Received February 20, 2020, accepted February 26, 2020, date of publication March 2, 2020, date of current version March 12, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.2977464*

A Framework of Virtual War Room and Matrix Sketch-Based Streaming Anomaly Detection for Microservice Systems

HONGYANG CHEN, PENGFEI CHEN, AND GUANGBA YU



School of Data and Computer Science, Sun Yat-sen University, Guangzhou 510006, China

Corresponding author: Pengfei Chen (chenpf7@mail.sysu.edu.cn)

The work was supported in part by the National Key Research and Development Plan under Grant 2019YFB1804002, in part by the National Natural Science Foundation of China under Grant 61802448, Grant 6190244, and Grant U1811462, and in part by the Program for Natural Science Foundation of Guangdong under Grant 2019A1515012229.

 **ABSTRACT** Recently, microservice has been a popular architecture to construct cloud-native systems. Thisnovel architecture brings agility and accelerates the software development process signi cantly. However, it is not easy to manage and operate microservice systems due to their scale and complexity. Many approaches are proposed to automatically operate microservice systems such as anomaly detection. Nevertheless, those methods cannot be suf ciently validated and compared due to a lack of real microservice systems, which leads to the slow process of intelligent operation. These challenges inspire us to build a system named ‘‘VWR’’, a framework of Virtual War Room for operating microservice applications which allows users to simulate their microservice architectures with low overhead and inject multiple types of faults into the microservice system with chaos engineering. VWR can mimic user requests and record the end-to-end tracing data (i.e., service call chains) for each request in a way consistent with OpenTracing. With easily designed tests and the produced streaming tracing data, the users can validate the performance of their intelligent operation algorithms and improve the algorithms as needed. Besides, based on the streaming tracing data generated by VWR, we introduce a novel unsupervised anomaly detection algorithm based on Matrix Sketch and set it as a default intelligent operation algorithm in VWR. This algorithm can detect anomalies by analyzing high-dimensional performance data collected from a microservice system in a streaming manner. The experimental result in VWR shows that the matrix sketch based method can precisely detect anomalies in microservice systems and outperform some widely used anomaly detection methods such as isolation forest in some scenario. We believe more approaches on the intelligent operation of microservice systems can be constructed based on VWR.



 **INDEX TERMS** Microservice, virtual war room, matrix sketch, anomaly detection, chaos engineering.



**I. INTRODUCTION**

Nowadays, the microservice has become a widely adopted architecture for enterprises to deploy their large cloud-native applications. With the microservice architecture, a large soft-ware application is decoupled into a suite of independent, distributed, and loosely coupled services that can be allowed to implement in different programming languages and can also be managed by different teams [1]. However, we have to notice that no matter which architecture we use to deploy our application, faults and problems are inevitable due to

The associate editor coordinating the review of this manuscript and approving it for publication was Dong Wang .



software bugs or runtime environment changes [2], [3]. What is worse, if a fault happens in a microservice architecture, it is more dif cult for operators to locate root causes due to the increasing complexity of service dependencies. Any single problem in a service such as a hard disk fault or a network disconnection may cause the microservice system to break-down. In order to detect anomalies, infer the root cause and restore the failed service as soon as possible, operators rely on tracing systems such as X-trace [4], Zipkin [5] to pro le and monitor the microservice architecture. These tracing systems can record the execution path of each request. With these information, the complexity of a microservice system can be managed.

|  |  |  |  |
| --- | --- | --- | --- |
| VOLUME 8, 2020 | This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/ | 43413 |  |
|  |  |

H. Chen *et al.*: Framework of VWR and MS-Based Streaming Anomaly Detection for Microservice Systems

Extensive studies have been done to resolve performance problems in distributed systems such as anomaly detection [6] [8], root cause analysis [2], [9] [11]. In these studies, the researchers need to deploy their own microservice bench-marks and inject some faults into these systems in order to get abnormal cases. Otherwise, they need some existing performance data provided by commercial enterprises or communities. However, this approach poses some challenges for researchersV

**Very few microservice benchmarks.** It is always nec-essary to have microservice benchmarks to validate the ef ciency of intelligent operation methods. However, there are very few open source microservice benchmark-ing systems except some simple architectures, like Sock-shop [12]. It is time-consuming and even unfeasible for researchers to design and deploy their own microservice systems at scale.

**A high cost in distributed tracing.** A large scalemicroservice system always comprises numerous inter-connected services. To operate and understand their behavior, a distributed tracing which records the execu-tion paths of each request has been a default module. However, the tracing system ether requires code instru-mentation into the application and modi cations of the run-time environment which means that researchers need to be familiar with the benchmark and instrument codes on their own.

**Dif culty in fault injection.** In reality, the cases offaults and anomalies are fewer than normal cases. To col-lect more abnormal cases, we always rely on fault injec-tion to mimic faults. Therefore, some fault injection tools are needed. There are some methods and existed tools for the researchers to conduct a chaos engineer like Net ix ChaosMonky [13], [14], and Istio [15]. However, it is still an annoying task in microservice systems due to their scale and complexity.

**A large volume of chaos tracing data.** Although manyexisting real-world microservice systems have collected a large volume of tracing data, these tracing data are always chaos with a lot of noises and errors such as missing data, disordered event sequences, and incorrect measurement values. Due to these chaos, it is nontriv-ial to nd ef cient algorithms to operate microservice systems. Moreover, most of collected tracing data are not labeled with ‘‘normal’’ or ‘‘abnormal’’. These data are collected in a streaming manner which should be processed in time.

To address the aforementioned challenges and shortages of previous work, we propose Virtual War Room(VWR) based on Spigo,[1](#page2) a framework to mimic the behavior of the microservice architecture and to inject faults readily with chaos engineering [13], which has been open sourced in the Github. VWR[2](#page2) can deliver tracing data with labels

* Spigo: https://github.com/adrianco/spigo

2VWR backend: https://github.com/chenhy97/sysu\_VWR

43414

(i.e., normal and abnormal) to users and provide a platform[3](#page2) for users to test their intelligent operation algorithms with the produced data. VWR is inspired by the concept of ‘‘war room’’ which is a battle eld where the army leaders can get together to simulate any possible problems and propose the corresponding methods. In IT and Internet companies, there are also war rooms to resolve problems coming from online systems. Similarly, we construct a virtual war room without real systems to test the ef ciency and effectiveness of some intelligent operation algorithms. However, combing the concept of war room and the microservice, we nd it hard for us to simulate faults in microservice systems due to lack of microservice benchmarks and fault injection tools. Therefore, we build a digital version of microservice systems in VWR by simulating the behavior of every service in a microservice system and de ning the communication protocol between each service. By moving microservice from reality to the digital world, the researchers can be allowed to run any microservice systems as soon as possible and inject any faults to any service of the systems which can help them observe the action of the microservice and nd a way to detect and locate faults. After the simulation, by uploading their algorithms to the platform, the researchers can test the performance of their algorithms with the produced tracing data on VWR before applying them to a real system.

Anomaly detection is a typical kind of intelligent opera-tion. However, existing approaches have shortcomings to ana-lyze the high-dimensional streaming tracing data. In addition to building VWR, we leverage VWR to analyze the tracing data and study the performance of some anomaly detection algorithms. Moreover, we introduce an anomaly detection algorithm based on randomized Matrix Sketch to resolve the challenge brought by high-dimensional streaming trace data. After validating its effectiveness, we set it as a default anomaly detection strategy in VWR. The effectiveness of this anomaly detection algorithm is validated and compared against Isolation Forest and Support Vector Machine (SVM) based methods. The experiment results show that VWR can be a good simulation benchmark for microservice systems and Matrix Sketch outperforms other anomaly detection methods in a streaming detection mode.

Beyond anomaly detection, VWR can also be leveraged to validate other intelligent operation algorithms such as perfor-mance prediction, root cause analysis, and so on. However, these algorithms are out of scope of this paper. We will extend our research by considering root cause analysis in the future work.

Generally speaking, the contributions of this paper are three-foldV

We rstly propose the concept ‘‘Virtual War Room’’ which helps researchers to observe the behavior of the microservice system and to validate the effectiveness and performance of their intelligent operation algo-rithms.

3VWR frontend: https://github.com/chenhy97/VWR\_front

VOLUME 8, 2020

H. Chen *et al.*: Framework of VWR and MS-Based Streaming Anomaly Detection for Microservice Systems

We design and develop ‘‘VWR’’ to mimic real microser-vice systems digitally for simulation, tracing, chaos engineering and fault analysis. Moreover, the labeled tracing data produced by the platform can help the evo-lution of research on anomaly detection algorithms.

We introduce an unsupervised anomaly detection algo-rithm based on Matrix Sketch to detect anomalies in microservice systems. This algorithm works in a real-time mode and analyzes the high-dimension run-time data. It can achieve a comparative result.

This paper is organized as follows. In Section 2, we present the related work. In Section 3, we introduce the system design of VWR. Section 4 introduces the details of the anomaly detection method based on Matrix Sketch. Section 5 shows the results of the experimental evaluation. In Section 6, we conclude this paper.

**II. RELATED WORK**

Previous studies have attempted to use system simulator to gain knowledge on performance/resource trade-offs in sys-tems that care about request latency. There are two classical simulation systems that are close to VWR, namely, BigHouse

1. and qSim [17]. BigHouse is an event-driven queueing simulator targeting datacenter service simulation using pre-calculated workload models and system models. However, BigHouse only models each application as a single queue and some intra-microservice stages cannot be captured by a single queue, causing poor simulation of microservice sys-tems. To address the problem, qSim takes a different method by explicitly modeling each application’s execution phases, and accounting for queuing effects throughout execution. What is more, qSim supports user-de ned microservice architecture. However, the target of qSim and BigHouse is to get insight on performance of distributed systems and they are not able to produce large volume of labeled trac-ing data immediately. What is more, VWR provides chaos engineering for users which is another important aspect for research on arti cial operation algorithms on microservice systems.

