AUTOMATIC PERFORMANCE DIAGNOSIS AND RECOVERY IN CLOUD MICROSERVICES

vorgelegt von
M.Sc.
Li Wu

an der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -
genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Markus Brill
Gutachter: Prof. Dr. Odej Kao
Gutachter: Prof. Dr. Johan Tordsson
Gutachter: Prof. Dr. Guillaume Pierre

Tag der wissenschaftlichen Aussprache: 18. November 2021

Berlin 2022

Li Wu
Automatic Performance Diagnosis and Recovery in Cloud Microservices

To my family.

ACKNOWLEDGEMENTS

I am deeply grateful to my advisor, Odej Kao, for your valuable advice and guidance in recent years. Thank you for the helpful feedback on drafts of this thesis and your continuous encouragement during the hard pandemic time. The completion of this thesis would not have been possible without the support of my co-advisors: Johan Tordsson and Erik Elmroth. I would like to express my most profound appreciation to both of you for the countless discussions, your insightful suggestions, and professional guidance. I also want to thank Johan Tordsson and Guillaume Pierre for your helpful comments on improving this thesis.

It has been a great honor to be part of the Distributed and Operating System group. I appreciate all my colleagues for creating an enjoyable working environment and enriching my studies directly or indirectly. Special appreciation goes to Jasmin, Sasho , Alex, Florian, and Lauritz for inspiring me to work harder, aim higher, and for your support towards this thesis. And to my colleagues, Sören, Thorsten, Morgan, Jonathan, and others, thank you all for making my study period a little less lonely and for impacting me both on a personal and professional level. A special thanks to Jana for your generous support in settling me down in the lab, the university, and the German system at large.

My appreciation also goes to the FogGuru project for the financial support, the project partners (Elastisys and Las Naves) for sharing their expertise, and all the colleagues in this project. (**This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No.765452**). Big thanks to Mulugeta, Hamid, and Felipe for the exciting discussions and for making moving around Europe a bit less challenging; as well as Guillaume, Mozhdeh, Dava, and others for making this adventure in Europe the most unforgettable experience in my life.

My deep and sincere gratitude to my family, relatives and friends. This journey would have been uneasy without your unconditional love and support. I am forever indebted to my parents for encouraging me to explore and seek my own destiny. Thank you to my grandma, my brother Bob, my cousins for providing me the encouragement. Finally, I want to thank my teachers, Prof. Qiao Wang, Prof. Lijun Chen, Prof. Changping Zhu, and my former colleagues in IBM, for the inspiration and support for my PhD study.

ABSTRACT

Microservices have emerged as a popular pattern for developing large-scale applications in cloud environments for its benefits of flexibility, scalability, and agility. A microservices-based cloud system (named as *cloud microservices*) comprises hundreds or thousands of disparate services that communicate via lightweight messaging protocols, share a finite set of hardware and software resources, and are frequently updated to meet customer requirements. In such a complex and dynamic environment, the occurrence of performance problems (e.g., slow application responses) has become the norm rather than the exception, resulting in decreased revenue, damaged reputation, and significant human effort spent on performance diagnosis and recovery. Moreover, manual operation and maintenance of cloud microservices tend to be error-prone or even impracticable. Therefore, there is an urgent need for an automatic performance problem management system that can not only detect anomalous behaviors (performance anomalies) but also uncover the root causes and recommend recovery actions.

In this thesis, we investigate methods for automatic performance diagnosis and recovery in cloud microservices. The core objectives of this thesis are to identify *where* and *why* a performance anomaly occurs, and further to decide *how* to mitigate it. To this end, this thesis contributes: (1) a method for locating the faulty service from which a performance anomaly originates, including a graphical model for capturing the propagation of the anomaly in the system; (2) two methods for identifying the anomalous metrics that cause a performance anomaly, using deep learning and Spatio-temporal causal inference (CI). In addition, we evaluate the performance of CI techniques on performance diagnosis in cloud microservices through extensive experiments; (3) a method for selecting the most appropriate recovery action to mitigate an identified performance anomaly. Overall, the methods presented in this thesis diagnose root causes and recommend remedies in real-time without requiring any application instrumentation and historical failure instances.

Our methods were implemented in prototypes and experimentally evaluated on representative microservices benchmarks. The evaluations show that they can support automatic performance diagnosis and recovery in cloud microservices and improve service reliability and end-user experience. The results have been peer-reviewed and published at renowned international conferences.

ZUSAMMENFASSUNG

Microservices haben sich aufgrund ihrer Flexibilität, Skalierbarkeit und Agilität zu einem beliebten muster für die Entwicklung umfangreicher Anwendungen in Cloud-Umgebungen entwickelt. Ein auf Microservices basierendes Cloud-System (auch als *Cloud-Microservices bezeichnet*) besteht aus Hunderten oder Tausenden von unterschiedlichen Diensten, die über leichtgewichtige Nachrichtenprotokolle miteinander kommunizieren, häufig aktualisiert werden, und sich eine begrenzte Anzahl von Hardware- und Softwareressourcen teilen. In einer derart komplexen und dynamischen Umgebung ist das Auftreten von Leistungsproblemen (z. B. langsame Anwendungsreaktionen) eher die Regel als die Ausnahme. Darüber hinaus sind der manuelle Betrieb und die Wartung von Cloud-Microservices oft fehleranfällig. Daher besteht ein dringender Bedarf an einem automatischen System zur Verwaltung von Leistungsproblemen, das abnormales Verhalten erkennen, die Ursachen aufdecken und Abhilfemaßnahmen empfehlen kann.

In dieser Arbeit werden Methoden zur automatischen Ursachenlokalisierung und Behebung von Leistungsproblemen in Cloud-Microservices erforscht. Das Hauptziel dieser Arbeit ist es, zu identifizieren, wo und warum ein Leistungsproblem auftritt, und dann zu entscheiden, wie es behoben werden kann. Zu diesem Zweck bietet diese Arbeit (1) eine Methode zur Lokalisierung des fehlerhaften Dienstes, von dem ein Leistungsproblem ausgeht; (2) zwei Methoden zur Identifizierung der anomalen Metriken, die ein Leistungsproblem verursachen, unter Verwendung von Deep Learning bzw. räumlich-zeitlicher kausaler Inferenz (CI); (3) eine Methode zur Auswahl der am besten geeigneten Wiederherstellungsmaßnahmen zur Entschärfung des Leistungsproblems. Insgesamt diagnostizieren die in dieser Arbeit vorgestellten Methoden die Grundursachen und empfehlen Abhilfemaßnahmen in Echtzeit, ohne dass eine Anwendungsinstrumentierung und historische Fehlerfälle erforderlich sind.

Unsere Methoden wurden in Prototypen implementiert und experimentell an repräsentativen Microservices-Benchmarks evaluiert. Die Auswertungen zeigen, dass sie zur Unterstützung der automatischen Leistungsdiagnose und -wiederherstellung in Cloud-Microservices, zur Verbesserung der Servicezuverlässigkeit und der Endbenutzererfahrung eingesetzt werden können. Die Ergebnisse wurden auf renommierten internationalen Konferenzen veröffentlicht.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

MSA Microservices Architecture

MS Microservice

OM Operation and Maintenance

SOA Service Oriented Architectures

QoS Quality of Service

SLA Service Level Agreements

SLO Service Level Objectives

HTTP Hypertext Transfer Protocol

REST Representational state transfer

RPC Remote Procedure Call

MSE Mean Squared Error

CPU Central Processing Unit

KPI Key Performance Indicator

MTTR Mean Time to Repair

CI Causal Inference

GCE Google Cloud Engine

IoT Internet of Things

VM Virtual Machine

OS Operating System

BIRCH Balanced Iterative Reducing and Clustering using Hierarchies

CF Cluster Feature

GC Granger Causality

DAG Directed Acyclic Graph

AD Anomaly Detection

MAP Mean Average Precision

AIC Akaike Information Criterion

SCM Structural Causal Model

LiNGAM Linear Non-Gaussian Model

1 INTRODUCTION

Cloud computing has been a popular topic in the industry and academia for decades. Because of its promises, plenty of companies, from large to small and medium sizes, have migrated to the cloud [1]. Although the cloud provides many benefits such as high availability and scalability, most monolithic architectures that encompass all functionality in a single binary cannot fully exploit, and adapting them to the cloud environment is a non-trivial task [2].

Microservices have emerged as a novel architecture style that overcome the shortcomings of monoliths and can fully benefit from the cloud environment due to their inherent attributes, such as scalability. Since 2014, microservices have gained increasing popularity in the cloud [3] and other domains, such as the internet of things (IoT) [4], web, and mobile [5]. According to the survey conducted by Nginx, nearly 70% of organizations are either using or investigating microservices, including enterprises Netflix, Uber, Amazon, and eBay [6]. In addition, O'Reilly 2020 reports that 92% of organizations are experiencing success with microservices [7].

With the microservices architecture (MSA), an application is decomposed into self-contained and independently deployable services with communicating via lightweight messaging protocols [8–10]. This loosely coupled design of MSA enables independent development, testing, and deployment of services, allowing developers to select the most appropriate programming stacks for each service and to freely apply updates to adapt to technology changes and dynamic customer requirements [8]. Moreover, the decomposition of MSA improves the resilience of services since the problem diagnosis and resolution can be applied to specific components, as opposed to monoliths, where these processes often affect the entire application. Lastly, the modularity of microservices fits well to the model of container-based cloud environments, where each service is accommodated in a single container and orchestrated by a platform [11]. This container-based cloud system, whose applications follow the MSA design, is referred to as *cloud microservices* throughout this thesis (the detailed description of cloud microservices is presented in Chapter 2.1).

1.1 RESEARCH MOTIVATION

In a cloud microservices environment, there are hundreds or thousands of heterogeneous services distributed across separate and often geographically dispersed physical (virtual) machines, with a large number of messages exchanged between them [12]. These services are deployed in containers that can be migrated from one node to another within or across data centers, and their functionalities are frequently updated to meet dynamic customer requirements. Such a complex and dynamic environment indicates that performance problems (e.g., slow application responses) have become the norm rather than the exception. However, the performance is critical to the guaranteed Quality of Service (QoS) and Service-level Agreement (SLA), which are directly related to the user experience and enterprise revenue. For instance, Google reported that an extra 0.5 seconds in search page generation time results in a 20% drop in traffic; Amazon reported that every 100ms of latency costs 1% of revenue per year [13]. Even worse, a prolonged performance degradation tends to affect other services or even cause a severe outage with a considerable revenue loss (e.g., the e-commerce company eBay reported $1.3 billion loss from one-hour outage [14]).

Due to the financial and operational implications of performance deterioration, handling performance incidents thus had become a significant challenge in the daily operation of cloud microservices. Operators are required to quickly detect abnormal performance behaviors (performance anomalies), discover the symptoms and root causes, and execute corrective strategies to address the problems. Even though performance anomalies can be detected sooner or later (e.g., with a deterministic threshold), diagnosing the root causes and recovering the system is time-consuming and tedious for operators. Overall, the challenges are from two aspects: the property of performance anomalies and the characteristics of cloud microservices. On the one hand, a performance problem often ties with a specific interaction of components rather than a single component; therefore, more anomalous events are reported for one problem. In particular, a large number of components in cloud microservices (e.g., Netflix has over 1000 microservices [15], and Uber has 4000 microservices [16]) makes the interactions more complex, inflating the number of performance anomalies. On the other hand, the polyglot design of microservices, frequent updates of software and hardware, and a wide range of root causes lead the same performance anomaly to manifest differently across separate services or distinct anomalies to manifest in similar ways. As a result, it becomes difficult or even impracticable for operators to identify the root cause of such large volume and complicate anomalies.

Once the root cause(s) have been identified, it is also challenging for operators to determine the corrective actions. Due to the dynamic nature and complexity of cloud microservices, it is unlikely to eliminate false positives in both performance anomaly detection and diagnosis. Such incorrect analysis tends to introduce delusive corrective actions, increasing the risk of executing incorrect actions that may create performance anomalies instead of resolve. Additionally, the resource-sharing cloud and frequent-updated microservices yield tremendous uncertainties to the system. All in all, it is tough for operators to foresee the consequences of the potential recovery action and make the appropriate decision to recover the system.

