (Zhang et al., 2022)	96%	0.018	0.03	93%	97%	97%	95%	97%	97%	0.93
TraceGra (Chen et al., 2023)	95%	0.015	0.07	96%	93%	97%	94%	95%	95%	0.93

failed trace data samples against other failure detection methods. The mentioned comparative methods feed trace data samples to the trained model to calculate the failure score of the corresponding data sample and compare it with the threshold. So, the performance of the failure detection of the comparative methods depends on the threshold. If the value of the threshold is considered too large, it increases the false-negative rate, and if it is considered too small, the false-positive rate increases. Also, Midiag (Meng et al., 2020) and TraceGra (Chen et al., 2023) train a distinct model for each pattern of microservices' call path. Therefore, due to the large number of microservices, the complexity of the communication between them, and the call paths with different patterns, many failure detection models should be trained; this causes overhead during training and detecting failures. So, we achieved improvement in terms of performance and simplicity against available failure detection methods.

5.6.5. Response to research questions

In the following, we answer the research questions according to the results reported in Tables 8 and 10.

RQ1: Can the proposed method detect failures in microservices' call paths with a low false-positive rate considering different types of call paths compared to the basic method?

As mentioned in Section 5.2, tracing data samples generated by VWR consist of different types of microservices' call paths. According to Table 8, the proposed method has achieved an accuracy of 98% and a false-positive rate of 0.02 to detect failures. So the proposed method is able to detect the failure in different types of microservices' call paths with a low false-positive rate.

RQ2: Can the proposed method maintain the performance of the failure detection model according to frequent updates of microservice-based systems to cover less or new call paths?

To evaluate the impact of updating the trained model, we divided the training data set into four parts and performed experiments according to DeepLog (Du et al., 2017). So that we first trained the initial model with the first 25% of the training data. Then, we updated the model in 3 steps using the next 25% of the training data. According to Table 10, the performance of the failure detection model has been improved in updating steps. So, the proposed method can improve and maintain the performance of the failure detection model according to frequent updates of microservice-based systems to cover less or new call paths with updating the trained model periodically.

5.7. Threats to validity

Validity is a key challenge for researchers and practitioners in conducting empirical research work. Therefore, in this section, we discuss possible threats to construct validity, internal validity, external validity, and reliability validity as follows:

Construct validity: Threats to construct validity refer to the relationship between theory and measures. There could be inaccuracies and omissions in the measurements for several reasons. Considering that in microservice-based systems, there are various call paths; to increase the accuracy and reduce the false-positive rate, we used the Bi-LSTM neural network to calculate the probability of occurrence of each event in each position of the tracing data sample according to its previous and next events. On the other hand, there may be microservice-based systems with different and complex architectures. So, we cannot claim that the proposed method has similar results to detect failure in microservice-

based systems with more complex architectures. To solve this problem, we considered that in microservice-based systems, call paths are a combination of one or more basic call paths. So, the microservice-based system architecture that we selected to simulate in the VWR benchmark has a variety of sequential, concurrent, alternate, and loop call paths. Therefore, we were able to compare the improvement of the proposed method with the basic method in detecting a failure in all types of call paths. Also, we calculated the accuracy of the proposed method according to the different types of basic call paths.

Internal validity: Threats to internal validity concern conditions that influence the confidence level of results. In this context, we formulated the failure detection problem as an anomaly detection problem. Therefore, to train the failure detection model, we assumed that the system works normally mostly and the failure rate in the system is much lower than the normal execution of the system. On the other hand, frequent updates in the microservice-based systems cause new features, including a new microservice or new call paths between microservices. Therefore, it causes the detection of new call paths as a failure and increases the false-positive rate. So, to cover new call paths, we periodically update the trained model.

External validity: External validity threats concern the generalization of results. Considering that to evaluate the proposed method, we did not have access to the distributed tracing data obtained from the execution of a real system; so, we performed the experiments using distributed tracing data obtained from the execution of a microservice-based system in the VWR benchmark. So that the VWR benchmark is a standard benchmark in the field of failure detection in microservice-based systems. Another limitation was the generation of failed trace data samples to evaluate the performance of the proposed method in distinguishing between failed and normal tracing data samples. In the VWR benchmark, it is possible to generate a failed trace data sample by injecting the deletion fault into each of the microservices' instances. But the failure in the call path of microservices can happen due to different reasons and under different scenarios; according to the basic method, we randomly generated failures in normal tracing data samples. But still, to evaluate the proposed method, there is a need to generate failed trace data samples in different call paths under different scenarios.