Furthermore, arti cial operations on microservice systems such as anomaly detection and Root cause analysis are chal-lenging topics. Extensive approaches have been proposed to detect anomalies or to pinpoint root causes. However, there are still some challenges, such as the lack of benchmarks for evaluation experiments. In the following, we present the related work brie y.

**A. ROOT CAUSE ANALYSIS**

The main method of this area are based on end-to-end request tracing. There are lots of work and systems designing to instrument source code, collect tracing data, and nd out requests with long response latency such as Magpie [18], X-trace [4], Pinpoint [19], The Mystery Machine [20]. Based on the suf cient tracing data, [21], [22] pinpoints system prob-lems by comparing traces. Another main approach depends

VOLUME 8, 2020

on the dependency graph which does not need the tracing data. Microscope [10] depends on the system calls it inter-cepts on Network to build dependency graphs and use SLO metrics to nd the root cause. NetMedic [23] infers the root cause by constructing component dependencies with preset templates. Although, we do not introduce root cause analysis in this paper, VWR can provide a large volume of tracing data to validate the effectiveness and ef ciency of those root cause analysis approaches.

**B. ANOMALY DETECTION**

Anomaly detection is a well-studied topic and there are many approaches such as the method we introduce from

1. Reference [24] is an unsupervised technique that is suitable for the high-dimension data in real-time applica-tions. The state-of-art approach IForest [25] is an ef cient method that is not suitable for real-time scenarios. What is more, some other supervised approaches have been proposed for anomaly detection such as [26] and [27] which needs labeled data.
2. **MICROSERVICE BENCHMARKS FOR TRACE-BASED APPROACH**

For those approaches of root cause analysis or anomaly detec-tion on microservice systems, most of them have to rely on microservice benchmarks. There are several benchmarks being released from academia or industry, such as Cloudsuite [28], Tailbench [29], Sirius [30]. However, the limitation of these benchmarks is that they only pay attention on single-tier applications, or at most three-tiers services, which is different from the way microservice systems are deployed. DeathstarBench [31] focuses on deploying large-scale appli-cations with several different services, which allows us to study the performance effect that only exists in large-scale systems. What is more, DeathstarBench is designed with ve microservice systems integrated with tracing system which is helpful for the researches [32] on arti cial oper-ation algorithm based on tracing. However, because of the challenges mentioned in Section [I,](#page1) VWR is another impor-tant tool. Compared to them, VWR does not need to be deployed in a real distributed environment. With its help, you can run any microservice systems on one machine and conduct any experiments with chaos engineering with a very limited cost.

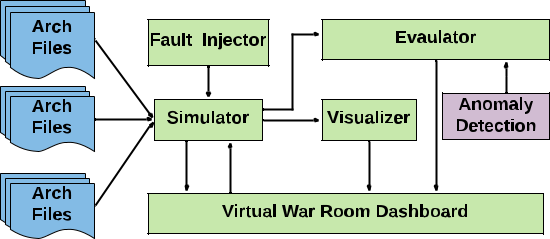
1. **SYSTEM DESIGN A. SYSTEM OVERVIEW**

The overall system design of VWR is showed in [Fig.1.](#page4) Once the architecture de nition le of a microservice system is uploaded by the user and running parameters including sim-ulation time are con gured, then VWR can be launched. The simulation starts while the tracing data are produced and stored in a local le. When the simulation starts, VWR can visualize the microservice architecture and help users run some algorithms to detect anomalies. The core functions of VWR are listed as followed.

43415

H. Chen *et al.*: Framework of VWR and MS-Based Streaming Anomaly Detection for Microservice Systems

**FIGURE 1.** Framework of virtual war room.



1) ARCH FILE

Fig.2 is three arch les for AWS, Net ix and LAMP which can help users to see how to de ne an architecture that can be simulated on VWR. In the arch le, users set the simulated architecture’s name by the ‘‘arch’’ parameter. Services of the simulated architecture are recorded by the ‘‘services’’ parameter. The ‘‘name’’ parameter records each service’s name, the ‘‘package’’ parameter stipulates the behavior each service takes during simulation and the ‘‘count’’ parameter records the number of each service. The connection between services is con gured by the ‘‘dependencies’’ parameter.

2) SIMULATION

In the simulation stage, VWR needs to parse the uploaded microservice architecture de nition les and set the running parameters for the simulation. After that, the simulator will simulate the behavior of important services in the microser-vice system such as a database, a load balancer and so on. The tracing data will be produced and stored to make further analysis.

3) FAULT INJECTION

During the simulation, the fault injection component will receive injection requests from users and inject faults to the simulation system.

4) VISUALIZATION

In the visualization stage, there are two primary tasks needed to be handled. First of all, when the microservice architecture le is parsed, the visualization component will visualize the topology of the architecture, which facilitates users if they need to inject faults. Moreover, when the tracing data is generated, the visualization component will convert it to time series that can help users understand the behavior of the simulated microservice and the effect of injected faults.

5) EVALUATION

In the evaluation stage, VWR will detect anomalies in the generated tracing data using the uploaded algorithms from the users. By default, VWR employs Matrix Sketch (MS) based method to detect anomaly due to the collected high-dimensional performance data. When the evaluator receives the tracing data, the evaluator will pre-process the tracing data and convert it to the format that MS needs.Matrix Sketch is an effective and online algorithm that can extract primary

43416

components of the continuously generated streaming data. The core of Matrix Sketch is PCA (Primary Components Analysis) [33] which consists of three parts. The rst one is encoding the original data ( i.e., a large matrix *M*) into a smaller matrix *N* which has the features of the original data. The second one is using the matrix *N* to calculate the anomaly score for every data point of the coming data. The third one is labeling the data point with a high score as an anomaly and use them to update the matrix *N* . Matrix Sketch will be introduced in detail in Section [IV.](#page7)

* 1. **SYSTEM DETAILS**

1. SIMULATOR

*a: KEY COMPONENTS*

To support the simulation, the simulator has some key compo-nents, and the design of the simulator is shown in [Fig.3.](#page5) Four key components will be introduced in detail in the following.

**‘‘parser’’**: Once the user uploads the architecture def-inition le to the simulator, ‘‘parser’’ will parse the architecture and send the architecture con guration to ‘‘asgard’’. In the de nition le, the user needs to illus-trate the name of the service, the number of instances per service, the type of service and the architecture topology.

**‘‘asgard’’**: According to the received architecture con-guration, ‘‘asgard’’ will spawn every needed service instance and help all the instances to build their depen-dency lists. The simulator of VWR supports well up to 100,000 independent service instances in a few GB of RAM. After spawning all the needed service instances, these instances will begin to interact with each others using the communication protocol and produce the com-munication data. The communication protocol will be introduced in detail in Section [III-B.1.c.](#page5)

**‘‘collector’’**: During the simulation, all the producedcommunication data are collected and converted into the form of tracing data by ‘‘collector’’. After that, ‘‘col-lector’’ will writes these data to a write-only log le. ‘‘collector’’ also leverages a data forwarder tool named lebeat [34] to fetch the change of this log le and feed the new added data into ElasticSearch which allows users to get the latest tracing data whenever they want.

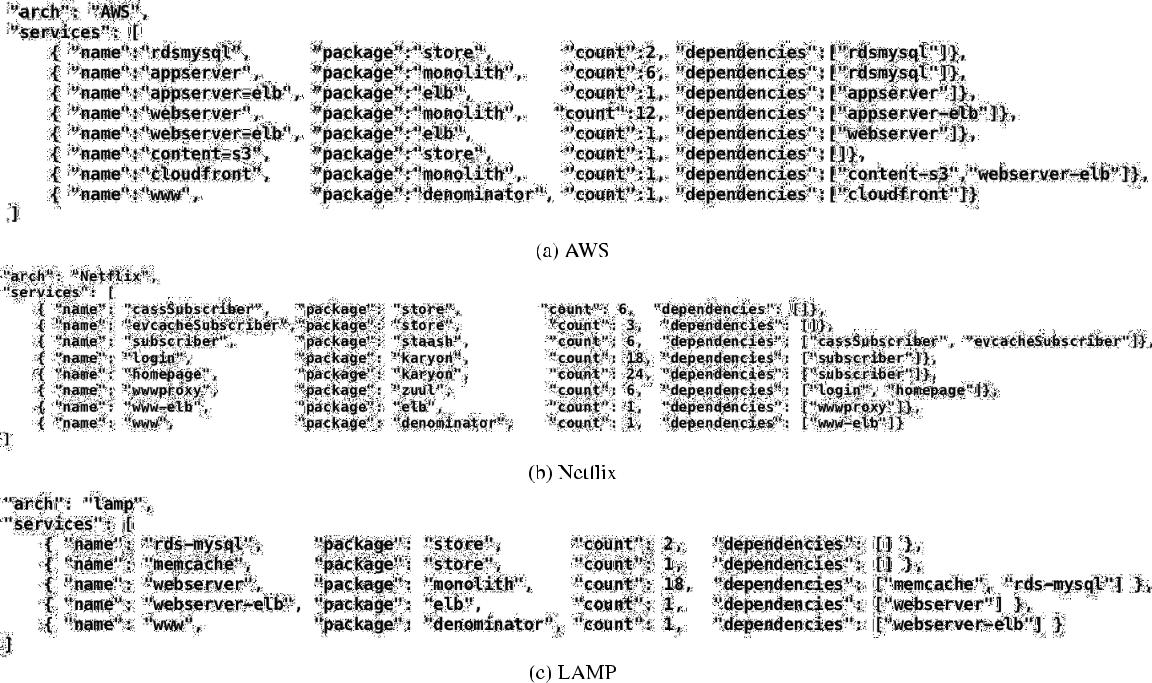
**‘‘chaosmonkey’’**: VWR’s ability to inject fault is a keyfeature. The fault injection component of the simulator is chaosmonkey which waits for the fault requests from the user,extracts the fault information from requests and sends the fault message to the target service according to the parameters in the fault request. There are three types of faults in chaosmonkey which will be introduced in detail in Section [III-B.2.](#page6)