To this end, to reduce monetary loss and relieve operators from urgent calls and tedious work, there is an imperative need to automate performance diagnosis and recovery, or at least some valuable insights to steer troubleshooting and remediation.

1.2 RESEARCH OBJECTIVES

The main goal of this thesis is:

To investigate how analytic methods can be applied to observational monitoring metrics to enhance automatic performance diagnosis and recovery in cloud microservices.

The overall objective is to develop methods that leverage monitoring metrics to resolve service-level performance degradations in cloud microservices, focusing on localizing root causes of performance anomalies and recommending corrective actions. To scope this research, we formulate three research objectives embodied in the main goal and addressed in this thesis, where RO1 and RO2 are for performance diagnosis and RO3 is for recovery.

- **RO1**: To design, develop, and evaluate methods for localizing **where** a performance anomaly originates (*Coarse-grained cause localization*).
- **RO2**: To design, develop, and evaluate methods for identifying **why** a performance anomaly occurs (*Fine-grained cause localization*).
- **RO3**: To design, develop, and evaluate methods for deciding **how** to select the best recovery action(s) for an identified performance anomaly (*Recovery action selection*).

1.3 RESEARCH CONTRIBUTIONS

This thesis contributes to knowledge in automatic performance problem management in cloud microservices within the context of the above three research objectives.

Our key contributions are briefly summarized as follows:

- To address **RO1**, we propose a method named **MicroRCA**[1], to locate the faulty service that causes the performance degradation in cloud microservices. MicroRCA locates the culprit service in real-time, without any application instrumentation. It constructs an attributed graph with service and host nodes to model the anomaly propagation in the system, and then identifies the most likely culprit service by correlating service performance symptoms with corresponding resource utilization. Evaluation on a testbed where a microservices benchmark is deployed and a range of performance anomalies are injected shows that MicroRCA can identify the faulty service with high precision and outperforms several state-of-the-art methods (Chapter 5).

- We address **RO2** with two methods that aim to identify the culprit metrics that explain the performance anomaly in cloud microservices. Both methods are application-agnostic and require no historical failure data. One, named **MicroRCA+**, applies a deep learning algorithm to relevant metrics of a faulty service to detect the most likely performance bottlenecks. It is an extension of MicroRCA developed to address RO1 and target culprit metrics that deviate significantly from the normal. The other, named **MicroDiag**, employs Spatio-temporal Causal Inference (CI) analysis to metrics of all components to uncover the reasons for performance deterioration. MicroDiag identifies the most likely bottleneck metrics through modeling the anomaly propagation across metrics with a metric causality graph. This graph is derived through a combination of spatial propagation analysis among components and a mixture of temporal causal inference among metrics of interactive layers. Both MicroRCA+ and MicroDiag have been experimentally evaluated on a well-known microservices benchmark. Additionally, we perform a comprehensive evaluation of CI techniques on locating root causes of performance anomalies in cloud microservices to fill the gap in understanding the overall performance of this class of methods, as few CI methods have been used in the literature. The results of this experimental study also form the basis for the MicroDiag method. (Chapter 6).

1 MicroRCA - **Micro**services **R**oot **C**ause **A**nalysis

- We address **RQ3** with a method named **MicroRAS**[2] that automatically selects the best recovery action to mitigate the performance anomaly in the absence of historical failure data. MicroRAS is a model-based method for real-time mitigation in cloud microservices. It predicts the effectiveness associated with each candidate recovery action, including the benefit (positive effect) and risk (negative effect) on service performance, using a graphic model and fuzzy logic, and then selects the most appropriate recovery action with a tradeoff between action effectiveness and recovery time, minimizing the duration of degraded performance or outage. Experimental evaluation shows that the actions selected by MicroRAS can effectively recover the faulty services, while reducing the inference to other services and the recovery time compared to several baseline methods (Chapter 7).

The proposed methods are based on the assumption that monitoring metrics are available in cloud microservices, commonly used in the industry by deploying non-intrusive monitoring tools inside the environment. In addition, our methods are designed to address service-level performance degradation problems (e.g., slow responses of services), the causes of which can manifest directly or indirectly in the observational metrics.

In addition to this thesis, this research has contributed to the following peer-reviewed publications:

1. Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. "MicroRCA: Root cause localization of performance issues in microservices." In: NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium. IEEE. 2020, pp. 1–9.

2. Li Wu, Jasmin Bogatinovski, Sasho Nedelkoski, Johan Tordsson, and Odej Kao. "Performance diagnosis in cloud microservices using deep learning." In: International Conference on Service-Oriented Computing. Springer. 2020, pp. 85–96.

3. Li Wu, Johan Tordsson, Jasmin Bogatinovski, Erik Elmroth, and Odej Kao. "MicroDiag: Fine-grained Performance Diagnosis for Microservice Systems." In: Proceedings of the 43rd ACM/IEEE International Conference on Software Engineering Workshops. ICSEW. 2021.

4. Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. "Causal Inference Techniques for Microservice Performance Diagnosis: Evaluation and Guiding Recommendations," In: 2021 2nd IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), 2021.

2 MicroRAS - **Micro**services **R**ecovery **A**ction **S**election

5. Li Wu, Johan Tordsson, Alexander Acker, and Odej Kao. "MicroRAS: Automatic Recovery in the Absence of Historical Failure Data for Microservice Systems." In: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC). 2020, pp. 227–236.

1.4 OUTLINE OF THE THESIS

The remainder of this thesis is structured as follows:

Chapter 2 presents the background for understanding our research topic and methods, including cloud microservices, performance problem management, and analytic methods used in this thesis.

Chapter 3 presents the related work for performance diagnosis and automatic recovery in the literature.

Chapter 4 describes the main problems and challenges of performance diagnosis and recovery in cloud microservices. We also present a reference architecture for performance problem management in cloud microservices, on which we position and briefly describe our proposed methods.

Chapter 5 presents the MicroRCA method for locating the faulty service from which the performance anomaly originates. We also present the testbed setup, evaluation results, and comparisons with several state-of-the-art methods.

Chapter 6 presents two methods, MicroRCA+ and MicroDiag, and their evaluations for locating culprit metrics that explain the performance anomaly. We also present an experimental study of CI techniques for microservices performance diagnosis.

Chapter 7 presents the MicroRAS method for deciding the most appropriate recovery action, after giving a concrete example to illustrate the motivation. A detailed evaluation of mitigating performance degraded services is presented at the end of this chapter.

Chapter 8 concludes this thesis by summarizing our results and providing directions for future work.

2

BACKGROUND

Contents

This chapter presents the fundamentals of our research topic and method, including cloud microservices, performance problem management, and the analytic methods used in this thesis.

2.1 CLOUD MICROSERVICES

A *cloud microservices* environment represents a cloud computing system with applications that follow the design of MSA, which is built on a set of concepts, including cloud computing, microservices architecture, containerization, and service meshes.

2.1.1 *Cloud computing*

Cloud computing is a paradigm that delivers on-demand computing services (e.g., computational resources, development tools, and applications) to users, typically over the internet and on a pay-as-you-go basis [17]. The scalable and elastic resources offerings provide cloud users with a time- and cost-efficient way to fulfill their services. Generally, requested resources can be provisioned in minutes, even geographically from multiple data centers, and users are charged only for what they actually use.

According to the offered services, cloud computing can be broken down into three models [18], as shown in Figure 1(a), which offer different levels of control, flexibility, and management so that users can select the right package of services for their needs.

- Infrastructure-as-a-Service (IaaS) – provides cloud users with on-demand computing resources such as physical or virtual servers, storage, and

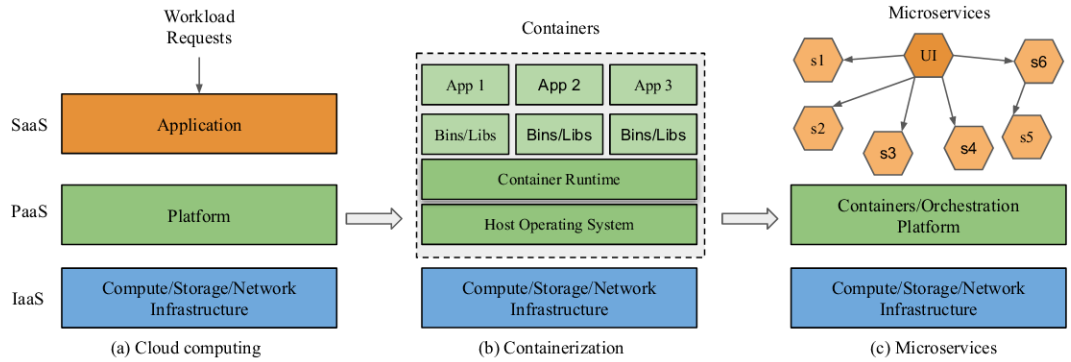


Figure 1: Key elements of cloud microservices.

network. With IaaS, users have low-level control of the computing resources (e.g., scale or shrink resources on their demands) and are responsible for installing the operating system. Examples of IaaS providers include Amazon EC2, Google Compute Engine, Microsoft Azure Virtual Machines, and Alibaba Elastic Compute Service [19].

- Platform-as-a-Service (PaaS) – provides cloud users with an on-demand platform, including hardware, complete software stack, and development tools for developing, deploying, and managing applications. With PaaS, users have control over applications, like configuring the architecture and hosting environment, but not the underlying operating system (OS) and hardware. Google App Engine, Microsoft Azure Functions, Salesforce Heroku, and AWS Elastic Beanstalk are some of the most popular PaaS offerings today [20].

- Software-as-a-Service (SaaS) – also known as cloud applications, provides cloud users with complete applications (along with required software, OS, network, and hardware), which can be accessed directly through a client (e.g., a web browser). With SaaS, users typically have limited control over the application, platform, and underlying infrastructure, except for minimal customization. Examples of SaaS applications are Google G Suite, Microsoft Office 365, Slack, and Dropbox [21].

These service models are also named cloud computing stacks, where different technologies can be applied, e.g., containers for PaaS, MSA for SaaS.

2.1.2 *Containerization*

Nowadays, PaaS is often built on containers, which offer many benefits of high portability, resource and storage efficiency, and better fault tolerance [22,

23]. *Containerization* technology is a virtualization mechanism that allows an application to run in an isolated environment (user space) along with its dependencies. As shown in Figure 1(b), a container holds packaged portions of applications and, if necessary, middleware and business logic in the form of binaries and libraries. Compared to hypervisor-based virtualization, like virtual machines (VM), containers share the host OS, making them much lighter, faster to restart, and easier to scale [24].

Because of its benefits, containerization is considered the ideal virtualization solution for deploying microservices-based applications [25, 26]. Typically, in a cloud microservices environment, services are encapsulated in containers, communicating via network and interacting with volumes of data storage. These containerized services can be easily scaled out to handle more workloads [27] or restarted to mitigate intermittent issues. Moreover, the isolation property of containers implemented with the OS kernel features (i.e., namespaces and control groups (cgroups)) can facilitate inference among services. For example, if a process running in a container exhausts its resources, it may not affect other services if resource limits are set [25].

Since each container covers only a portion of an application and is isolated from others, an orchestration platform is required to manage dependencies among containers for deployments and operations. Many popular orchestration tools, such as Kubernetes [28], Docker Swarm [29], and Apache Mesos [30] are used to manage the lifecycle of containers. Kubernetes is becoming widely adopted among these orchestration tools in computing systems, such as cloud microservices, in virtually all industry sectors (and academia). It enables automated deployment, scaling, and management of containers and improves application resiliency.

2.1.3 *Microservices*

Microservices are novel software architecture styles first proposed by James Lewis and Martin Fowler in 2014 [31]. The core idea of MSA is to decouple complex applications into self-contained and independently deployable units, named microservice (MS), that intercommunicate through lightweight messaging protocols, like Representational state transfer (REST) APIs and Remote Procedure Call (RPC) APIs [32]. Figure 1(c) gives a conceptual overview of microservices where each service, such $s_1 - s_6$ and service UI, implements a single task following the Single-responsibility Principle (SRP). Because of the loosely coupled nature, each service in a microservices-based application can be developed, tested, and deployed independently, thus reducing the complexity of software development and maintenance.