Another threat to the external validity of this study is related to the evaluation of the impact of the update phase to improve and maintain the performance of the failure detection model. Considering that, we did not have access to the tracing data obtained from the execution of a real system and the creation of new call paths under the system update; according to the experiments performed in the other studies, we divided the training data set into 4 parts and evaluated the effect of updating the trained model on the performance of failure detection. But it is still necessary to investigate the effect of the update phase to improve and maintain the performance of the detection model on new call paths.

Reliability validity: It is related to the reproducibility of the study to achieve similar results in similar conditions. So, we described the process of the proposed method and the process of performing the experiments in a reproducible way. In this way, the evaluation process has been described in full detail.

6. Conclusion and future work

Microservice architecture is the latest trend in the design and development of software systems based on modularization. Considering the importance of monitoring and maintaining a microservice-based system, in the current study, we improved the process of failure detection in such systems. There are challenges to detecting failures in microservice-based systems due to the inherent characteristics of such systems, including complex communications, frequent updates, dynamicity at runtime, and complex log management. In a microservice-based system, to overcome the challenge of complex communication and dynamicity at runtime in failure detection, distributed tracing data is used. By using distributed tracing data, the call path between microservices to respond to a request is obtained. A call path may be faulty from two perspectives: 1) The microservices call sequence in the call path may be incorrect, and 2) The response time of a microservice in a call path may be out of expectation. So that in this study, we focused on detecting the failure in microservices' call paths.

According to the purpose of this study to improve the performance of failure detection in terms of increasing the accuracy and reducing the false-positive rate in the microservices' call paths using distributed tracing data, we first reviewed the related works. Then, we selected one of the best existing works as the basic method and improved it. For this purpose, we designed a process in three phases. The process phases of the proposed method are 1) training and evaluating the failure detection model, 2) online failure detection, and 3) updating the failure detection model. In the training and evaluating phase of the failure detection model, we modeled the failure detection problem as a sequence-to-sequence multiclass classification problem and used the tracing data set obtained from the normal execution of the system as a training data set. In the second phase, we detected the failure of the tracing data samples obtained from the execution of the system online by using the trained model. Finally, we performed the third phase intermittently to update the failure detection model to cover new or less frequent call paths using the new tracing data set. We used the VWR benchmark to evaluate the proposed method. Using the VWR benchmark, tracing data from the execution of a microservice-based system can be obtained. The evaluation results of the proposed method show that with 98% Accuracy, 90% Precision, 100% Recall, 98% Specificity, 95% F-score, 99% G-mean, 99% Balance, MCC of 0.94, a false-negative rate of 0, and a false-positive rate of 0.02, we achieved a significant improvement over the basic method.

In the future, we are interested to propose a comprehensive method for detecting both types of detectable failures using distributed tracing data. So that the failure detection model detects the failure in the microservices' call path and the failure in the response time of the call path.

CRediT authorship contribution statement

Zahra Purfallah Mazraemolla: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Formal analysis, Conceptualization. **Abbas Rasoolzadegan:** Supervision, Project administration, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors do not have permission to share data.

References

Balalaie, A., Heydarnoori, A., Jamshidi, P., 2016. Microservices architecture enables DevOps: migration to a cloud-native architecture. IEEE Software 33 (3), 42–52.
Banerjee, K., et al., 2020. Exploring Alternatives to Softmax Function.
Bellur, U.A., 2007. Methodology & Tool for determining inter-component dependencies Dynamically in J2EE environments. In: Third International Conference on Autonomic and Autonomous Systems. ICAS'07).
Bischl, B., et al., 2023. Hyperparameter optimization: foundations, algorithms, best practices, and open challenges. WIREs Data Mining and Knowledge Discovery 13 (2), e1484.
Bogatinovski, J., et al., 2020. Self-supervised anomaly detection from distributed traces. In: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC), pp. 342–347.
Brandón, Á., et al., 2020. Graph-based root cause analysis for service-oriented and microservice architectures. J. Syst. Software 159, 110432.