1. *DIFFERENT SIMULATED SERVICES*

As we all know, real microservice architectures are made up of different small services that are responsible for different single and simple functions. To simulate the behavior of the

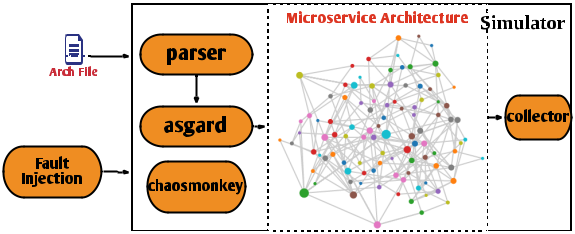
VOLUME 8, 2020

H. Chen *et al.*: Framework of VWR and MS-Based Streaming Anomaly Detection for Microservice Systems



**FIGURE 2.** Examples of arch file to simulate in VWR.

**FIGURE 3.** Framework of simulator.



microservice systems, we need to build the key services in our simulator. The introduction about the key services in a real microservice system and how we simulate them in VWR are illustrated as below.

**Database**: To store the data and provide easy accessto data, microservice systems use the database to man-age their data. Furthermore, the key feature of the microservice system is loosely coupled. Therefore, dif-ferent service components may use different databases to manage their data. There are two types of main-stream databases, SQL and NoSQL (e.g., Cassandra, Redis, etc). As those SQL databases are centralized. Therefore, we implement them as an individual service ‘‘store’’ in our simulator which supports ‘‘Put’’, ‘‘GetRequest’’, and other database operations inside the simulation. For those NoSQL databases, many of them are distributed and should be built as a cluster to manage data. There-fore, we implemented Cassandra NoSQL as one service ‘‘priamCassandra’’ in our simulator which supports dis-tributed data management.

VOLUME 8, 2020

**Load Balance**: In microservice systems, the load bal-ancer allows designers to add more instances of their microservice in a way that is transparent to any service consumer, which gives designers an increased capacity to handle the workload, and also reduces the probability of a single host failing. Therefore, we also simulated it as an ‘‘elb’’ service in our simulator which spreads messages over microservices in those available zones.

**Service registry**: In microservice systems, we needone mechanism for a service instance to register itself. We should also have a way to discover the service once it is registered. There are three main-stream dynamic service registers: *Zookeeper* [35], *Consul* and *Eureka* [36]. In our simulator, we simulate the basic function of *Eureka* which records existing services and also uses around-robin lookup approach to discover new services.

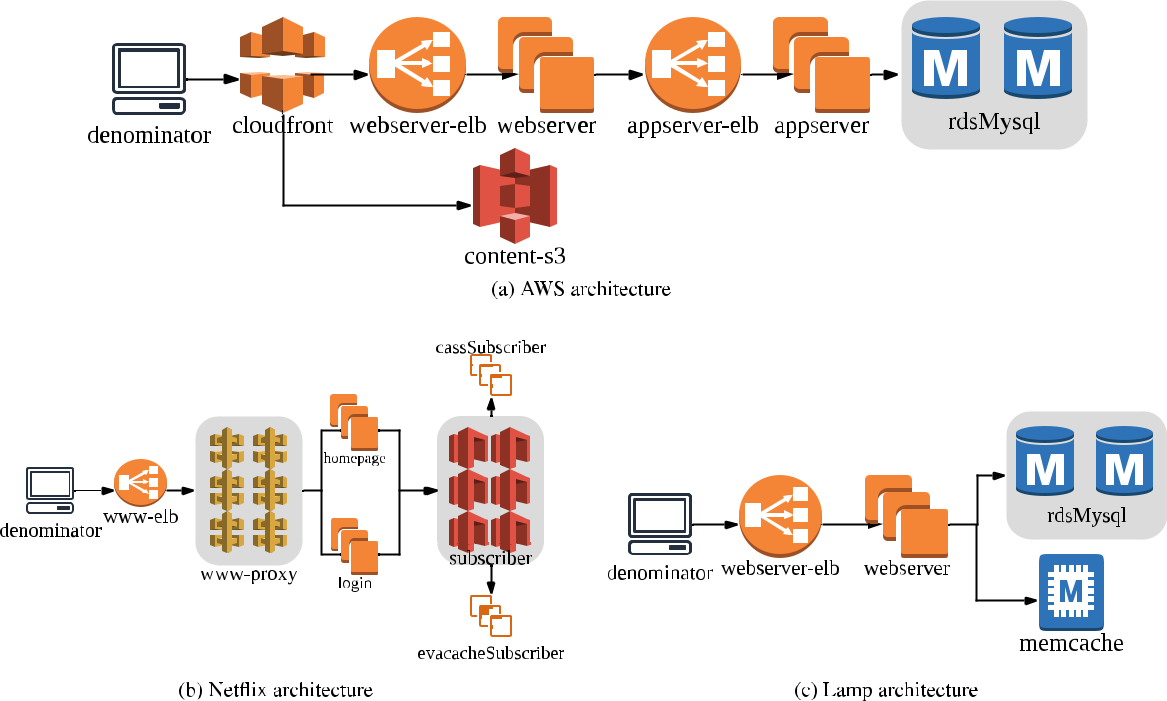
**API gateway**: An API gateway encapsulates the internalarchitecture and provides an API that is tailored to each client. Moreover, it is also responsible for request rout-ing and so on. In our simulator, we simulate it with the *Zuul* [37] service. *Zuul* is an open source API gatewayprovided by Net ix. which routes the requests to one speci c service.

1. *THE COMMUNICATION PROTOCOL*

In practice, the microservice system is always deployed in a distributed way. Services usually communicate with each other using simple protocols such as RPC or HTTP. To sim-ulate these interactions, we construct a simple communica-tion protocol by Golang’s **channel** mechanism. There are

43417

H. Chen *et al.*: Framework of VWR and MS-Based Streaming Anomaly Detection for Microservice Systems



**FIGURE 4.** Different microservice architectures.

some different types of messages which need to be introduced in detail.

**Hello**: At the beginning, ‘‘Hello’’ messages are sent toall service instances to help initialize the con guration of these instances. Using Hello message, the service instance can con rm its name and its address of channel.

**Namedrop**: When all instances are created, ‘‘asgard’’will send ‘‘Namedrop’’ messages to all service instances. As shown before, every service instance man-ages a dependency service list which records succes-sor service instances’ names and channels. Therefore, ‘‘Namedrop’’ messages sent to an instance should con-tain information about their successor service instances’ names and channels. By this way, all instances can build their dependency lists.

**Put**: ‘‘Put’’ message simulates the process of insertingdata into the data table of the database. When ‘‘Put’’ messages arrive at a ‘‘store’’ service instance, it will record the key and the value into the map it manages. If messages arrive at one ‘‘PriamCassandra’’ service instance, the ‘‘Put’’ message will be sent repeatedly to other instances.

**GetRequest**: ‘‘GetRequest’’ message simulates a searchrequest to the database in practice. ‘‘GetRequest’’ from ‘‘denominator’’ service will be forwarded to the database service. When ‘‘GetRequest’’ message arrives at any instance of ‘‘store’’ service, the instance will search for requested values in its map. If the requested value exists, the instance will respond with a ‘‘GetResponse’’ message. Otherwise, that instance will have no response. If the message arrives at one

43418

‘‘PriamCassandra’’ service instance, the ‘‘GetRequest’’ message will be forwarded to other instances of the same service to search for that value.

**GetResponse**: ‘‘GetResponse’’ message simulates theresponse to the ‘‘GetRequest’’ message. The response contains a message generated randomly.

1. CHAOS ENGINEERING

Beyond the simulator of the microservice architecture, we also integrate the fault injector in VWR. Similar to the ‘‘ChaosMonkey’’ provided by Net ix, the name of our fault injector in the simulator is also called chaosmonkey. We implement it as a component of the simulator. During the simulation, the fault injector shown in Fig.1 will receive fault injection requests from users and extract the fault information from requests including the type of fault, the target service and so on. After parsing, the injector will use the ‘‘chaos-monkey’’ to inject faults.