Driven by the advantages of microservices (e.g., flexibility, scalability, and agility), many organizations, including companies like Netflix, Amazon, and Uber, have shifted from monoliths to microservices for designing their large-scale web applications. The monolith is an architecture that wraps all functionalities into one single unit. While the monolith pattern is simple and suitable for small applications, the complexity of the application and the time required to launch new features or enhancements increase dramatically as the number of functions increases. It is also difficult to operate and maintain monolithic applications in distributed systems, such as the cloud. If one component has a performance problem, operators need to look into the entire application to get the root cause. Even worse, any modifications or recovery actions affect the performance of the entire application, usually resulting in significant downtime [33].

In contrast, microservices are self-contained and self-managed and are more flexible in development and maintenance. Performance degradation in microservices tends to affect only a portion of the services rather than causing an outage in monoliths. When a performance degradation is detected, the diagnosis in microservices can focus on a set of anomalous services rather than the entire application, and the bottleneck service can be independently updated, scaled, or restarted to mitigate the problem.

2.1.4 *Service meshes*

Service mesh has become increasingly popular in recent years for its advancements in mitigating the operational complexity of microservices. Although the microservices methodologies greatly improve the development and deployment efficiency, the complex interdependencies between hundreds of services and dynamics inherent in orchestration platform scheduling and frequent service updates pose a significant challenge to understanding and operating the applications at runtime.

A service mesh, serving as a fully manageable service-to-service communication platform, is designed to provide observability of the entire network of distributed microservices (e.g., the topology of services, end-to-end response times) and standardize the runtime operations over traffic flows between services (e.g., load balancing, circuit breaker). In particular, the visibility provided by service meshes can dramatically accelerate the identification of performance anomalies and corresponding bottlenecks; traffic control can speed up the mitigation process by making appropriate and rapid changes to traffic. For example, when a service becomes bottlenecked in an application, the common mitigation strategy is retrying. However, it might exacerbate the is-

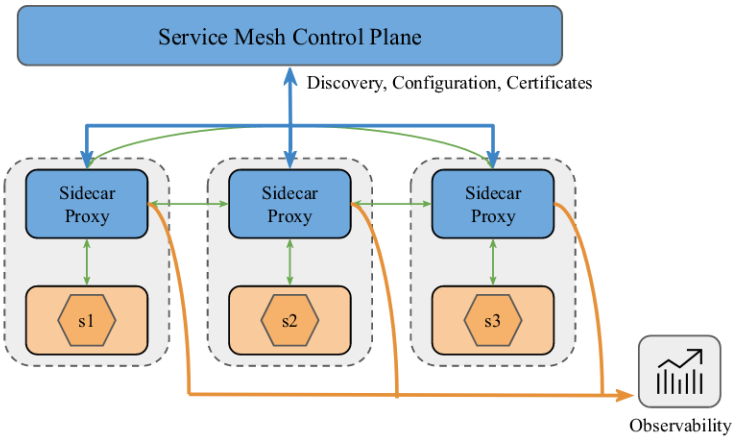


Figure 2: The structure of service mesh.

sue with timeouts. Operators can easily break the circuit to the failed services with service meshes to disable non-functioning replicas and keep the API responsive.

As shown in Figure 2, a service mesh is a dedicated infrastructure over microservices, typically implemented as a scalable set of lightweight network proxies that are deployed alongside application (a pattern sometimes called a sidecar), without the application needing to be aware [34, 35]. These proxies mediate and control all network communication between microservices and act as a point where the service mesh features can be introduced, such as authentication/authorization, load balancing, service discovery, and observability. The proxies comprise the data plane of the service mesh and are controlled as a whole by the control plane, which manages and configures the data planes to enforce policies without being aware of communications in the system.

In our cloud microservices environment, a service mesh is deployed along microservices to collect end-to-end service response times through integrating with a cloud-native monitoring tool named Prometheus[1]. A response time metric clearly shows the duration of a service handling a request and the invocation between two services. For example, a metric of the service mesh Istio[2] named *istio_request_duration_milliseconds* with labels source_workload and destination_workload shows the duration of requests from the service labeled in source_workload to the service labeled in destination_workload. The source and destination information allow to discover and visualize the dependencies between the services.

1 Prometheus - https://prometheus.io/
2 Istio - https://istio.io/

2.2 PERFORMANCE PROBLEM MANAGEMENT

The performance of a system is closely related to its availability, reliability, and scalability and provides crucial information for capacity planning, scheduling, and other decision-making. In addition, performance is a critical indicator of QoS, which reveals the efficiency of a system in accomplishing specific tasks. Typically, it can be measured in terms of time for accomplishing tasks, the rate of processing tasks, or the consumed resources while performing tasks.

Cloud systems rely on effective performance problem management to meet the SLA in runtime operations, particularly when a performance degradation occurs. Overall, performance management aims to: (1) detecting unexpected performance behaviors, referred to as performance anomalies; (2) analyzing monitoring metrics to gather evidence to locate the root cause(s) responsible for the observed performance anomalies; (3) either recommending strategies that may resolve the problem or automatically executing the remedy actions. In this section, we present the details of the above steps after describing the basic concepts, including performance metrics, performance anomalies, root causes, and symptoms.

2.2.1 *Performance metrics and anomalies*

2.2.1.1 *Performance metrics*

In cloud operations, there are typically a set of metrics that are vital to business interests of an organization. These metrics, called key performance indicators (KPIs) or *performance metrics*, are used to measure high-level performance objectives and can be defined at different levels of abstraction (e.g., hardware, virtual machine, application) or from different perspectives (e.g., infrastructure providers, platform providers, and end-users of applications). For cloud microservices, two sets of performance metrics are commonly used:

The first set of performance metrics defines the performance experienced by the end-user of applications. It is relatively small and follows the RED method [36]. The RED method, which includes three key metrics: Rate, Error, and Duration, loosely follows the principle described in Google's four golden signals [37]. These three metrics can be collected from any service and are defined as follows:

- (Request) Rate – the number of requests per second served by services.
- (Request) Error – the number of failed requests per second.

- (Request) Duration – the amount of time each request takes expressed as a time interval.

The second set of performance metrics measures the low-level performance of the underlying system, particularly computational resources used by applications (e.g., CPU, disks, memory, and other physical server functional components). These system-level metrics follow the USE method [38], which includes the following three key metrics:

- Utilization – the average time that the resource was engaged in processing work. It is usually expressed as a percentage over a time interval.
- Saturation – the degree to which the resource has additional work that it cannot serve, often queued. It is usually expressed as a queue length.
- Errors – the count of error events. It is usually expressed as scalar counts.

In cloud microservices, both the USE and RED methods can be implemented with cloud-native monitoring tools. For example, metrics collected by service mesh Istio[3] can implement the RED method and metrics from Node-exporter[4] or Cadvisor[5] for the USE method.

2.2.1.2 *Performance anomalies*

Anomalies also referred to as outliers, deviations, and exceptions can be viewed as a point or group of data points that significantly deviates from the well-defined normal patterns of a dataset [39]. A general widely accepted definition of anomaly is introduced by Hawkings [40]:

"An outlier (anomaly) is an observation which deviates so much from other observations as to arouse suspicions that it was generated by a different mechanism."

Figure 3 illustrates anomalies in a simple two-dimensional dataset. The data have two homogeneous normal regions N_1 and N_2, and most observations lie in these two regions. Points that are sufficiently far away from these two regions, e.g., points o_1, o_2, and points in region o_3, are anomalies. In performance studies, the observations are discrete measurements of a performance metric, which is time series, and the unexpected deviations are called *performance anomalies*.

3 Istio - https://istio.io/
4 Node-exporter - https://github.com/prometheus/node_exporter
5 Cadvisor - https://github.com/google/cadvisor

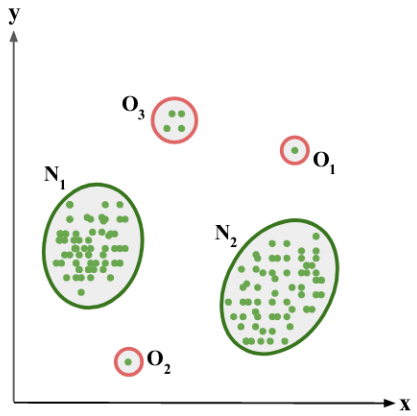


Figure 3: An illustration of anomalies.

In a complex and dynamic system, (performance) anomalies tend to appear in different forms. Three basic types of anomalies are identified by Chandola et al. [39]. (1) Point anomaly – any point(s) that deviates from the range of expected values in a given dataset (e.g., points o_1 and o_2 in Figure 3); (2) Collective anomaly – a homogeneous group of data points that deviate from the normal regions of the rest of data (e.g., points in region o_3 in Figure 3); (3) Contextual anomaly – an anomalous pattern that manifests only under specific execution environments or contexts. More details about different types of anomalies are provided in the research area of anomaly detection [39, 41].

We note that performance problems or issues are interchangeable with performance anomalies throughout this thesis. The performance problems targeted by our performance diagnosis and recovery methods are assumed to have been detected as anomalies. In addition, performance problems are different from failures in a system. A failure is the result of an error caused by a fault in a system, and it usually ends up with an outage which makes the service unresponsive [42]. In contrast, a performance problem puts the service in a degraded state, like taking a long time to respond to the requests. However, a prolonged performance degradation left unattended might induce an outage with failures.

2.2.2 *Root causes and symptoms*

A *root cause* is considered as the source of performance anomalies, which is highly likely to cause unexpected behaviors of other components in the system, and its removal tends to eliminate the unexpected events.

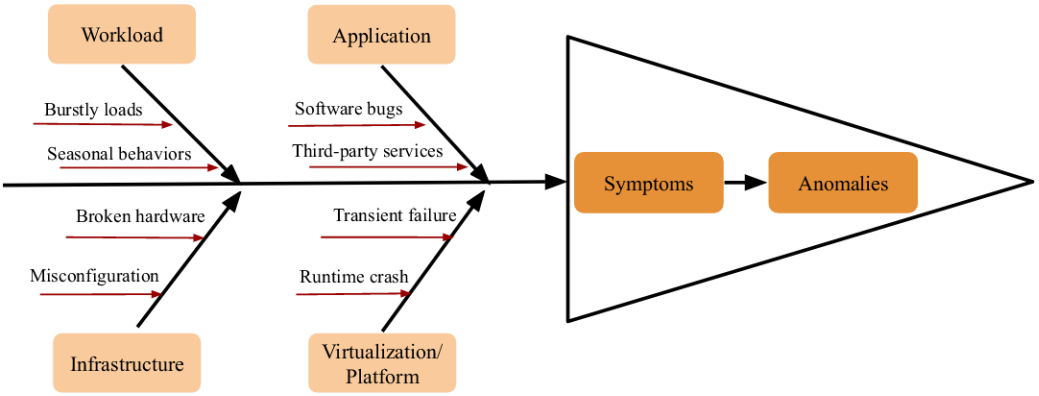


Figure 4: The relationships of root causes, symptoms, and anomalies in performance problem management.

Root causes can be multi-faceted, and some are unlikely to be directly observed in the detected anomalies. Instead, they manifest themselves in the form of symptoms. In performance problem management, *symptoms* are defined as visible external information, such as metrics, logs, traces, and other observational data. Symptoms might include direct observation of the root cause, e.g., an unpredictable surge of requests can be observed in the metric of request rate. Alternatively, they might include visible indicators that are intermediate causes rather than root causes, e.g., the observed high memory utilization of service is intermediate to the unobserved infinite loops in the software.

Figure 4 shows the relationships between root causes, symptoms, and anomalies in performance problem management using a fishbone diagram. The light orange boxes on the left are the main categories of root causes, and the red horizontal arrows are examples of primary causes that further explain each category. The dark orange rectangles on the right are the main effects of the primary causes, including symptoms and anomalies. The thick horizontal arrows depict how primary and intermediate causes can explain the detected anomalies from left to right.