Chandola, V., Banerjee, A., Kumar, V., 2009. Anomaly detection: a survey. ACM Comput. Surv. 41 (3).
Chen, H., Chen, P., Yu, G., 2020. A framework of virtual war room and matrix sketch-based streaming anomaly detection for microservice systems. IEEE Access 8, 43413–43426.
Chen, J., et al., 2023. TraceGra: a trace-based anomaly detection for microservice using graph deep learning. Comput. Commun. 204, 109–117.
Chung, J., et al., 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.
Cinque, M., Corte, R.D., Pecchia, A., 2022. Microservices monitoring with event logs and black box execution tracing. IEEE Transactions on Services Computing 15 (1), 294–307.
Dahouda, M.K., Joe, I., 2021. A deep-learned embedding technique for categorical features encoding. IEEE Access 9, 114381–114391.
Data Model. Available from:: https://zipkin.io/pages/data_model.html.
Du, M., et al., 2017. DeepLog: anomaly detection and diagnosis from system logs through deep learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, Dallas, Texas, USA, pp. 1285–1298.
Gan, Y., et al., 2019. 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. Association for Computing Machinery, Providence, RI, USA, pp. 19–33.
Gan, Y., et al., 2021. Sage: Using Unsupervised Learning for Scalable Performance Debugging in Microservices.
Guo, X., et al., 2020. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery: Virtual Event, USA, pp. 1387–1397.
Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.
Jamshidi, P., et al., 2018. Microservices: the journey so far and challenges ahead. IEEE Software 35 (3), 24–35.
Jin, M., et al., 2020. An anomaly detection algorithm for microservice architecture based on robust principal component analysis. IEEE Access 8, 226397–226408.
Kim, M., Sumbaly, R., Shah, S., 2013. Root cause detection in a service-oriented architecture. SIGMETRICS Perform. Eval. Rev. 41 (1), 93–104.
Kingma, D.P., Ba, J., 2015. Adam: A Method for Stochastic Optimization.
Li, Z., et al., 2021. Practical root cause localization for microservice systems via trace analysis. In: 2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS).
Liu, P., et al., 2020. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE).
Liu, D., et al., 2021. 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), pp. 338–347.
Mariani, L., et al., 2018. Localizing faults in cloud systems. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST).
Meng, L., Sun, Y., Zhang, S., 2020. Midiag: a sequential trace-based fault diagnosis framework for microservices. In: Services Computing – SCC 2020: 17th International Conference, Held as Part of the Services Conference Federation, SCF 2020, Honolulu, HI, USA, September 18–20, 2020, Proceedings. Springer-Verlag, Honolulu, HI, USA, pp. 137–144.
Meng, L., et al., 2021. Detecting anomalies in microservices with execution trace comparison. Future Generat. Comput. Syst. 116, 291–301.
Mishra, A.D., Garg, D., n.d.. Selection of best sorting algorithm. Int. J. Intell. Inf. Process. 2, 363–368.
Nedelkoski, S., Cardoso, J., Kao, O., 2019a. Anomaly detection and classification using distributed tracing and deep learning. In: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID).
Nedelkoski, S., Cardoso, J., Kao, O., 2019b. Anomaly detection from system tracing data using multimodal deep learning. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD).
Pirani, M., et al., 2022. A comparative analysis of ARIMA, GRU, LSTM and BiLSTM on financial time series forecasting. In: 2022 IEEE International Conference on Distributed Computing and Electrical Circuits and Electronics (ICDCECE).
Scheinert, D., et al., 2021. Learning dependencies in distributed cloud applications to identify and localize anomalies. CoRR, 05245 abs/2103.
Schuster, M., Paliwal, K.K., 1997. Bidirectional recurrent neural networks. IEEE Trans. Signal Process. 45 (11), 2673–2681.
Skrobek, D., et al., 2022. Implementation of deep learning methods in prediction of adsorption processes. Adv. Eng. Software 173 (C).
Soldani, J., Brogi, A., 2022. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: a survey. ACM Comput. Surv. 55 (3). Article 59.
Srivastava, N., et al., 2014. Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. 15 (1), 1929–1958.
Thones, J., 2015. Microservices. IEEE Softw 32 (1), 113–116.
Wang, P., et al., 2018. CloudRanger: root cause identification for cloud native systems. In: 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID).
Wang, Q., et al., 2022. A comprehensive survey of loss functions in machine learning. Annals of Data Science 9 (2), 187–212.
Wu, L., et al., 2020. MicroRCA: root cause localization of performance issues in microservices. In: NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium.
Yoon, S.-H., Yu, H.-J., 2020. A simple distortion-free method to handle variable length sequences for recurrent neural networks in text dependent speaker verification. Appl. Sci. 10 https://doi.org/10.3390/app10124092.
Yu, G., et al., 2021. MicroRank: end-to-end latency issue localization with extended spectrum analysis in microservice environments. In: World Wide Web Conference, pp. 3087–3098. ACM/IW3C2.
Zhang, C., et al., 2022. DeepTraLog: trace-log combined microservice anomaly detection through graph-based deep learning. In: 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE).
Zhou, X., et al., 2019. 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. Association for Computing Machinery, Tallinn, Estonia, pp. 683–694.
Zipkin API. Available from: https://zipkin.io/zipkin-api/#/default/post_spans.