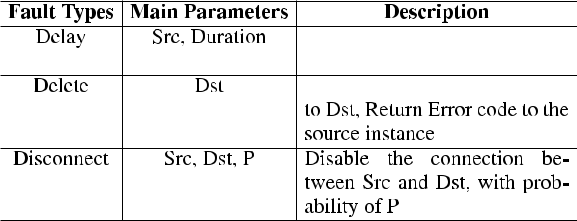
In previous work [38] and [39], researchers collected 22 representative microservice faults and listed the detail description of these faults. For those faults that result in the malfunctioning of system services by raising errors or producing incorrect results, researchers regard them as **func-tional faults**. Those faults that in uence the quality of servicecomponents such as performance and reliability are regarded as **non-functional faults**. In fact, our simulator only sim-ulates the communication between service components in the simulated architecture without any ne-grain function of service components, any detailed malfunctioning of sys-tem services cannot be re ected in our simulator. Therefore,

VOLUME 8, 2020

H. Chen *et al.*: Framework of VWR and MS-Based Streaming Anomaly Detection for Microservice Systems

**TABLE 1.** Three types of faults that can be injected. 4) VISUALIZATION

we provide three non-functional faults in VWR as shown in Table [1.](#page7)



In reality, due to the overload of servers, all requests sent to a speci c service may not be answered in time. Therefore, our rst fault **‘‘Delay’’** behaves like long time responses to requests of one speci c service. Secondly, due to improper or inconsistent con gurations of microservices or their envi-ronments, a service instance would crash. Therefore, another fault **‘‘Delete’’** behaves as one speci c service instance fails. Thirdly, caused by the low network performance, the connection between 2 services would be lost. Therefore, we simulated the **‘‘Disconnect’’** fault which disables the connection between 2 speci c services. Moreover, since there is a large volume of service instances in microservice sys-tems, the fault injections will be conducted randomly to one instance.

Next, the process of how these faults are injected into the simulated architecture will be described in detail. For the **‘‘Delay’’** fault, the user needs to give the name of the target service *A* and the delay time in the request which will be sent to VWR. After that, ‘‘chaosmonkey’’ will send the ‘‘Delay’’ message to one instance of service *A*. When it receives the ‘‘Delay’’ message, it will halt until the end of the injection duration. For the **‘‘Delete’’** fault, the user needs to give the target service *A* in the request. After that, ‘‘chaos-monkey’’ will send the ‘‘Delete’’ message to one instance of *A* which will soon stop forwarding other received messages and send ‘‘GetResponse’’ messages with Error code (e.g., 404) back to those service instances that depend on it which will remove the deleted instance from the dependency list. For the **‘‘Disconnect’’** fault, users need to give two target services*A*,*B*, and a probability . Then, ‘‘chaosmonkey’’ will send the‘‘Delete’’ message to instances of *A* and remove *B* from *A*’s dependency list.

3) EVALUATOR

There are two tasks the evaluator needs to nish. First of all, when users send requests to activate the evaluator, the evalu-ator will access to ElasticSearch [40], and fetch the produced tracing data. After that, the evaluator will convert the tracing data into the format that the anomaly detection algorithm needs and then detect anomalies. Besides, users can upload the anomaly detection or root cause analysis algorithms before evaluation. After the evaluation, users can download the results of these algorithms from VWR.

We integrate Net ix’s Vizceral [41] into VWR to visualize the de ned architecture. The architecture les uploaded by users will be converted to the format that Vizceral needs and then Vizceral will help to show the topology in the dashboard. What is more, VWR uses Grafana [42] to provide different widgets, e.g., time series, tables, text elds for single metrics.

**IV. MATRIX SKETCH BASED ANOMALY DETECTION**

Due to the continuously generated data, traditional of ine algorithms that attempt to store the entire stream for analy-sis is not able to handle a large volume of streaming data. What is more, an important requirement of anomaly detec-tion is to detect anomalies in real-time before the system is seriously impacted by anomalies. Therefore, we construct an unsupervised anomaly detection algorithm based on the method proposed in [24] and integrate it in VWR to provide a baseline for users. The algorithm is based on Matrix Sketch and it can ef ciently and effectively detect anomalies in large online data streams. What is more, this algorithm consumes limited memory and requires just one pass over the data which supports users to detect anomalies as soon as possible. In this section, we will show the detail of this algorithm and illustrate how to use this algorithm to detect anomalies on the tracing data.

Firstly, there are two ways to get the duration of a request. Fig.5 shows the tracing process of a request from a client to a server. In this gure, when a client service instance sends a request to the server service instance (i.e., cs), we will get a new record which is called ‘‘span’’ [43] in the tracing data with a new endpoint whose value is ‘‘cs’’ and the timestamp is *Tcs*. When the server service instance receives this request (i.e., sr), the request is processed at the server end. After that, the server service instance will send the response back to the client service instance (i.e., ss), then the span will get two new endpoints whose value is ‘‘sr’’ and ‘‘ss’’ with times-tamps *Tsr* ; *Tss* respectively. When the client service instance receives the response from the server service instance (i.e., cr), the span will get another endpoint whose value is ‘‘cr’’ with timestamp *Tcr* .



**FIGURE 5.** The tracing process between a client and a server.

*Duration*2 in the equation [2](#page8) means the processing time of the received request. *Duration*1 in the equation [1](#page8) means the duration of the whole request including *Duration*2 and the propagation time of the request and the response between the client and the server. Therefore, we can use *Duration*1 or *Duration*2 of every request between two service instances as a performance indicator. If one of these two indicators of a request is much larger than other requests, we can regard this request as an abnormal request. In our experiment, we choose

|  |  |
| --- | --- |
| VOLUME 8, 2020 | 43419 |

H. Chen *et al.*: Framework of VWR and MS-Based Streaming Anomaly Detection for Microservice Systems

*Duration*1 as our evaluation indicator because it covers differ-ent transmission phases which shows the performance status of a request more holistic.

|  |  |
| --- | --- |
| *Duration*1D *Tcr Tcs* | (1) |

**Algorithm 1** Anomaly Detection Algorithm

**Input**:

new datapoints *Yt* 2 *Rm n*;

r-rank matrix with orthogonal columns *U*(*t* 1)*r* ; prior fault probability ;

|  |  |  |  |
| --- | --- | --- | --- |
| *Duration*2D *Tss Tsr* | (2) | **Output**: |  |
|  |  |

Secondly, when we get the tracing data from VWR, the data will be converted into a matrix in the following manner. Regard every service instance as a bucket and the sent requests as objects in the bucket. Then, we count the number

* of arriving requests at time *t* in every bucket and get the duration of every request by subtracting the timestamp in the endpoint whose value is ‘‘cs’’ and the timestamp in the endpoint whose value is ‘‘cr’’. For every service instance, it will sum up the duration of all requests at time *t* and divide the duration sum by *N* . Thus, we can get the average request completion time at time *t* for every service instance. The produced tracing data arrives in streaming and every service instance will send requests at any time. By doing this, we can form a high-dimension matrix as an input data for the Matrix Sketch algorithm, the number of rows represents the number of service instances, the number of columns represents the number of all coming datapoints at time *t* and the element of the matrix denotes the instance’s average request completion time at time *t*. Let f*Yt* 2 *Rm n*; *t* D 1; 2; 3; : : :g denote a sequence of streaming tracing data, and *Yt* means all datapoints arrive at time t. The goal of the Matrix Sketch algorithm is to detect ‘‘anomalous data-points’’ in *Yt* at every time *t*. We assume that *N*[*t* 1]*n* D [*N*1*n* ; *N*2*n* ; *N*3*n* ; : : : ; *Nt* 1*n* ] represents the set of all datapoints that had been identi ed as non-anomalous by the algorithm

in *Y*[*t* 1] D [*Y*1; *Y*2; *Y*3; : : : ; *Yt* 1]. At time *t*, *Ntn* denotes all non-anomalous datapoints in *Yt* and *U*(*t* 1)*r* is assumed as a

lower rank matrix with orthogonal columns that can linearly reconstruct *Ntn* in *Yt* . At time t D 1, we can rstly gather a small set of non-anomalous data and initialize *N*1*n* from it, because it is easier for us to collect a set of non-anomalous data. Or we can also collect a small set of data and use any unsupervised anomaly detection algorithm to label the data which we can use to construct *N*1*n* .

After illustrating the detail of how to get the input matrix and the goal of our algorithm, we will talk about the framework of the algorithm. The framework of the stream-ing anomaly detection using Matrix Sketch is listed in Algorithm [1.](#page8) This algorithm alternates between an anomaly detection and a singular vector updating step.

The rank *r* matrix which can represent all datapoints in *N*[*t* 1]*n* is *U*(*t* 1)*r* , and we can use this matrix to detect anoma-lous datapoints in *Yt* by calculating the score for every point *yi* 2 *Yt* through the calculation [3.](#page8) Note that in this calculation, Im is Identity matrix which size is m and after the subtraction, we get the *L*2-norm of the result.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *score* D |  | (*Im* | *U*(*t* 1)*r* *U*(*Tt* 1)*r* | )*yi* |  | 2 | (3) |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

updated set of non-anomalous datapoints *N*[*t*]*n* ; updated r-rank orthogonal matrix *Utr* ;

**1** *Nta*;*Ntn* []; [];

* //*Nta* ; *Ntn* denote abnormal and normal data points respectively
* **foreach** *column yi*2*Yt* **do**
* Calculate anomaly score:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **5** | score = (*Im* | *U*(*t* 1)*r* | *U*(*Tt* 1)*r* | )*yi* |  | 2 |
|  |  |  |  |  |  |  |

* **if** *score***then**

|  |  |
| --- | --- |
| **7** | *Ntn*[*Ntn* ; *yi*] |

* **else**
* *Nta*[*Nta* ; *yi*]

1. **end**
2. **end**

**12** *N*[*t*]*n* [*N*[*t* 1]*n* ; *Ntn* ]

1. *Update singular vectors:*

**14** *Utr* *Update*(*U*(*t* 1)*r* )

1. return *N*[*t*]*n* ; *Utr*

After getting the anomalous score for every coming point in *Yt* , we use the threshold to decide which points are anomalous. If the score is greater than , we say the corre-sponding data point is abnormal. In this step, we can also extract a set of non-anomalous points *Ntn* in *Yt* . With *Ntn* , we start to update the matrix *U*(*t* 1)*r* and get the next sketch matrix *Utr* .