As shown in Figure 4, the root causes in cloud microservices can lie in workload and different abstraction layers as follows:

- Workload: Bursty user requests with continuous peak arrival rates that significantly deviate from the average or expected workload, and seasonal workloads are two examples of root causes from external users.

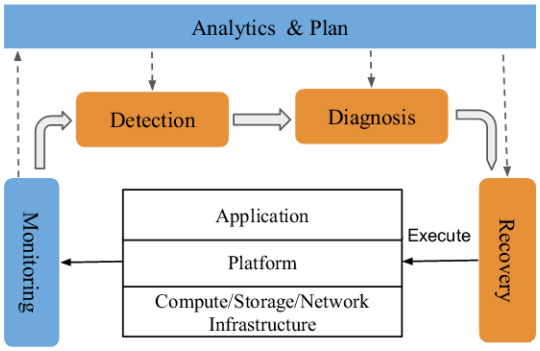


Figure 5: An overview of performance problem management system.

- Infrastructure: Defective hardware (e.g., disks or switches), resource contention from noisy neighbors, unexpected power-off of a machine with an operation error are examples of root causes in the infrastructure layer.
- Containerization and Platform: Transient failures introduced by underlying operating systems (e.g., memory hardware errors), misconfigured network service like Calico in Kubernetes, traffic congestion in API servers, stuck Docker runtime, or misconfigured node attributes that fail the scheduling are the potential causes of containerization and platforms in cloud microservices.
- Application: Application-level issues such as buggy code, software updates, and unstable third-party services or storage are examples that cause service performance degradation. Incorrect application configurations and updates may introduce unexpected resource bottlenecks that can manifest themselves in symptoms.

2.2.3 *Performance problem management system*

Figure 5 provides an overview of the performance problem management system, which includes four modules, namely monitoring, detection, diagnosis, and recovery. This system resembles a self-healing system and loosely follows the MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge) feedback loop, the most influential reference control model for automatic and self-adaptive systems [43].

The problem management process relies heavily on the monitoring module. It collects detailed information, including performance metrics, from different layers of a cloud system and forwards them as input to functions for the analytics and plan. Each function in the analytics and plan module applies one

or more analytics techniques, e.g., statistical analysis, machine learning, time series analysis, etc., to extract insights and structure actions needed to achieve goals and objectives. In performance problem management, the analytics and plan module includes the following three functions:

- (Performance anomaly) Detection: aims to identify or predict anomalies from performance metrics provided by the monitoring module. When an anomaly is detected, it first checks whether its symptoms are similar to a previously seen problem so that a known root cause and recovery action can be recommended. This similarity matching accelerates the diagnosis process and avoids escalation. For new symptoms, this function gathers evidence that partially explains the anomaly and helps narrow the search path for possible root causes.

- (Performance anomaly) Diagnosis: infers the most likely root causes from the observable symptoms along with detected performance anomalies (e.g., identifying abnormal resource consumption in a physical machine). Based on the identified root causes, potential recovery strategies can be determined to mitigate the anomaly in the recovery function. An accurate root cause localization of performance anomalies is essential for mitigating the problem, restoring the system to a stable state, and future recall.

- (Performance anomaly) Recovery: aims to resolve the performance anomalies automatically (e.g., roll back a faulty patch or suspend an offending container) or recommend strategies for operator or a higher-level control system to mitigate the problem. It creates or selects a strategy (ranging from a single command to a complex workflow) to perform an alteration in the managed system. It relies on information and recommendations from the last two phases or policies and historical data stored in a knowledge base. After the recommended actions get executed, the system changes can be used to update the knowledge.

Many Application Performance Management (APM) tools (e.g., Dynatrace [44] and AppDynamics [45]) have been developed to meet the stringent performance and availability requirements in the industry. This type of tool provides a sophisticated view of the monitored system and helps operators quickly identify, isolate, and resolve problems by diagnosing root causes. However, most APMs are more capable of detecting anomalies but less in diagnosis and recovery.

2.3 ANALYTIC METHODS

In this section, we describe the analytic methods investigated or used in this thesis.

2.3.1 *BIRCH*

BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) [46, 47] is an integrated hierarchical clustering algorithm that aims to cluster data points within large datasets. It processes given data points on the fly and stores them in a Clustering Feature Tree (CF-tree), where Clustering Feature (CF) is a 3-tuple that summarizes the information of the cluster. The concepts of CF and CF-tree allow for a fixed size memory usage and logarithmic time for iterative updates, thus improving BIRCH in clustering large datasets on speed and scalability.

A CF is used to represent a cluster by a vector of three values $CF = (N, Ls, Ss)$:

- N is the number of multi-dimensional data points x_i in a cluster,
- Ls is the linear sum of $\sum_i^N x_i$ of the data points,
- Ss is the squared sum $\sum_i^N x_i^2$ of the data points.

This CF summary is not only efficient because it stores many fewer data points for a cluster, but also accurate for calculating the measurements (centroid C and radius R) to make clustering decisions in BIRCH. Furthermore, the centroid and radius of a cluster can be derived from the CF vector as follows:

- the centroid $D = \frac{Ls}{N}$
- the radius $R = \sqrt{\frac{N \cdot (\frac{Ls}{N})^2 + Ss - 2 \cdot (\frac{Ls}{N})^2 \cdot Ls}{N}}$.

A CF-tree is a height-balanced tree with two parameters: branching factor B and threshold T, where B is the maximum number of CF subclusters in each node, and T is the maximum radius value of the cluster. This is a very compact representation of the dataset since each entry in a leaf node is not a single data point but a subcluster (which absorbs many data points with a radius below a certain threshold T).

While data points enter into BIRCH, a CF-tree is built incrementally. Each new point starts at the root and recursively walks down the tree, entering the subcluster of the nearest center. When adding a point at the leaves, a new

cluster is created if the cluster radius R exceeds the threshold T; otherwise, the point is added to the nearest cluster. If the creation of a new cluster results in more than B child nodes of the parent cluster, the parent cluster is split. To ensure that the tree remains in balance, the nodes above it might need to be split recursively [47].

2.3.2 *Autoencoders*

An autoencoder is a type of artificial neural network used to learn a compressed representation of unlabeled data. As an unsupervised deep learning method, the autoencoder and its variants have been widely used due to its excellent performance in data dimensionality reduction [48], image processing [49], and anomaly detection [50].

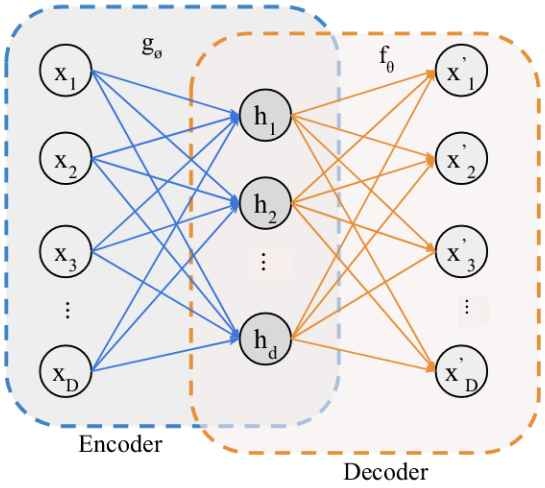


Figure 6: Structure of an autoencoder.

An autoencoder consists of two parts: an encoder and a decoder [51], as illustrated in Figure 6. The encoder maps an input $x_i \in \mathbb{R}^{d_x}$ to a reduced hidden representation $h_i \in \mathbb{R}^{d_h}$ by a function g_ϕ,

$$h_i = g_\phi(W_{x_i}). \tag{1}$$

where g is either the identity function for a linear projection or a sigmoid function $\frac{1}{1+e^{-W_x}}$ for a nonlinear mapping. The parameter W is a $d_h \times d_x$ weight matrix. Then encoder projects the incoming data to a smaller dimension by ignoring the noise in the data.

The decoder reconstructs $x'_i \in \mathbb{R}^{d_x}$ from the hidden representation h_i

$$x'_i = f_\theta(W'_{x_i}). \tag{2}$$

The parameter W' is another $d_x \times d_h$ weight matrix, which can be W^T. Similar to g_ϕ, f_θ is either the identity function for a linear reconstruction or a sigmoid function for a binary reconstruction.

To model the relationship between the data, the decoder reconstructs a set of instances indexed by $\Omega_i = j, k...$ with specific weights $Si = s_{ij}, s_{ik}....$ The weighted reconstruction error is

$$e_i(W, W') = \sum_{j \in \Omega_i} s_{ij} L(x_j, x_i') \tag{3}$$

where L is the reconstruction error. Generally, the squared error $L(x_j, x_i') = \|x_j - x_i'\|$ is used for linear reconstruction and the cross-entropy loss $L(x_j, x_i') = -\sum_{q=1}^{d_x} x_j^q \log(x_i'^q) + (1 - x_j^q)\log(1 - x_i'^{(q)})$ for binary reconstruction [52].

The total reconstruction error E of n samples is

$$E(W, W') = \sum_{i=1}^{n} e_i(W, W') = \sum_{i=1}^{n} \sum_{j \in \Omega_i} s_{ij} L(x_j, x_i') \tag{4}$$

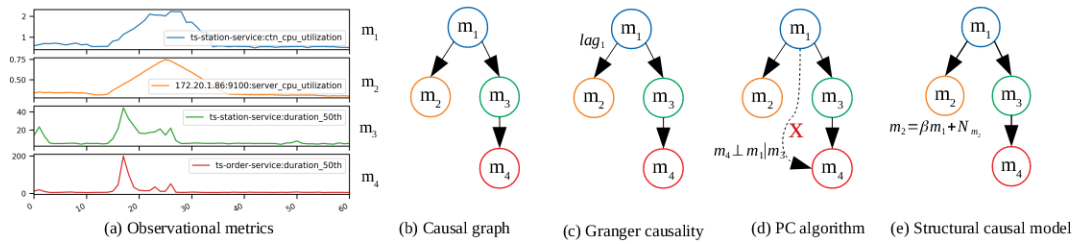
The training of autoencoders aims at optimizing the parameters (W, W') by minimizing the reconstruction error E. In the end, the decoder function approximates the encoded version back to the original dimensions.

2.3.3 *Causal inference techniques*

Causal inference techniques have been used extensively to infer causal relations from observational data, which is influential in such diverse fields as epidemiology [53], sociology [54], and machine learning [55–57]. Here we present causal inference techniques from three categories, which have been experimentally studied and partly used in our method.

2.3.3.1 *Granger causality*

Granger causality (GC) is one of the most common methods for inferring the causal relationship between time series based on temporal precedence. A



(a) Observational metrics (b) Causal graph (c) Granger causality (d) PC algorithm (e) Structural causal model

Figure 7: Overview of causal inference techniques.

time-series $X = \{x_1, x_2, ..., x_t, ...\}$ is said to Granger-cause another time series Y if the inclusion of information about the past of X significantly increases the predictive accuracy of the current value y_t of Y compared to a prediction based on the past values of Y alone, which can be explained by the following two models [58].