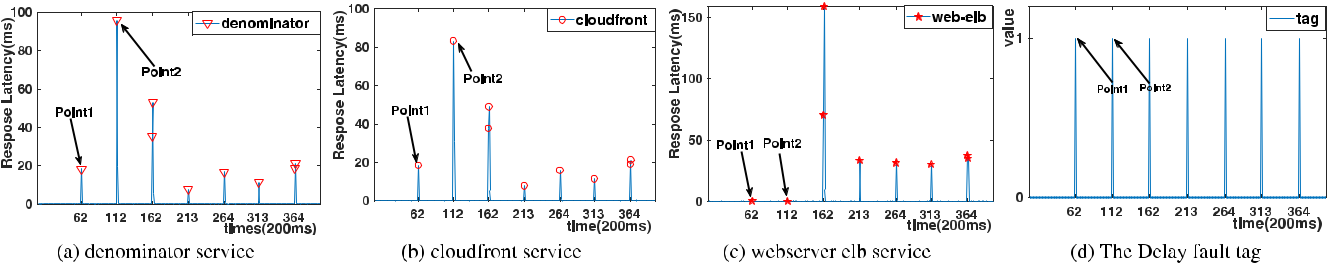
There are three ways to update the matrix *U*(*t* 1)*r* in [24]. The rst algorithm is based on global algorithm, which generates the matrix from the globally collected sample set *N*[*t*]*n* D[*N*[*t* 1]*n* ; *Ntn* ]. As we can see, both the compu-tational and memory requirement of this global algorithm will increase with time. According to [24], the random sketch update algorithm almost gets the same result as the global algorithm more ef ciently. Therefore, we use the random sketch update algorithm to update the matrix *Ut* . The procedure of the random sketch update is listed as algorithm [2.](#page8)

In each iteration, matrix *Mt* will be computed by combin-ing current sketch matrix *Et* 1 with *Ntn* . *E*0 will be initial-ized as an empty matrix. Then, *Y* will be generated using *Mt* and a random Gaussian distributed matrix, and then *Y* will be conducted a QR factorization [44] to get matrix *Q* and *R*. Next, we get the eigen value P2*t* and eigen vec-tor *At* by conducting the eigen function with *Q* and *Mt* . After that, we multiply *Q* by *At* to get *Ut* , from which we choose top-*r* vectors to form matrix *Utr* and choose top-

‘ to update matrix *Et* where *r* and ‘ represent the number of vectors.

43420 VOLUME 8, 2020

H. Chen *et al.*: Framework of VWR and MS-Based Streaming Anomaly Detection for Microservice Systems



**FIGURE 6.** Fig.(a)-Fig.(c) is the example view of services latency and Fig. (d) shows the time we injected Delay faults. To get Fig.(a)-Fig.(c), in every200ms,we collect all traces, sum up the request completion time of all traces and divide it by the count of traces.

**Algorithm 2** Random Update Algorithm

**Input**:

|  |  |  |
| --- | --- | --- |
|  |  | *Nt* 2R*m n*; |
|  |  | randomized matrix sketch updated at *t* 1, |
|  | *Et* | 1 2 *Rm l* ; |
|  |  | *r* ‘; |
|  | **Output**: | |
|  |  | randomized matrix sketch update at *t*,*Et* 2 *Rm l* ; |
|  |  | *Utr* ; |
| **1** | *Mt* | [*Et* 1; *Nt* ] |
| **2** | *k* | 100‘ |

* Generate an m k random Gaussian matrix

**4** *YMt MtT*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **5** | *QR* |  | *QR*(*Y* )(QR factorization for Y) | | | |  |  |
|  | *t*P | | *diag*( *t*1; : : : ; *tk* )) | | |  |  |  |
| **6** | *At* | *t*2 | *AtT* | *EIG*(*QT Mt MtT Q*) (where | | |  |  |
|  | 2 |  |  | 2 | 2 |  |  |  |
|  | P*t* | D | *t* | *t* | *t* | *t* | *t* |  |
| **7** | *U* | *QA* | (*QQT* *M* | *MT QQt* approximates *M* | | *MT* ) |  |
| **8** | *Ut*‘ |  | [*u*1; : : : ; *u*‘](where *Utk* | | | D [*u*1; : : : ; *uk* ] and | |  |

‘ *k*)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | (*trunc*) | |  |  | q) |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | q *t*‘1 | |  |  |  |
| **10** | *E* *t*‘ | *U* | *t*‘ | (*trunc* | *t*1 | *t*‘ | | *t*‘ | |  |
| **9** | P*t* |  | *diag*( |  | 2 | 2;:::;2 | | | | 2; 0) | |  |
| **11** | *Utr* | [*u*1P;:*t*:‘ : *ur* | | | ] (where *Utl* | | | | D [*u*1; : : : *u*‘]) | | | |  |

1. return *Et* and *Utr*

**V. EXPERIMENT EVALUATION**

In this section, we now validate the effectiveness of VWR and the performance of Matrix Sketch by simulating three classical architectures shown in Fig.4. The con guration of each corresponding architecture is shown in Fig.2. We vali-date VWR with respect to two aspects.

We validate that against the actual microservice struc-tures, VWR is able to gain great improvement. Firstly, VWR can rapidly generate large volume of tracing data. Secondly, VWR has the ability to simulate multiple self-de ned architectures. Finally, the cost and the overhead of VWR is lower than real systems.

We validate VWR can simulate the latency curves of real applications. Ensuring that VWR can reproduce the latency increasing pattern is essential in its effectiveness.

Before our experiment, we obtain three architectures AWS, Net ix and LAMP from [45] [47]. In order to apply these

VOLUME 8, 2020

architectures to VWR, we make some simpli cation and describe the architectures in arch les using the format VWR requires. For AWS, ‘‘rdsMysql’’ service and ‘‘content-S3’’ service are Database services which are con gured as ‘‘store’’ package in the arch le, ‘‘app server-elb’’ service and ‘‘web server-elb’’ service are Load Balance services which are con gured as ‘‘elb’’ package in the arch le, ‘‘cloudfront’’ service, ‘‘webserver’’ service and ‘‘appserver’’ service are the services that simulate a monolithic business logic. They are all con gured as ‘‘monolithic’’ package. Another architecture Net ix is applied to provide stream-ing media service. Therefore, ‘‘cassSubscriber’’ service and ‘‘evcacheSubscriber’’ service are Database services which are con gured as store package, ‘‘subscriber’’ service which is con gured as ‘‘staash’’ package mainly accesses the data from all downstream Database services and forwards data to the upstream service, ‘‘login’’ service and ‘‘homepage’’ ser-vice are con gured as ‘‘monolithic’’ package, ‘‘wwwproxy’’ service simulates an API proxy service which is con gured as ‘‘zuul’’ package, ‘‘www-elb’’ service is con gured as ‘‘elb’’ package. The last architecture simulates the simple LAMP stack. Therefore, ‘‘rdsMysql’’ service and ‘‘mem-cache’’ service are Database services which are con gured as ‘‘store’’ package, ‘‘webserver’’ service simulates the Apache server which is con gured as ‘‘monolithic’’ package and the ‘‘webserver-elb’’ service is con gured as ‘‘elb’’ package. LAMP stack’s Linux component and PHP component can not be demonstrated on VWR, since VWR only simulates the behavior of each service and ignores the detailed logic of services. For three architectures, the ‘‘www’’ service con-gured as a ‘‘denominator’’ package drives the architectures via continuous simulated requests.

**A. VALIDATING THE IMPROVEMENT OF VWR**

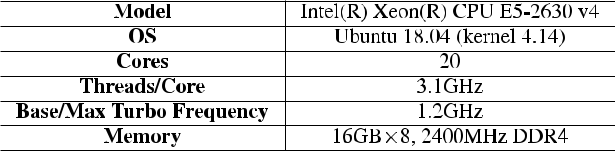
To demonstrate the improvement of VWR, we conduct exper-iments using AWS architecture on our server. Table [2](#page10) shows the specs of our experimental platform. There are three aspects shown as followed.

**Simulation Diversity.** As we have shown before, VWRachieves most of key services’ action which ensures that users can convert many real microservice systems into their digital versions that can run on VWR.

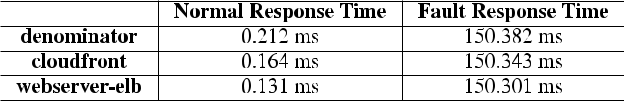
43421

H. Chen *et al.*: Framework of VWR and MS-Based Streaming Anomaly Detection for Microservice Systems

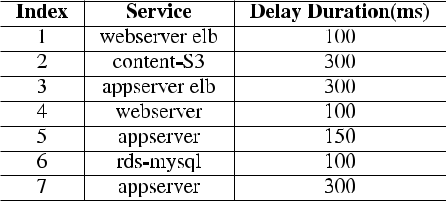
**TABLE 2.** Platform specification.



**TABLE 3.** The request completion time of three services in differentcondition.



**TABLE 4.** The delay fault injection rule.



**Fast Traces Generation.** In order to obtain enoughdata to conduct intelligent analysis, the speed of trace data generation is another key aspect that requires us to notice. Running AWS in VWR generates 124961 traces in a 300s simulation and generates 251289 traces in a 600s simulation. Moreover, the generation speed can be tuned according to users’ requirement.

**Low Simulation Overhead.** Due to the architec-ture simulation, traces generation, and the amount of resources needed to store trace data, VWR also brings additional overhead. After testing the performance of VWR during simulation, we nd the overhead is low. During a 100s simulation, VWR costs only 3% CPU utilization. What is more, the memory usage of VWR increases as the simulation goes and after 100s, VWR uses about 20MB. As VWR is simulating, traces are cached in memory that will be preserved in the disk once the simulation is done.

1. **SIMULATED FAULTS VALIDATION**

We conducted experiments to evaluate the effectiveness of fault injection, by injecting faults into a microservice system spawned by the AWS architecture shown in Fig.4a.

1) DELAY FAULT

According to injection rules listed in Table [4,](#page10) we injected 7 delay faults into different services shown in Fig.4a, then used collected traces to create data graphs in Fig.6 and create Table [3.](#page10) Fig.6a-Fig.6c represent average requests completion time of denominator service, cloudfront service, and web-server elb service respectively. Fig.6d shows timestamps of

43422

injected anomalies. Table [3](#page10) records one request’s response time of denominator service, cloudfront service and web-server elb service in normal condition or in fault condition when one webserver service is delayed. From the trend that re ects on Table [3,](#page10) we nd that the delay of up-streaming ser-vice(i,e, webserver) will affect the latency of down-streaming services(i,e, denominator, cloudfront and webserver elb). Furthermore, from the following aspects, we nd that the fault data created from the simulation are consistent with the data from real microservice systems.

First of all, we notice that, in the microservice architec-ture, the effect of delay fault will be accumulated along the service call chain. For example, if one instance of the app-server elb is injected with a delay fault, the effect will be propagated to the instance of the denominator service. In the AWS architecture, these three services lie in the same service call chain, so we can nd that the curves of these three services are quite similar.

Secondly, if one up-streaming service depends on only one down-streaming service which failed, the fault effect of the failed down-streaming service will be fully re ected on the up-streaming service. In our simu-lated AWS architecture, the denominator service only depends on the cloudfront service. Therefore, we can nd that their performance curves are the same.

Thirdly, in real microservice systems, if one service instance is delayed because of its low performance, the request issued to this instance will not be processed in time. Therefore, if the up-streaming service depends on a down-streaming delayed service, the delay effect will be re ected in the up-streaming service. For exam-ple, at Point 1 shown in Fig.6, the delay fault is injected into webserver elb service while the delay fault effect is re ected in the cloudfront service. Moreover,

Fourthly, in the real microservice system, if the up-streaming service depends on two down-streaming services, the delay fault effect re ected on the up-streaming service may come from one of the down-streaming services. For example, cloudfront service depends on webserver elb service and content-S3 ser-vice. Therefore, at Point 2 shown in Fig.6, the fault effect re ected on cloudfront service comes from the content-S3 service.

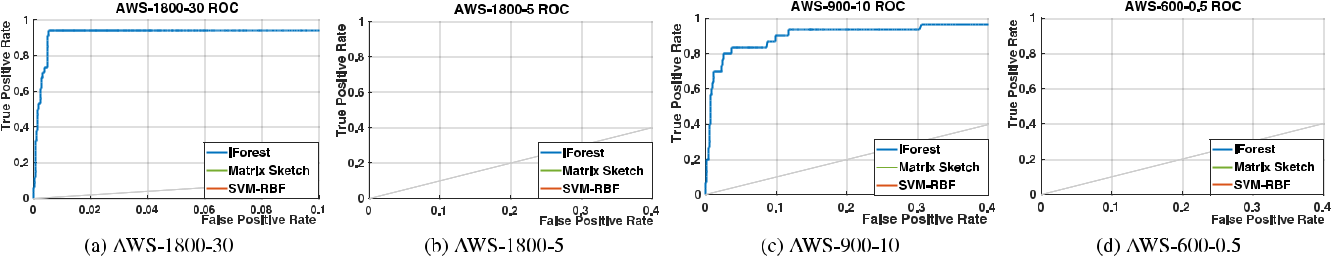
Because of these observations, the produced tracing data can be used in the Matrix Sketch algorithm and we will introduce the result of the algorithm in detail in the next section.

2) DELETE FAULT

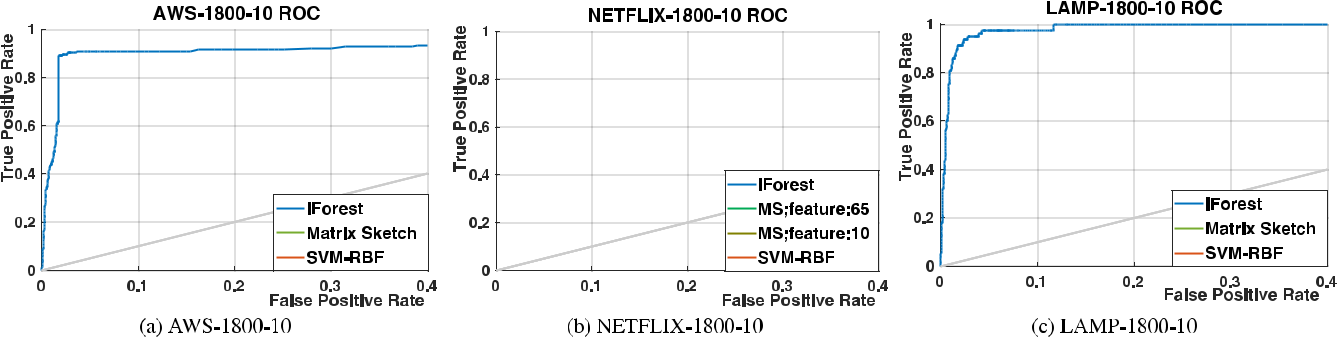
Moreover, we injected a Delete fault to delete the appserver service which has only one instance and aggregated all traces in the simulation. Because there is only one instance of the deleted service along the service call chain, incomplete traces and abnormal traces with error code may be produced. However, when the deleted appserver service has more than one instance, we can only get incomplete traces because the

VOLUME 8, 2020

H. Chen *et al.*: Framework of VWR and MS-Based Streaming Anomaly Detection for Microservice Systems



**FIGURE 7.** ROC curves for matrix sketch, iForest, SVM-RBF on datasets that have different anomaly rates.



**FIGURE 8.** ROC curves for Matrix Sketch, iForest, SVM-RBF on datasets that are produced by simulating three different architectures with uniformanomaly rate.

subsequent requests from the predecessor instances can be sent to other instances of the deleted service.

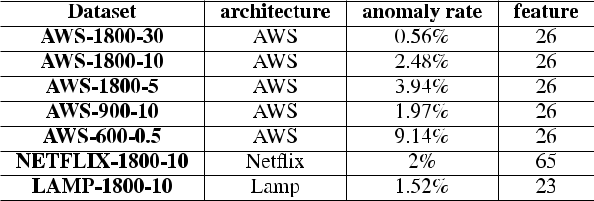
**C. MATRIX SKETCH PERFORMANCE EXPERIMENTS**

In the previous section, we illustrated the effectiveness of the simulated faults. In this section, we will use the produced tracing data to evaluate the performance of Matrix Sketch.

1. EXPERIMENT PREPARATION *a: DATASETS*

We simulated three classical architectures shown in Fig.4 and periodically injected delay faults to different services. From Fig.4 and Table [5,](#page11) the AWS architecture has 8 services and 26 instances; the Net ix architecture has 8 services and 65 instances; the Lamp architecture has 5 services and 23 instances. The feature in Table [5](#page11) means the number of instances of each architecture. After simulation, we get the tracing data and convert them into different datasets shown in Table [5.](#page11) We conducted experiments on datasets to detect the performance of Matrix Sketch. The **AWS-1800-30** dataset is produced in 1800 seconds simulation on theAWS architecture shown in Fig.4a. We injected delay faults per 30 seconds approximately and the anomaly rate of this dataset is 0.56%. The **AWS-1800-10** dataset is also produced in 1800 seconds simulation on the AWS architecture. How-ever, we injected delay faults per 10 seconds approximately and the anomaly rate is 2.48%. The **AWS-1800-5** dataset is obtained in 1800 seconds simulation on the AWS architec-ture. We injected delay faults per 5 seconds approximately

**TABLE 5.** Description of the datasets.



and the anomaly rate is 3.94%. The **AWS-900-10** dataset is obtained from a 900 seconds simulation on the AWS architec-ture. We injected delay faults per 10 seconds approximately and the anomaly rate is 1.97%. The **AWS-600-0.5** dataset comes from a 600 seconds simulation on the AWS archi-tecture and was injected delay faults per 0.5 seconds. The anomaly rate is 9.14%. The **NETFLIX-1800-10** dataset and the **LAMP-1800-10** dataset are produced by 1800 seconds simulation on Net ix architecture(Fig.4b), Lamp architec-ture(Fig.4c). They were injected delay faults per 10 seconds and their anomaly rates are 2%, 1.52%.