The first model, named *restricted model*, calculates the extent to which the time series X, Y can be predicted with their own past (x_{t-j} and y_{t-j}, with $j = 1, 2, ...$), resulting in the residual error variance $\Delta_1 = var(\delta_{1t})$ and $\Gamma_1 = var(\gamma_{1t})$. The model order is denoted by P, which indicates how many previous time points are considered, and the length of the time series by T, with P < T.

$$x_t = \sum_{j=1}^{P} a_j x_{t-j} + \delta_{1t}, \quad y_t = \sum_{j=1}^{P} a_j y_{t-j} + \gamma_{1t} \tag{5}$$

In the second, called *unrestricted model*, the prediction is based on both the own past of the time-series as well as the past of the other time series. This results in the residual error variance $\Delta_2 = var(\delta_{2t})$ and $\Gamma_2 = var(\gamma_{2t})$. The linear influence from x to y $\mathcal{F}_{x\to y}$, and from y to x, $\mathcal{F}_{y\to x}$, can be calculated as the ratio between the variances of the residual errors in two models, i.e.,

$$x_t = \sum_{j=1}^{P} a_j x_{t-j} + \sum_{j=1}^{P} b_j y_{t-j} + \delta_{2t}, \quad y_t = \sum_{j=1}^{P} a_j y_{t-j} + \sum_{j=1}^{P} b_j x_{t-j} + \gamma_{2t} \tag{6}$$

A reduction of error variable with including the past of another time series results in a larger F-ratio, the difference G-causality, i.e., $\mathcal{F}_{x\to y} - \mathcal{F}_{y\to x}$, is calculated to assess the dominant direction of information flow between x and y.

$$\mathcal{F}_{x\to y} = \ln\frac{var(\gamma_{1t})}{var(\gamma_{2t})} = \ln\frac{\Gamma_1}{\Gamma_2}, \quad \mathcal{F}_{y\to x} = \ln\frac{var(\delta_{1t})}{var(\delta_{2t})} = \ln\frac{\Delta_1}{\Delta_2} \tag{7}$$

Figure 7(c) states that the metric m_1 is G-causing m_2 if lag_1 is detected between them. GC [58] was originally formalized as a predictive model using vector autoRegressive (VAR). However, the regression model used assumes linearity and stationarity, and the accuracy of the causal inference is strongly affected by the predefined model order (which indicates how many prior time points are included in the regression). Some extensions of the basic GC concept include modeling nonlinear dependencies with transfer entropy, including multiple variables in GC, etc. Nevertheless, GC is limited to lagged causal dependencies and has known shortcomings for contemporaneous effects in sub-sampled time series [59].

2.3.3.2 *Causal network learning algorithms*

One of the most advanced theories widely recognized in the discovery of causality is the causal Bayesian network (CBN), which uses a Directed Acyclic Graph (DAG) to represent the causal relationships. There are two main approaches for learning a DAG: score-based approaches and constraint-based approaches. GES [60] is a score-based approach. It starts with an empty graph and then greedily searches for a Markov equivalent class that maximizes a score function. The PC algorithm [56] is one of the state-of-the-art constraint-based approaches. It starts with a complete, undirected graph and recursively deletes edges based on conditional independence tests. For example, in Figure 7(d), the edge between metric m_1 and m_4 is deleted as $m_4 \perp m_1 | m_3$. Once the skeleton is obtained with conditional independence tests, the edges are oriented with D-separation [61].

Learning a DAG from data is highly challenging and complex because the number of possible DAGs is super-exponential to the number of nodes. When applied to a high-dimensional dataset, the runtime of the PC algorithm is exponential to the number of nodes in the worst case. Nevertheless, if the true underlying DAG is sparse, which is often a reasonable assumption, the runtime is reduced to a polynomial run. There are some other methods, such as FCI [56], that can account for unobserved direct common causes and still partially identify which links must be causal. PCMCI [62] incorporates time-order as a constraint (causes precede effects) and utilizes a set of causal orientation rules to identify causal directions.

2.3.3.3 *Structural causal models (SCMs)*

The key idea of SCMs is to exploit the invariance of structural equations without committing to a specific functional form [63]. It assumes that the value of each variable is a deterministic function of its direct causes in the graph and some independent factors, such as unmeasured disturbances. A general SCM is defined as Equation 8, where g and f are arbitrary functions, X is the random input variables, and E is the environmental random variable that is independent of X. We say that X causes Y if they satisfy a generative process, such as Equation 8.

$$Y = g(f(X), E). \tag{8}$$

Several methods have been proposed for causal inference in the SCMs. Many of these methods rely on independence testing, where the algorithms model the data as Equation 8 (and vice versa) and decides upon the side

that provides a better fitting in terms of mapping accuracy and independence between $f(X)$ and E. The linear non-Gaussian acyclic model (LiNGAM) [64] algorithm assumes that the SCM takes the form of $Y = \beta X + E$, where $X \perp\!\!\!\perp E$, and $\beta \in \mathbb{R}$ and E is non-Gaussian. The LiNGAM learns β such that X and $Y - \beta X$ are independent by applying independent component analysis (ICA). In our example Figure 7(e), the effect metric m_2 can be defined as a function of the causal metric m_1 with an Non-Gaussian noise N_{m2}.

The ANM approach [65] extends the LiNGAM model and assumes that the effect is a function of the cause: $Y = f(X) + E$, where $X \perp\!\!\!\perp E$. The function is trained to map between Y and X, and then kernel independence tests are used to test whether X and $Y - f(X)$ are independent.

3

RELATED WORK

Contents

In this chapter, we present the related work in performance diagnosis and automatic recovery in the literature.

3.1 PERFORMANCE DIAGNOSIS

In the literature, many approaches have been proposed to diagnose root causes of performance problems, particularly in domains like clouds [41, 66], networks [67], and microservices [68]. Overall, these approaches are based on the observational data collected from the system, such as logs, traces, and metrics.

3.1.1 *Logs-based methods*

System logs are helpful in understanding the operations. When a performance problem occurs, operators usually examine recorded logs to gain insights into the issue and identify the potential root causes. Traditionally, operators perform simple keyword searches (such as error or exception) of logs that may be associated with the issue. Such a manual searching approach is often time-consuming and error-prone. In order to automate the problem diagnosis from system logs, many approaches have been proposed [69–81].

The primary idea to diagnose root causes from logs is to build a reference model for the normal operations, and a deviation from normal patterns implies the hidden root causes. DeepLog [70], LogSayer [71], Logsy [82], [72], LogCluster [73], and DISTALYZER [74] learn the normal patterns with machine learning. DeepLog [70] models a system log as a natural language sequence with a deep neural network utilizing Long Short-Term Memory (LSTM). It automatically learns log patterns from normal execution, detects anomalies, and performs root cause analysis when log patterns deviate from

the model trained from log data under normal execution. LogSayer [71] represents the system state by identifying suitable statistical features (e.g., frequency, surge), which are not sensitive to the exact log sequence first, then measures changes in the log pattern when an anomaly occurs. LogSayer uses LSTM neural networks to learn the historical correlation of log patterns and applies a backpropagation neural network for adaptive anomaly decisions. Logsy [82] uses an attention-based encoder model to learn log representations to distinguish the differences between normal and anomalous logs. Mi et al. [72] propose an approach that first classifies user requests with a clustering algorithm, then applies principal components analysis to extract the primary methods, and finally compares behaviors of the primary methods in normal and abnormal statues to locate the potential causes. Similarly, LogCluster [73] clusters the logs and utilizes a knowledge base to check if the log sequence occurred before, thus significantly reducing the number of logs that should be examined, meanwhile improving the identification accuracy. LogDC [83] pinpoints the root causes in terms of abnormal declarative in deployment files and log entries for cloud-native applications. It employs clustering to separate normal and abnormal deployments and build reference models based on normal samples. Then it localizes the anomaly by comparing the current deployment file and the reference deployment. DISTALYZER [74] uses machine learning techniques to compare system behaviors extracted from the logs and automatically infers the strongest associations between system components and performance. Decaf [84] leverages Random Forest models along with custom scoring functions to identify the root causes from logs that correlate to performance degradation.

Some approaches [75, 76] represents the normal pattern with graph models by mining the log data. Jia et al. [75] build a hybrid graph model to capture normal execution flows of intra-services and inter-services. They model the intra-service request flows as a service topology by utilizing the frequency of logs, and model the inner-service request flows as a time-weighted control flow graph from logs of each service. Pooja et al. [76] focus on the error rate of runtime logs and transform them into multivariate time series for inferring the causal graph, which is used to discover the ranked list of possible faulty components in microservices.

LOGAN [77] utilizes log correlations to construct behavioral reference models from logs that represent the normal patterns. When a problem occurs, it enables operators to inspect the divergence of current logs from the reference model and highlights logs likely to contain hints to the root cause. Uilog [78] uses a fault keyword matrix to classify logs by the fault type in real-time and locates the root causes with log correlation analysis. Draco [79] uses a

top-down approach, starting with identifying user interactions that may have failed (e.g., dropped calls) and drilling down to identify groups of attributes that best explain the difference between failed and successful interactions with a scalable Bayesian learner.

POD-Diagnosis [80] builds fault trees to capture potential root causes by using process context to log analysis.

Some approaches are specifically for diagnosing issues in MapReduce systems [81, 85], Openstack [86], Spark [87]. For example, Kahuna [85] diagnoses the performance issues in MapReduce systems using peer-similarity. It detects a node that behaves differently as the likely culprit of a performance issue if they behave alike in normal operations. Instead of extracting execution sequence from logs for all system tasks, Yuan et al. propose an approach that automatically recognizes the exception logs of ERROR logging level generated by a system task, which are critical snippets of execution traces for failure diagnosis [86]. Sherlog [88] uses runtime logs to diagnose failures in source code.

Even though logs-based approaches can discover more informational causes, they are hard to work in real-time and require abnormal information hidden in logs.

3.1.2 *Traces-based methods*

One important tool for diagnosing performance anomalies in distributed systems is tracing, which can precisely record the execution paths and locate the root causes by instrumenting to the system or source code of applications. To obtain the tracing and diagnosis problems, many tools have been proposed in the academia and industry [89–104].

Many tools traces the requests of applications in order to locate the faulty service, such as X-Trace [90], Pinpoint [92], Google's Dapper [100], Facebook's Canopy [103], X-ray [98], PreciseTracer [105] and the Mystery Machine [93]. Pinpoint traces the client requests and uses data mining techniques to correlate these requests' believed failures and successes to determine which components are most likely to be at fault. PreciseTracer debugs performance problems of multi-tier services by constructing a component activity graph to present causal paths of requests. Magpie monitors the fine-grained events generated by kernel, middleware, and application components with low-overhead instrumentation and accurately extracts requests and constructs representative models of system behavior. By constructing concise workload models, it can predict the performance and detect changes in the system. In addition,

distributed tracing tools, like Zipkin[1] and Jaeger[2] are developed for microservice architecture. Distributed tracing enables the operators to track the path of each transaction as it travels through a distributed system and analyze the interaction with every service it visits. This capability helps the operators deeply understand the performance of every service, thus helps pinpoint where failures occur and what causes poor performance.

Some tools are for the kernel or system (e.g., Perfscope [99] and Pip [95]). Perfscope collects system call trace via kernel tracing tools, such as Linux Tracing Tool next generation (LTTng)[3] and aims at performance anomalies that are caused by anomalous interactions between the application and the kernel. Pip instruments to the system to log the actual system behavior, such as the system's communication structure, timing, and resource consumption, and detects structural errors and performance problems in distributed systems by comparing actual behavior and the expected behavior. Some other tools are for software bugs [96–98] by collecting function-level traces from each process. If the application has issues, the traces are compared automatically, detecting the anomalies by identifying processes that stopped earlier or behaved differently from the rest.

Specifically, to diagnose the root causes from tracing data, Mi et al. [106] cluster the end-to-end request tracing into different categories according to request call sequences and select major categories, then extract the abnormal principal methods that might be causes of performance degradation. The Mystery Machine [93] uses traces to create a causal model to show how software components interact during the end-to-end processing or request. Then it uses the causal model to perform several types of distributed systems performance analysis: finding the critical path, quantifying slack for segments not on the critical path, and identifying segments correlated with performance anomalies. Sambasivan et al [107] builds end-to-end request-flow tracing within and across components and identifies the root causes of changes by comparing request flows from two executions (e.g., of two system versions or periods).

GMTA [108], MEPFL [109], and MicroRank [110] diagnose problems in microservices. GMTA abstracts traces into different paths and further groups them into business flows, represented as a graph, for understanding the architecture and diagnosis problems. MEPFL trains prediction models at both the trace level and the microservice level using the system traces collected from automatic executions of the target application and its faulty versions produced by fault injection. The trained prediction models are then used in

1 Zipkin - https://zipkin.io/
2 Jaeger - https://www.jaegertracing.io/
3 LTTng - https://lttng.org/

the production environment to predict latent errors, faulty microservices, and fault types for traces captured at runtime. MicroRank locates the root cause by identifying the abnormal traces, differentiating the importance of traces to extended-spectrum techniques, and ranking based on the weighted spectrum information from PageRank Scorer.

Although traces are beneficial to debug distributed systems and software, the overhead brought by these tools is considerable. Distributed systems today may generate millions of events per second, resulting in traces consisting of billions of events. Such large traces can overwhelm existing trace analysis tools. In addition, deploying these tools is also a daunting job requiring the developers to understand the source code well to instrument the tracing code.

3.1.3 *Metrics-base methods*

Metrics along with logs and traces are the three pillars of observability for capturing the system states and the quality of services. Compared to logs and traces, metrics are able to manifest the abnormality in the system and require no instrumentation to the source code. Many approaches in the literature employ observational metrics to infer the root causes of performances issues in distributed systems, cloud, network, and microservices [111–118].

One of the dominant methods for performance diagnosis is to capture the dependency between components in the system, including software and hardware components, then infer the root causes from the constructed dependency graph [111, 113–117, 119–130]. Dependency inference learns the relationship between interconnected components for problem diagnosis, particularly when localizing the source of the problem that propagates across a distributed system. Specific insights sought by dependency inference include service dependencies, request or call paths, deployment information, and transaction tracking through a distributed system.

In order to infer dependency, various approaches have been proposed in the literature. Orion [119], Project5 [120], and E2EProf [121] use network traffic to infer the dependency, which requires no domain knowledge about the applications, so that they can operate in a black-box manner. For example, Orion [119] discovers the dependencies using packet headers and timing information in network traffic. Sherlock [123] and NetMedic [124] leverage application-level knowledge such as configuration information and historical failure/success records, along with network traffic information, to infer dependencies. MonitorRank [117] periodically generates service dependencies which are then leveraged by unsupervised machine learning models for suggesting root cause candidates.

Many approaches leverage causal inference techniques to construct the dependency graph [111, 113–116, 129–131]. Microscope [113] combines both network traffic and PC (named after its authors, Peter and Clark [56]) algorithm to infer the service causality graph by considering both communicating and non-communicating dependencies in microservices. CauseInfer [111] constructs a two-layered hierarchical causality graph, including service dependency graph and metric causality graph of each service, to show the dependency among metrics. MS-Rank [130] introduces the concept of implicit metrics and proposes a hybrid impact graph construction algorithm, using multiple types of metrics to discover causal relationships between services. MicroCause [129] applies a variant of the PC algorithm that considers the time-order of metrics to metrics from multiple layers in microservice systems to capture the causality among metrics. Sieve [115] and Loud [116] systems construct an anomaly propagation graph across all metrics using the Granger causality test. Sage [131] uses Causal Bayesian Networks to capture the dependencies between microservices.

With the dependency graph constructed, the next step is to rank the root causes in the graph. Random walk [114, 117, 125, 130, 132–134] and graph traversal [111, 113, 127] are the two mainstream methods. FluxInfer [128] applies a weighted PageRank algorithm to localize root cause-related KPIs. Loud [116] uses graph centrality algorithms to get the root causes. MonitorRank [117], MS-Rank [130], and AutoMap [132] customize the random walk with forward, selfward, and backward transitions. Additionally, MonitorRank considers internal and external factors and proposes a pseudo-anomaly clustering algorithm to classify external factors before the random walking. MS-Rank establishes a self-adaptive mechanism to update the confidence of different metrics dynamically according to their diagnostic precision. CloudRanger [114] and ServiceRank [134] use a second-order random walk to identify the root causes. ServiceRank also proposes a correlation calibration mechanism for the circuit breaker - a typical protection pattern in the microservice architecture. Microscope [113], CauseInfer [111], and FacGraph [127] identifies the root causes by traversing the dependency graph to find the root cause candidates. Microscope [113] traverses the constructed graph from the front-end service to find the root cause candidates and ranks them based on the similarity of metrics. CauseInfer [111] uses the depth-first search method to traverse the metric causality graph of each service to get the candidate culprit metrics and ranks with the significance of abnormality. FacGraph [127] leverages breadth-first ordered string (BFOS) to get the root causes. Additionally, MicroHECL [135] analyses possible anomaly propagation chains and ranks candidate root causes based on correlation analysis.

The second class of methods is based on machine learning. Machine learning approaches [118, 131, 136–138] diagnose root causes by identifying features that significantly deviate from the predicted values estimated by machine learning algorithms. This approach is based on the assumption that the anomalous causal feature has a significant deviation from its normal behavior. Seer [118] identifies the faulty services and which resource, like CPU overhead, causes the service performance degradation. It is a proactive method that applies deep learning to massive data to identify root causes. This approach requires source code instrumentation to get metrics; meanwhile, its performance may decrease when microservices are frequently updated. UBL [136] leverages Self-Organizing Maps to capture emergent system behaviors and predict unknown anomalies for cloud infrastructures. Sage [131] leverages unsupervised learning to identify the culprit of unpredictable performance in complex graphs of microservices. It applies counterfactuals through a Graphical Variational Autoencoder to examine the impact of microservices on end-to-end performance based on the microservice dependency graph. Scheinert et al. [138] uses Arvalus and its variant D-Arvalus, a neural graph transformation method that models system components as nodes and their dependencies and placement as edges, to identify the root causes from metrics.

Correlation analysis is also commonly used to identify the root causes [139–148]. They correlate the observed metrics with the problematic behaviors of systems to identify the root causes. [142] correlates SLA violation to the system-level metrics; [143] correlates system changes to the failure or success symptoms of systems; Fa [144] builds correlation models among monitoring data in a normal state by using clustering methods; [145] correlates time series data with event data. [139] correlates throughput and load of each server in an n-tier system. [140] correlates workloads with the metrics of application performance and resource utilization. FD4C [141] uses an online incremental clustering method to recognize access behavior patterns and correlates the workloads with the application performance/resource utilization metrics in a specific access behavior pattern. In addition, correlation analysis is also used in other methods, like Microscope [113], to characterize the abnormality of components.

FChain [149], PAL [150], and [151] pinpoint the root causes by localizing the change points in the observed time services data. FChain pinpoints faulty components immediately after a performance anomaly is detected. FChain first discovers the onset time of abnormal behaviors at different components by distinguishing the abnormal change point from many change points caused by normal workload fluctuations. Faulty components are then pinpointed

based on the abnormal change propagation patterns and inter-component dependency relationships. PAL can pinpoint the source of faulty components in distributed applications by extracting anomaly propagation patterns. PAL provides a robust critical change point discovery algorithm to accurately capture the onset of anomaly symptoms at different application components. PAL then derives the propagation pattern by sorting all critical change points in chronological order.

PeerWatch [152] and [153] compare the behaviors of peer machines or peer software components, under the assumption that the peers should perform similarly in normal status and the majority of peers in the system are fault-free and identify the outliers deviating from the similarity are the problematic nodes or components. Wang et al. [154] locates the root-cause metrics with the log anomaly score. They collect anomaly scores by log anomaly detection algorithm and identify root-cause metrics by robust correlation analysis with data augmentation.

For performance diagnosis, there is an ongoing trend to combine all the observational data, including logs, traces, metrics, customer tickets, and other information. Brandón et al. [155] use all the logs, traces, and metrics from previous failures to build anomaly graphs, then identify the root cause of a new anomaly by matching the anomalous pattern with previous knowledge.

3.2 AUTOMATIC RECOVERY

A wide variety of techniques and approaches have been proposed to recover the abnormality ([156–158] study the potential failures and fixes) in the cloud, networks, and distributed systems [159–163]. Some methods address functional failures, whereas others address performance failures. Some of them are about specific recovery strategy, such as reboot [164–166], checkpointing [167], self-adaptation [168–171], placement [172], load-balancing [173], [174] auto-scaling for container-based clusters, dynamic reconfiguration [175], recovery testing [176], and one of the techniques used in Recovery-Oriented Computing (ROC) [166] consists on isolating faulty components and replacing them with redundant ones whereas others focus on general recovery strategies [177–179]. Some of them work on planning and optimizing recovery policies to improve system performance best, using model-based or reinforcement learning methods [180–182]. Meanwhile, some work on selecting the optimal actions for the anomaly situation. We herein review the related work in general automatic recovery from the following aspects.

Rule-based approaches transfer the expert knowledge into IF-THEN rules and use policies to match the rules for reacting to the faults [183–187]. For ex-

ample, Rainbow [187] statically associates a set of action rules for each of the pre-identified failure causes. This kind of approach is easy to develop. However, formalizing the rules requires much expertise and is a time-consuming and challenging task. In addition, frequent human intervention to revise the rules is required to keep them up-to-date in dynamic microservices systems, which conflicts with the goal of automatic recovery.

More studies [184, 188–190] adopt Case-Based Reasoning (CBR) to provide failure recovery. This kind of method collects symptoms of failures and stores them in a problem experience repository as case-solution pairs. When a new fault occurs, it obtains the solution by matching the cases. The hypothesis of this technique claims that similar problems may be resolved by applying the same type of solution. This kind of approach can effectively avoid repeating past mistakes and adapt to the system's changes. However, similar problems in microservices commonly give rise to different symptoms due to technology heterogeneity and frequent updates. Thus, it is difficult and error-prone to apply case matching. Furthermore, the number of exposed metrics in microservices is very high; computing the similarity between these metrics would cause significant overhead and delay.

Learning-based approaches use reinforcement learning or deep learning to generate recovery policies [181, 182, 191] or commands [192] without human intervention. Q. Zhu et al. [182] use reinforcement learning to generate error type-oriented policies. H. Ikeuchi et al. [192] use seq2seq learning to generate the recovery command line. This kind of approach views the system as a black box and can adapt to the changes in microservices. However, it requires a large set of historical failure data to train the model, which is difficult to obtain in microservice systems. Our method can complement this approach to help recover newly updated services. Once the failure data are available, this learning-based method can provide another recommendation for the action.

Model-based approaches model different aspects of a healing process, such as the properties of the fault [177], the properties of the actions [193], or use theoretical techniques, like Markov decision theory [180, 193, 194]. Jyoti Shetty et al. [195] select the best applicable remediation technique by considering the impact and overhead of remediation technique, the severity of the fault, and priority of the application. K. R. Josh et al. [180] use Bayesian estimation to probabilistically diagnose faults and use the results to generate recovery actions. Madam [196] uses some utility functions to select the most suitable architectural variant to repair the fault. Consequences of recovery actions are considered in [197–199]. M.Fu et al. [197] define the impact of an action on service response times which are caused by the increasing requests introduced by different recovery patterns. However, the impact of a performance issue

in microservice systems, such as hardware failure, software bugs, resource contention, etc., cannot manifest in the number of requests. Others [198, 199] define their models based on probabilistic parameters learned from recovery history (e.g., the prior probability of the system being in a stable state after executing an action). However, these probabilistic parameters are difficult to obtain in frequently updated microservice systems. The analytic queuing models used in [200] to analyze the effects of recovery actions on applications.

In addition, some orchestration platforms where microservices run on have the capability of self-healing to help recover the issues. For example, Kubernetes enables healing by automatically restarting failed containers and rescheduling them when their host dies [28]. However, service performance degradation sometimes does not manifest itself as containers or nodes failure; thus, the auto-recovery mechanism of the platform can not be triggered. In addition, this capability probably impacts and delays the recovery time, even resulting in a severe service outage [11].

4

PROBLEM AND OVERVIEW

Contents

This chapter describes the formulation of the addressed problems, the assumptions, and the main challenges of automatic performance diagnosis and recovery in cloud microservices. It also positions the methods developed in this thesis in a general performance problem management system and provides a conceptual overview of our methods.

4.1 PROBLEM FORMULATION AND ASSUMPTIONS

This section provides the detailed formulation of the problems addressed in this thesis, namely coarse-grained cause localization, fine-grained cause localization, and recovery action selection, as well as the associated assumptions.