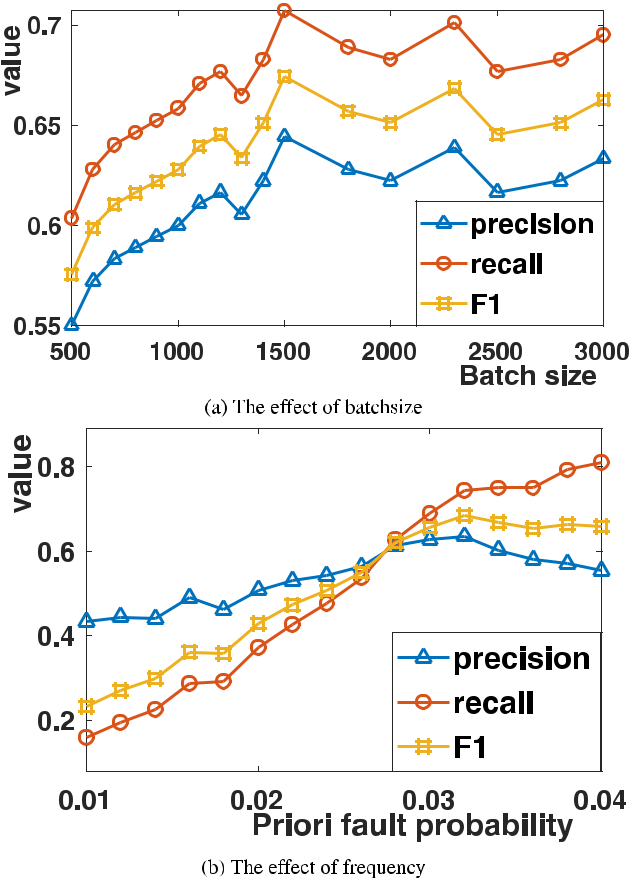
*b: BASELINES*

There are number of approaches for anomaly detection which is discussed in Section [II.](#page3) We compared against two popular and easily implemented algorithms for anomaly detection. These algorithms were chosen considering their scalability on large datasets. **SVM-RBF** is a one-class support vector machine classi er using a radial basis as the kernel. **Isolation** **Forest** is used as another baseline because of its capability in

|  |  |
| --- | --- |
| VOLUME 8, 2020 | 43423 |

H. Chen *et al.*: Framework of VWR and MS-Based Streaming Anomaly Detection for Microservice Systems

**FIGURE 9.** The effect of parameters on matrix sketch.



the high-dimensional analysis. For these algorithms, we tuned their parameters to get the highest Recall without loosing a high F1 score.

*c: PARAMETER SETTINGS*

As illustrated in Section [IV,](#page7) *Matrix* *Sketch* algorithm requires an initial training set to construct *N*1. In our experiment, we assume that there is a small training set of non-anomalous tracing data samples at the beginning. Therefore, we set the size of the initial training set as 3000. Moreover, we select training samples randomly from the set of non-anomalous traces.

We tuned other necessary parameters to achieve bet-ter performance of our Matrix Sketch when we compared the performance between algorithms. Besides, we also dis-cuss the effect of these parameters using **AWS-1800-10** at Section [V-](#page12)C.3,including the batchsize and the prior fault probability given for the detection.

1. COMPARISONS BETWEEN DIFFERENT ALGORITHMS

Fig.7 and Fig.8 show the *ROC* curves of selected algorithms. To increase the effectiveness of the comparison between these three algorithms, we conducted experiments with cross-validation. Each point represents the average True Posi-tive (TP) and Fault positive (FP) of a 20-fold cross-validation

43424

result. The training set was randomly selected from non-anomalous samples and the order of samples was also ran-domly shuf ed each time. In our experiments, we regard the anomaly as the positives while the non-anomalous data as the negatives because the main goal of our task is to detect anomalies that have lower probabilities. Therefore, TP of our *ROC* curve represents those anomaly data that are classi ed correctly, while FP represents the non-anomalous data that are classi ed incorrectly. We make the following observationsV

From Figures 7a, 7b, and 8a, it is evident that Matrix Sketch gets a better performance than other algorithms. However, as shown in Figure 7a and 7d, the other two compared algorithms perform slightly better than Matrix Sketch. Note that the performance of Matrix Sketch is better when the fraction of anomalies approximates to the real anomaly rate in real-world systems.

From Fig.8b, we can nd that Matrix Sketch using fewer features outperforms Matrix Sketch using more features (i.e., service instances). When fewer instances are used, the impact that some inconspicuous services do on the algorithm will be reduced and the algorithm can be easier to understand the impact that the anomalies do to bottleneck services. Due to the same reason, from Figures 8a, 8b, and 8c, it is evident that our approach outperforms other algorithms on AWS and Lamp archi-tecture which has relative small number of service instances, except on the Net ix architecture which has more instances. Therefore, our algorithm is sensitive to the architecture. It tends to work well when the number of service instances is smaller. However, if users want to improve the performance of the algorithm, using fewer features will help it work better.

1. PARAMETER IMPACT ANALYSIS

There are primarily two parameters that we can tune to nd a better performance of Matrix Sketch based anomaly detec-tion. We will discuss the impact of these two parameters, namely the value of the prior fault probability ( ) and the batch size *nt* of every step. ranges from 0.01 to 0.04 with increments of 0.002 while *nt* ranges from 500 to 3000 with increments of 100. *ROC* curves are similar between different values of parameters. Therefore, we calculate the precision, recall and F1 scores at different settings and draw Fig.9a, Fig.9b. Then, we get the following observationsV

From Fig.9a, we can nd that if we increase the value of the prior fault probability , the recall of it will increase which is easier to understand: when *nt* data points come, *nt* of them will be identi ed as anomaly, so ifis higher, Matrix Sketch will identify more data points as anomalies. The precision and F1-score will increase along with the increment of until is slightly larger than the real fault probability which demonstrates that if we use Matrix Sketch to detect anomalies, we should set slightly larger than the real value.

VOLUME 8, 2020

H. Chen *et al.*: Framework of VWR and MS-Based Streaming Anomaly Detection for Microservice Systems

From Fig.9b, it is evident that the trends of precision, recall, and F1 score are similar which demonstrates that the effect of batch size *nt* on them is similar. What is more, we can nd that the value of precision increases slightly along with the batch size until the batch size is 2500. When the batch size is larger than 2500, the value of precision stops increasing. These results indicate a very stable behavior of Matrix Sketch across different batch sizes.

**VI. CONCLUSION AND FUTURE WORK**

This paper designs and implements VWR, a framework with microservice architecture simulation, fault injection, and visualization, which aims to help researchers to test and compare their intelligent operation algorithms in multiple kinds of large-scale microservice systems. With VWR, it is convenient to validate their algorithms for anomaly detection and root cause analysis. We simulate three classical architec-tures and use the produced tracing data to conduct different experiments in order to validate the effectiveness of produced tracing data. We introduce an anomaly detection algorithm based on Matrix Sketch which is an unsupervised algorithm and performs well on the high-dimension streaming data. We compare this algorithm with two popular anomaly detec-tion algorithms namely *IForest* and *1SVM-RBF*. We nd that Matrix Sketch based anomaly detection algorithm per-forms well when the anomaly rate is approximate to the real anomaly rate in real-world systems. Moreover, we nd that this algorithm is sensitive to the architecture and we can improve its performance by using fewer features. In the future work, we will continue to use VWR to conduct experiments for other anomaly detection algorithms and root cause analy-sis algorithms.

**ACKNOWLEDGMENT**

The authors would like to thank anonymous reviewers and who helped to improve this paper.

**REFERENCES**

1. G. Yu, P. Chen, and Z. Zheng, ‘‘Microscaler: Automatic scaling for microservices with an online learning approach,’’ in *Proc. IEEE Int. Conf.* *Web Services (ICWS)*, Jul. 2019, pp. 68 75.
2. P. Chen, Y. Qi, P. Zheng, and D. Hou, ‘‘CauseInfer: Automatic and dis-tributed performance diagnosis with hierarchical causality graph in large distributed systems,’’ in *Proc. IEEE Conf. Comput. Commun. (INFO-COM)*, Apr. 2014, pp. 1887 1895.
3. P. Chen, Y. Qi, X. Li, D. Hou, and M. Lyu, ‘‘Arf-predictor: Effective pre-diction of aging-related failure using entropy,’’ *IEEE Trans. Dependable* *Secure Comput.*, vol. 15, no. 4, pp. 675 693, Jul./Aug. 2018.
4. R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, ‘‘X-trace: A pervasive network tracing framework,’’ in *Proc. 4th USENIX Conf. Netw.* *Syst. Des. Implement. (NSDI)*, Berkeley, CA, USA: USENIX Association,2007, p. 20.
5. *Zipkin Homepage*. Accessed: Oct. 21, 2019. [Online]. Available:https://zipkin.io/
6. S. Roy, A. C. Konig, I. Dvorkin, and M. Kumar, ‘‘PerfAugur: Robust diag-nostics for performance anomalies in cloud services,’’ in *Proc. IEEE 31st* *Int. Conf. Data Eng.*, Apr. 2015, pp. 1167 1178.
7. C. Zhou and R. C. Paffenroth, ‘‘Anomaly detection with robust deep autoencoders,’’ in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery* *Data Mining (KDD)*, 2017, pp. 665 674.