4.1.1 *Coarse-grained cause localization*

Giving a cloud microservices environment, it consists of a set of services $S = \{s_i\}_{i=1}^{n_s}$, running inside of containers $P = \{p_j\}_{j=1}^{n_p}$, and deployed on a number of (physical or virtual) hosts $H = \{h_k\}_{k=1}^{n_h}$, where n_s, n_p, and n_h are the number of services, containers, and hosts, respectively. We define all the services, containers, and hosts as a part of the components $C = \{S, P, H\}$ of the system.

To observe the states of the cloud microservices, a telemetry infrastructure is deployed alongside the components to provide a collection of metrics time series M. We denote M^c as exposed metrics of component $c \in C$, and m_i^c as an individual metric (e.g., response time, CPU utilization, disk IO read time) of component c, where i is the type of metric. In particular, the end-to-end service response times M_{rt}^S are used to detect performance anomalies. When

there is a performance problem in cloud microservices, a set of services S^a tends to be detected as anomalies in the corresponding metrics $m_{rt}^{S_a}$.

Based on the above definition, the coarse-grained cause localization problem is formulated as follows:

Coarse-grained cause localization: *Given a set of metrics* M *and detected performance anomalies* $m_{rt}^{S_a}$ *in cloud microservices, how can we best localize the faulty service* s_{rc} *from which the performance issue originates?*

In this problem, the identified faulty service s_{rc} is defined as the coarse-grained cause, and it provides *where* the performance degradation comes from. Notably, a performance anomaly tends to propagate across services in an interdependent cloud microservices environment; hence, in addition to the s_{rc}, a number of services relying on the s_{rc} are likely to be detected as anomalies.

4.1.2 *Fine-grained cause localization*

Similarly, a fine-grained cause localization problem can be formulated as

Fine-grained cause localization: *Given a set of metrics* M *and detected performance anomalies* $m_{rt}^{S_a}$ *in cloud microservices, how can we best localize the culprit metric* $m_{rc}^{s_{rc}}$ *that identifies not only the faulty service but also the detailed information about the anomaly?*

In this problem, the observable metrics and service performance degradation detection are the same as in the coarse-grained cause localization, but we take one step further into the metrics to get more details about the anomaly. We define the identified culprit metric $m_{rc}^{s_{rc}}$ as a fine-grained root cause, since it not only indicates the source of the performance anomaly s_{rc} but also explains *why* the anomaly occurs through the identified bottleneck metric m_{rc}. An example of a fine-grained cause can be $m_{cpu\_utilization}^{s_1}$, which represents the high CPU utilization of s_1 causing the performance degradation.

Fine-grained cause root localization is essential for performance recovery in cloud microservices since the identified details of the performance anomaly help reduce the number of potential corrective actions, which is beneficial to decrease the MTTR and the risk of executing delusive actions. For root causes that manifest themselves directly in the observational metrics, the identified culprit metrics are the actual causes; otherwise, the bottleneck metrics are the most likely symptoms that explain the performance anomalies.

4.1.3 *Recovery action selection*

Recovery action selection is the next step of performance diagnosis, which utilizes the identified root causes, determines the corrective action(s), and attempts to restore the system to a stable state.

Same as performance diagnosis, monitoring metrics M of all components are provided. After locating the root causes, the possible faulty service s_{rc} and/or the bottleneck metrics for the performance anomalies can be determined. Following that, a list of possible recovery actions $A = \{a_i\}_{i=1}^{n_a}$ can be obtained (e.g., from expert knowledge or operator-maintained scripts), where n_a is the number of actions. These potential recovery actions can be service restart, service scale-out/up, host restart, etc. Furthermore, we formulate the recovery action selection problem as follows:

Recovery action selection: *Given a set of metrics M, an identified faulty service s_{rc}, and a list of possible corrective actions A, how can we select the most appropriate action(s) to mitigate the anomaly?*

The most appropriate action defined in the problem statement is the one that can recover the performance degraded services without disrupting other functional services and also minimize the time of recovery.

4.1.4 *Assumptions*

We use the following assumptions throughout this thesis unless otherwise stated.

The first general assumption is the existence of observational metrics of cloud microservices that provide information about the system behaviors, application behaviors, and components dependencies, such as the service invocations and deployment information. For cloud microservices, there are many monitoring tools to understand the states of the system, including the hardware, OS, container runtime, orchestration platform, and applications. For example, Cloud Watch [201] in Amazon, Cloud Monix [202] in Microsoft Azure, and the open-source cloud-native monitoring tools Prometheus for Kubernetes clusters. In this thesis, we deploy a set of cloud-native monitoring tools alongside the cloud microservices environment to collect metrics.

The second general assumption is that the metrics used for performance diagnosis and recovery are capable of reflecting abnormal system behaviors and harboring a performance anomaly in the system. In our evaluations, we

manually inject performance anomalies into microservices and filter out the cases that fail to exhibit anomalous behaviors.

In addition to the general assumptions, performance diagnosis assumes that root causes are directly or indirectly reflected in the observational metrics. For the observable causes, the localization process is able to locate them as culprit metrics. Otherwise, we can only obtain the most likely visible symptoms, the intermediate causes, and give evidence about the actual causes. We aim to identify faulty services only for problems that can only be found by reproducing and debugging the application source code.

Regarding recovery action selection, apart from the assumption that the root causes have been identified, we additionally assume that a list of candidate corrective actions is available, and their properties are stored in a knowledge base. In practice, these feasible actions can be obtained from expert knowledge or previous experiences, and the latter are usually managed in the form of scripts or playbooks [37].

4.2 CHALLENGES OF PERFORMANCE DIAGNOSIS AND RECOVERY

In this section, we present the challenges of coarse-grained and fine-grained cause localization and recovery action selection in cloud microservices.

4.2.1 *Challenges of coarse-grained cause localization*

The challenges in locating the faulty service in cloud microservice can mainly be two-fold. On the one hand, the challenges lie in the following characteristics of cloud microservices:

- **Complex dependencies.** The number of services in cloud microservices can often be hundreds or thousands (e.g., Uber has deployed 4000 microservices [16]). These services are interdependent in their business logic implementation, highly distributed across the cloud infrastructure, and sharing a finite set of hardware and software resources. Consequently, the dependencies between services are much more complex than in traditional distributed systems. When a performance anomaly arises in one service, it tends to propagate widely and causes disruptions to a number of services. Locating the faulty service from many anomalous services with complex dependencies is a challenging task.

- **Heterogeneous services.** Technology heterogeneity [33], one key benefit of MSA, allows development teams to select the most appropriate

programming languages and technology stacks for each service. This polyglot design may cause the same type of performance problems to manifest themselves differently across heterogeneous services. These diverse anomaly symptoms make it difficult to identify the root causes.

- **Dynamic services and infrastructures.** In cloud microservices, services are updated frequently to meet customer demands (e.g., Netflix updates thousands of times per day [203]), and infrastructures constantly change to adapt to workloads and orchestration platform scheduling. For example, workloads vary temporally and spatially across services and servers in the environment; services can be dynamically created, migrated, and terminated to achieve resource efficiency or service reliability. This highly dynamic environment aggravates the difficulty of locating the root causes. Moreover, the dynamic nature of cloud microservices implies that historical failure data is limited, resulting in methods that need to learn from failure events, such as pattern recognition-based methods, not working.

- **A wide range of root causes.** For cloud microservices, the root cause of a performance anomaly can vary widely and evolve with the frequent updates of microservices. As a result, unseen performance issues with unknown root causes are not uncommon, and pinpointing them becomes difficult even for experienced operators. Especially when distinct root causes manifest themselves with similar symptoms in the metrics, it is much harder to find the root causes.

On the other hand, coarse-grained cause localization is challenged by the detection of performance anomalies. The scale, complexity, and dynamics of cloud microservices make it challenging to detect the anomalies accurately. For example, numerous normal instances are wrongly reported as anomalies, while sophisticated anomalies are overlooked. These false alarms in the detection phase would complicate the modeling of anomaly propagation and prolong the performance diagnosis.

4.2.2 *Challenges of fine-grained cause localization*

Except for the above challenges in coarse-grained cause localization, some new difficulties emerge while looking for the culprit metrics.

- **A large volume of anomalous metrics.** A performance problem is likely to result in a large number of components emitting anomalous metrics,

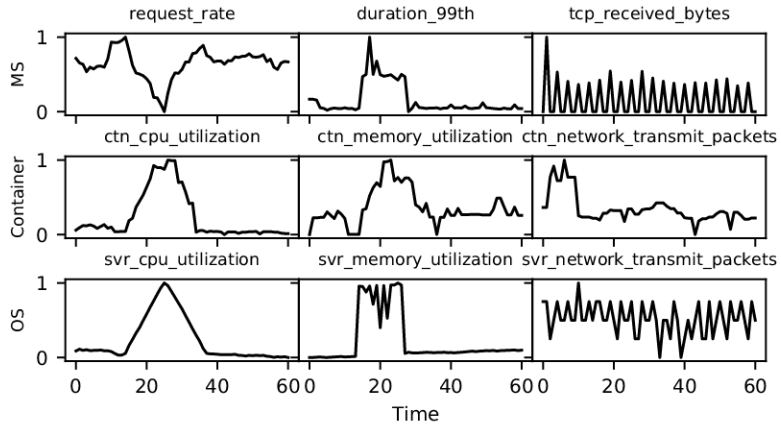


Figure 8: Multiple metrics become abnormal due to the anomaly propagation from a CPU hog issue in a service. Moreover, the anomaly patterns in metrics differ due to the heterogeneity and dynamic of cloud microservices as well as the control mechanisms in the system.

as the anomaly tends to propagate horizontally across interdependent services or resources, as well as vertically across multiple layers. To illustrate the phenomenon, Figure 8 shows that metrics from multiple data resources (MS, container, and OS) become abnormal when a service exhausts the allocated CPU resource. To complicate further, the number of services can be large in cloud microservices, and the number of available monitoring metrics is high (e.g., Netflix exposes 2 million metrics, and Uber exposes 500 million metrics [115]), which tend to magnify the volume of anomalous metrics. Identifying root causes from such a large number of anomalous metrics is analogous to finding a needle in a haystack.

- **Diverse anomaly patterns in metrics.** A performance problem tends to manifest different anomaly patterns in the observed metrics, varying with the types of metrics, frequently updated and polyglot services, and the control mechanisms in the system (e.g., load balancing, auto-scaling, restart, etc.). As shown in Figure 8, a CPU hog issue manifests itself differently in the metrics of duration, request rate, and resources. These diverse anomaly symptoms exaggerate the difficulty of modeling the anomaly propagation among metrics for locating the root causes.

4.2.3 *Challenges of recovery action selection*

In cloud microservices, deciding the most appropriate recovery action is difficult due to the following challenges:

- **Delusive corrective actions.** Due to the scale, complexity, dynamics, and diverse root causes in cloud microservices, the analysis of performance anomaly detection and diagnosis inevitably contains false positives. Such an incorrect analysis is likely to introduce delusive recovery actions in the candidates, which not only reduces the chance of selecting the best possible recovery actions but also increases the risk of executing the incorrect actions.
- **Limited historical failure data.** Due to the dynamics of services and cloud infrastructure, historical failure data is unavailable or is fast to become obsolete. Sames as performance diagnosis methods, learning-based recovery methods, which require a large amount of failure data to learn the recovery policies, are unlikely to make the appropriate decision to restore cloud microservices from performance anomalies.
- **Large state and action spaces.** The system state space is prominent in cloud microservices, varying with the behaviors of services and infrastructures, the type of performance problems, and the candidate recovery actions. The state space is further extended by the large number of services and machines of cloud microservices. The ample state space of cloud microservices complicates the creation of the system model and causes significant computation overhead and time delay during the decision-making process. Aside from the huge state space, the space of candidate actions can also be large due to the high cardinality root causes in cloud microservices.

Overall, it is challenging to determine the recovery actions in cloud microservices, where the decision is based on limited historical failure data, and predicting the consequences of actions requires tackling the uncertainties from large state and action spaces. More importantly, delusive corrective actions are inevitable in cloud microservices, and the risky ones that lead to severe disruptions must be avoided at all costs.