VOLUME 8, 2020

1. C. Sauvanaud, M. Kaaniche, K. Kanoun, K. Lazri, and G. Da S. Silvestre, ‘‘Anomaly detection and diagnosis for cloud services: Practical exper-iments and lessons learned,’’ *J. Syst. Softw.*, vol. 139, pp. 84 106, May 2018.
2. M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and
   1. Brewer, ‘‘Path-based faliure and evolution management,’’ in *Proc. 1st* *Conf. Symp. Netw. Syst. Design Implement. (NSDI)*, vol. 1. San Francisco,CA, USA: USENIX Association, 2004, pp. 1 23.
3. J. Lin, P. Chen, and Z. Zheng, ‘‘Microscope: Pinpoint performance issues with causal graphs in micro-service environments,’’ in *Proc. 16th Int. Conf.* *(ICSOC)*, Hangzhou, China. Berlin, Germany: Springer, 2018, pp. 3 20.
4. P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, and P. Chen, ‘‘CloudRanger: Root cause identi cation for cloud native systems,’’ in *Proc. 18th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.* *(CCGRID)*, May 2018, pp. 492 502.
5. Weaveworks Container Solutions. (2018). *Socks Shop A Microservices* *Demo Application*. Accessed: Dec. 3, 2019. [Online]. Available:https://www.weave.works/blog/sock-shop-microservices-demo-application
6. C. Bennett and A. Tseitlin. (2012). *Chaos Monkey Released Into* *the Wild*. Net ix Tech Blog. Accessed: Dec. 3, 2019. [Online].Available: https://net ixtechblog.com/net ix-chaos-monkey-upgraded-1d679429be5d
7. M. A. Chang, B. Tschaen, T. Benson, and L. Vanbever, ‘‘Chaos monkey: Increasing SDN reliability through systematic network destruction,’’ in *Proc. ACM Conf. Special Interest Group Data Commun. (SIGCOMM)*,2015, pp. 371 372.
8. *Istio*. Accessed: Oct. 21, 2019. [Online]. Available: https://istio.io/
9. D. Meisner, J. Wu, and T. F. Wenisch, ‘‘BigHouse: A simulation infrastruc-ture for data center systems,’’ in *Proc. IEEE Int. Symp. Perform. Anal. Syst.* *Softw.*, Apr. 2012, pp. 35 45.
10. Y. Zhang, Y. Gan, and C. Delimitrou, ‘‘ qSim: Enabling accurate and scalable simulation for interactive microservices,’’ in *Proc. IEEE Int. Symp.* *Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2019, pp. 212 222.
11. P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, ‘‘Using magpie for request extraction and workload modelling,’’ in *Proc. 6th Conf. Symp.* *Operating Syst. Des. Implement. (OSDI)*, vol. 6. Berkeley, CA, USA:USENIX Association, 2004, p. 18.
12. M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, ‘‘Pinpoint: Problem determination in large, dynamic Internet services,’’ in *Proc. Int.* *Conf. Depend. Syst. Netw. (DSN)*, Jun. 2002, pp. 595 604.
13. M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, ‘‘The mys-tery machine: End-to-end performance analysis of large-scale Internet services,’’ in *Proc. 11th USENIX Conf. Operating Syst. Des. Implement.* *(OSDI)*. Berkeley, CA, USA: USENIX Association, 2014, pp. 217 231.
14. F. Doray and M. Dagenais, ‘‘Diagnosing performance variations by com-paring multi-level execution traces,’’ *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 2, pp. 462 474, Feb. 2017.
15. R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman,
    1. Stroucken, W. Wang, L. Xu, and G. R. Ganger, ‘‘Diagnosing perfor-mance changes by comparing request ows,’’ in *Proc. 8th USENIX Conf.* *Networked Syst. Des. Implement. (NSDI)*. Berkeley, CA, USA: USENIXAssociation, 2011, pp. 43 56.
16. S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, ‘‘Detailed diagnosis in enterprise networks,’’ *ACM SIGCOMM Comput.* *Commun. Rev.*, vol. 39, no. 4, pp. 243 254, Aug. 2009.
17. H. Huang and S. P. Kasiviswanathan, ‘‘Streaming anomaly detection using randomized matrix sketching,’’ *Proc. VLDB Endowment*, vol. 9, no. 3, pp. 192 203, Nov. 2015.
18. F. T. Liu, K. M. Ting, and Z.-H. Zhou, ‘‘Isolation forest,’’ in *Proc. 8th IEEE* *Int. Conf. Data Mining*, Pisa, Italy, Dec. 2008, pp. 413 422.
19. C. C. Aggarwal, *Outlier Analysis*, 2nd ed. New York, NY, USA: Springer, 2016.
20. V. Chandola, A. Banerjee, and V. Kumar, ‘‘Anomaly detection: A survey,’’ *ACM Comput. Surv.*, vol. 41, no. 3, p. 15, 2009.
21. M. Ferdman, B. Falsa , A. Adileh, O. Kocberber, S. Volos, M. Alisafaee,
    1. Jevdjic, C. Kaynak, A. D. Popescu, and A. Ailamaki, ‘‘Clearing the clouds: A study of emerging scale-out workloads on modern hard-ware,’’ *ACM SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 37 48, Apr. 2012.
22. H. Kasture and D. Sanchez, ‘‘Tailbench: A benchmark suite and evaluation methodology for latency-critical applications,’’ in *Proc. IEEE Int. Symp.* *Workload Characterization (IISWC)*, Sep. 2016, pp. 1 10.

43425

H. Chen *et al.*: Framework of VWR and MS-Based Streaming Anomaly Detection for Microservice Systems

[30] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, and L. Tang, ‘‘Sirius: An open end-to-end voice and vision personal assistant and

its implications for future warehouse scale computers,’’ in *Proc. 20th* *Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2015,

pp. 223 238.

1. Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, and B. Jackson, ‘‘An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems,’’ in *Proc. 24th Int. Conf. Architectural Support Program. Lang.* *Operating Syst.*, 2019, pp. 3 18.
2. Y. Gan, M. Pancholi, S. Hu, D. Cheng, Y. He, and C. Delimitrou, ‘‘Seer: Leveraging big data to navigate the increasing complexity of cloud debug-ging,’’ in *Proc. ACM Int. Conf. Architectural Support for Program. Lang.* *Operating Syst. (ASPLOS)*, 2019, pp. 1 15.
3. Elasticsearch. (2018). *Filebeat*. Accessed: Dec. 3, 2019. [Online]. Avail-

able: https://www.elastic.co/beats/ lebeat

1. B. Elasticsearch, ‘‘Filebeat,’’ Software, Version 6, Tech. Rep., 2018, no. 1.
2. P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, ‘‘Zookeeper: Wait-free coordination for Internet-scale systems,’’ in *Proc. USENIX Annu. Tech.* *Conf.*, vol. 8, no. 9, 2010, pp. 1 14.
3. K. Ranganathan. (2012). *Net ix Shares Cloud Load Balancing and* *Fail Over Tool: Eureka!* Net ix Tech Blog. Accessed: Nov. 10, 2019.[Online]. Available: https://net ixtechblog.com/net ix-shares-cloud-load-balancing-and-failover-tool-eureka-c10647ef95e5
4. M. Cohen and M. Hawthorne. (2013). *Announcing Zuul: Edge Ser-*

*vice in the Cloud*. Net ix Tech Blog. Accessed: Nov. 10, 2019.

[Online]. Available: https://net ixtechblog.com/announcing-zuul-edge-service-in-the-cloud-ab3af5be08ee

1. X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, ‘‘Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study,’’ *IEEE Trans. Softw. Eng.*, to be published.
2. X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, ‘‘Latent error prediction and fault localization for microservice appli-cations by learning from system trace logs,’’ in *Proc. 27th ACM Joint* *Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*,2019, pp. 683 694.
3. C. Gormley and Z. Tong, *Elasticsearch: The De nitive Guide: A Dis-tributed Real-Time Search and Analytics Engine*. Newton, MA, USA:O’Reilly, 2015.
4. J. Reynolds and C. Rosenthal. (2016). *Vizceral Open Source*. Net ix Tech Blog. Accessed: Nov. 10, 2019. [Online]. Available: https://net ixtechblog.com/vizceral-open-source-acc0c32113fe
5. E. Betke and J. Kunkel, ‘‘Real-time I/O-monitoring of HPC applications with SIOX, elasticsearch, Grafana and FUSE,’’ in *Proc. Int. Conf. High* *Perform. Comput.* Jaipur, India: Springer, 2017, pp. 174 186.
6. *Zipkin Span*. Accessed: Nov. 10, 2019. [Online]. Available:https://zipkin.io/zipkin/2.11.3/zipkin/zipkin2/Span.html
7. W. Gander, ‘‘Algorithms for the QR decomposition,’’ *Res. Rep*, vol. 80, no. 2, pp. 1251 1268, 1980.
8. A. Cockroft, ‘‘Microservices: Why, what, and how to get there,’’ in

*Microservices Workshop All Topics Deck*. New York, NY, USA: ACM,2016.

43426

1. A. Cockcroft, ‘‘The evolution of microservices,’’ in *Proc. ACM Learn.* *Webinar*, 2016, pp. 1 61.
2. G. Lawton, ‘‘LAMP lights enterprise development efforts,’’ *Computer*, vol. 38, no. 9, pp. 18 20, 2005.

HONGYANG CHEN is currently pursuing the bachelor’s degree with the School of Data and Computer Science, Sun Yat-sen University, where he also prepares to pursue the master’s degree with the School of Data and Computer Science. His current research interests include distributed systems, microservices, and cloud computing.



PENGFEI CHEN received the degree from the Department of Computer Science, Xi’an Jiaotong University, in 2016. He is currently a Ph.D. advi-sor in Sun Yet-sen University. He was selected as a member of Rising Star Program of Microsoft Research Asia, from July 2012 to November 2012. After graduation, he worked as a Research Scien-tist with IBM Research, China, from June 2016 to January 2018. He was with the IBM T. J. Watson, from February 2017 to April 2017. He is currently



an Associate Professor with the School of Data and Computer Science, Sun Yat-sen University. He is currently interested in distributed systems, AIOps, cloud computing, microservices, and blockchain. Especially, he has strong skills in cloud computing. Until now, he has published more than 30 articles in some international conferences including the IEEE INFOCOM, WWW, the IEEE ICWS, the IEEE ICSOC, the IEEE Big Data, and journals, including the IEEE TDSC, the IEEE TR, the IEEE TSC, the IEEE TETC, and the IEEE TCC. He serves as a Program Committee Member of multiple conferences and a Reviewer of some internal journals such as the IEEE TRANSACTIONS ON CYBERNETICS, *Information Sciences*, and *Neurocomputing*.

GUANGBA YU received the B.E. degree from the Guangzhou University of Chinese Medicine, China, in 2018. He is currently pursuing the mas-ter’s degree with the School of Data and Com-puter Science, Sun Yat-sen University, China. His current research areas include distributed systems, cloud computing, and AI driven operations.



VOLUME 8, 2020