4.3 CONCEPTUAL OVERVIEW

To address the above challenges while considering the assumptions made in this thesis, we present multiple methods for automatic performance diagnosis

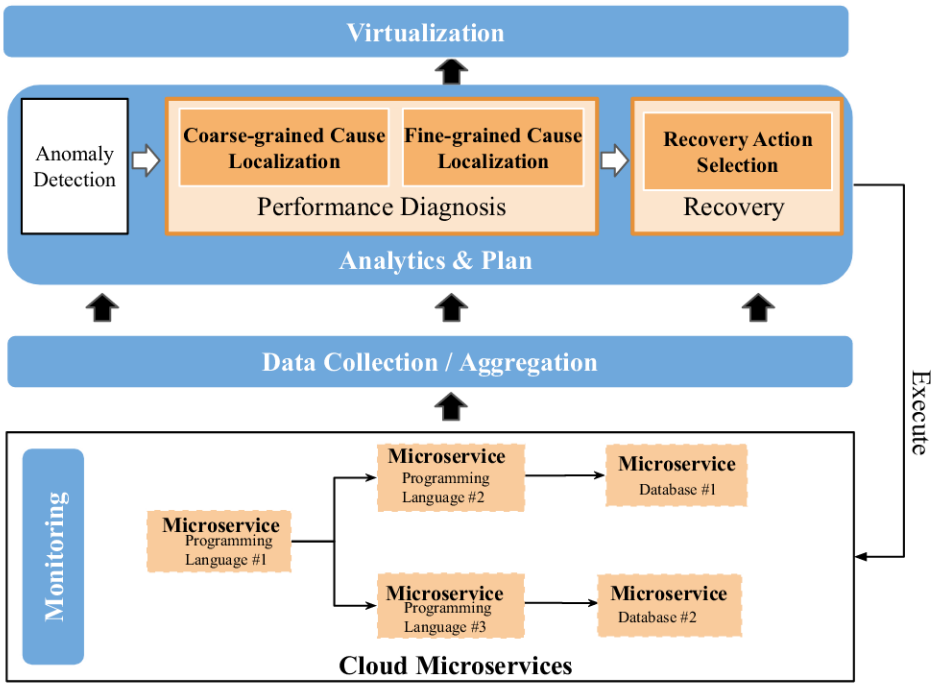


Figure 9: An overall architecture of performance problem management system for cloud microservices.

and recovery in cloud microservices. In this section, we position our work in the context of a performance problem management system customized for cloud microservices and further provide a conceptual overview of our methods.

4.3.1 *Performance problem management systems for cloud microservices*

Figure 9 shows an overall architecture of the performance problem management system for cloud microservices. It includes many microservices designed with heterogeneous programming languages and deployed in a containerized cloud computing system. All our performance diagnosis and recovery methods (highlighted in the orange boxes) are based on the monitoring metrics collected from the underlying cloud microservices environment, which is also the basis of performance problem management systems. In cloud microservices, monitoring tools are deployed inside to observe service performance, resource utilization, and other measurable performance metrics. Specifically, for applications, metrics such as response times, request rates,

and error rates are collected to indicate the high-level performance of the services; for system resources, metrics such as the utilization of CPU, memory, cache, and disk, the network transmit/received packets are collected to show the low-level system performance. These metrics are continuously collected, stored, aggregated, and forwarded to the analytics and plan module for analysis.

Before pinpointing root causes and deciding correctness, anomaly detection is required to identify the occurrence of performance anomaly. It monitors the service performance metrics and reports an anomaly when service performance deviates significantly from normal operations. In the literature, many methods [204–207] have been proposed to address the anomaly detection problem. Considering the challenges of limited historical data in cloud microservices and the requirement of minimal MTTR, we employ an unsupervised machine learning algorithm to detect performance anomalies in real-time without requiring any labeled historical data.

After detecting performance anomalies, the next step is to identify the root cause and determine the appropriate actions to restore cloud microservices to a stable state. Overall, each method presented in this thesis is designed to mitigate the challenges described earlier. For example, all our methods require no historical failure data that dynamic cloud microservices cannot provide. In addition, our methods are able to provide graphical views of cloud microservices, including component dependencies, anomaly propagation paths, potential consequences of corrective actions, and bottleneck services or metrics. These views can be visualized to the developers, operators, and management teams to provide better insights into performance anomalies, the system, and the design of microservices.

4.3.2 *Conceptual overview of our methods*

We then provide a conceptual overview of our methods and explain the challenges that they address.

In a cloud microservices environment, services are designed to communicate to each other to implement specific business logic and are deployed on distributed hosts. As shown in Figure 10, services $s_1 - s_5$ are deployed on hosts h_1 and h_2, and are interdependent along their invocations. In this environment, a large volume of monitoring metrics is observed. Once anomalies are detected on service performance metrics, coarse-grained cause localization, fine-grained cause localization, and recovery action selection are triggered to resolve the performance anomalies.

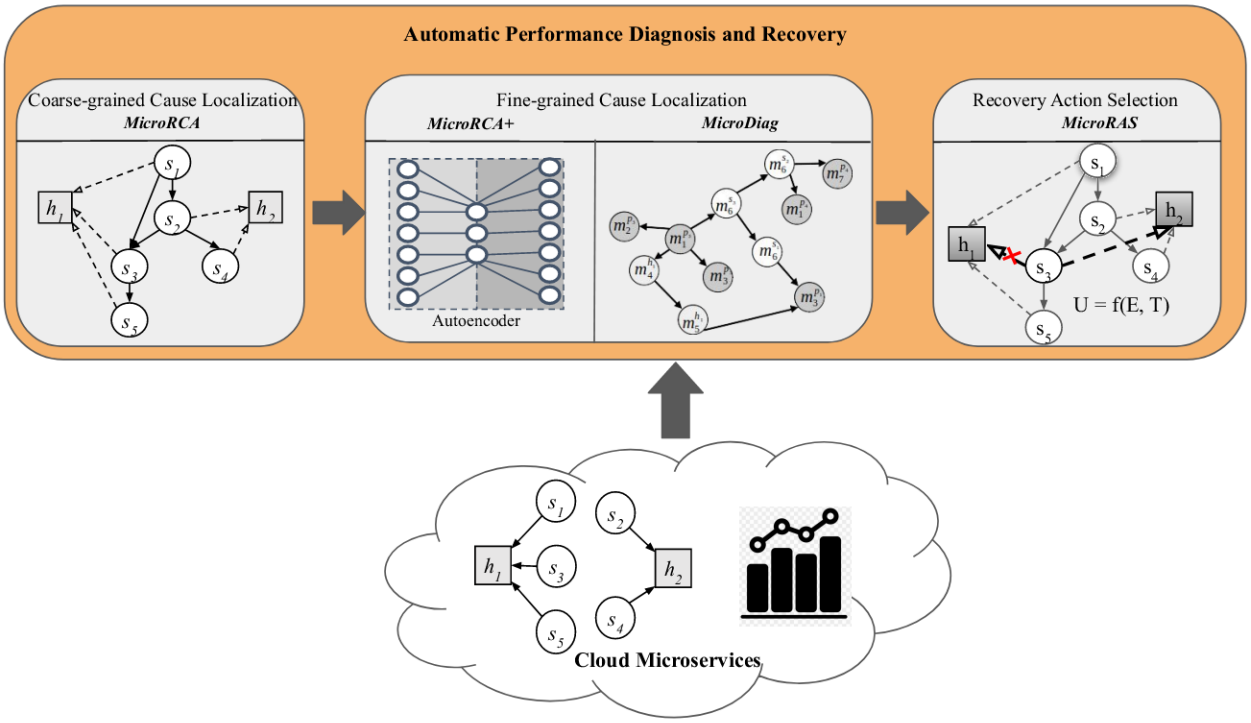


Figure 10: Conceptual overview of our performance diagnosis and recovery methods.

Coarse-grained cause localization. The challenges of coarse-grained cause localization complicate an accurate understanding of the origin of performance anomalies. The major questions are related to understanding how the anomaly propagates among services and which service is the source. We address these questions with a graph-based method named MicroRCA, which aims to obtain a ranked list of culprit services to serve as a guide for further troubleshooting and recovery. The key idea of MicroRCA is to construct an attributed graph with service and host nodes to model the anomaly propagation and then locate the bottleneck services by correlating service performance symptoms with corresponding resource utilization, which adapts well to the heterogeneity of microservices. Figure 10 gives an example of the attributed graph of the given cloud microservices. The constructed graph models the anomaly propagation not only along with service invocations but also through collocated hosts due to resource contentions. To reduce the number of services and metrics for analysis, MicroRCA leverages the detected anomalous services to prune the attributed graph in the localization. Overall, our method is application-agnostic, requiring no application instrumentation, and relying on real-time monitoring metrics only.

Fine-grained cause localization. Compared to coarse-grained cause localization, the main challenges of fine-grained diagnosis are the increased number of anomalous metrics and the diverse anomaly patterns. Using deep learning and causal inference, we address the above challenges with two methods: MicroRCA+ and MicroDiag.

The first method, MicroRCA+, is an extension of MicroRCA by using the identified faulty services. It applies autoencoders to the relevant metrics of a faulty service to detect the bottleneck that contributes to the service abnormality, assuming that the most likely culprit metric is the one deviates most significantly from its normal state. MicroRCA+ first employs autoencoders (the structure of the autoencoder is shown in Figure 10) to learn the normal patterns with metrics collected under normal operations, and then detects the anomalous metrics with the trained models and finally ranks them according to their reconstruction errors. Since MicroRCA+ leverages the results of coarse-grained cause localization, the number of anomalous metrics and the complexity of their anomalous patterns can be significantly reduced.

The second method, MicroDiag, identifies culprit metrics by modeling the anomaly propagation across metrics, represented as a metric causality graph, with Spatio-temporal causal inference. Figure 10 gives an example of metric causality graph. To obtain the graph, MicroDiag first analyzes the spatial propagation over components, beneficial to eliminate inference from unrelated metrics, and then uses a mixture of temporal CI on metrics from interactive layers to differentiate diverse anomaly patterns. In this regard, numerous metrics and diverse anomaly patterns in different layers are addressed.

In addition, we perform a comprehensive evaluation for understanding the overall performance of CI techniques on diagnosing root causes in cloud microservices, which is missing in the literature. Six representative CI methods from three categories are applied to locate the root causes of a range of performance anomalies injected to services in microservices benchmarks. The findings are reported and lay a foundation for the MicroDiag method.

Recovery action selection. One of the biggest concerns in recovery is to avoid the delusive actions or commands that deteriorate the system and cause a severe outage. Therefore, the core question of recovery is to select the appropriate action that resolves the issue but has little or no side-effects on other functional services, under the condition of limited historical failure data. To address this problem, we propose a model-based recovery action selection method named MicroRAS, which builds a graphic system-state model to predict the effectiveness of each candidate action, including the benefit of recovering the faulty service and the risk of affecting others. Figure 10 shows a graphic model and the prediction of the recovered state after an action is ex-

ecuted. In the end, the best possible action is selected from the candidates, with a tradeoff between the effectiveness of action E and the recovery time T.

The selected action can be evaluated by operators or executed directly to cloud microservices. The system states after the execution of the recovery action are then used to update the knowledge base for analysis or learning. All in all, MicroRAS can select the best recovery action in the absence of historical failure data and adapt well to the dynamics of cloud microservices. Moreover, the assessment of actions helps avoid the execution of delusive corrective actions inducted by false positives in performance anomaly detection and diagnosis.

5

COARSE-GRAINED CAUSE LOCALIZATION

Contents

Identifying the faulty service that explains *where* a performance anomaly originates in cloud microservices is challenging due to the complex dependencies, numerous metrics, heterogeneous services, and frequent updates. Existing methods either require instrumenting the application (e.g., [118, 208]) or analyzing numerous metrics (e.g., [115, 116]). The third class of methods [113, 114, 117] avoids these limitations by building a causality graph and inferring the causes along the graph using application-level metrics. These methods typically identify potential root causes by correlating back-end services with front-end services, which may fail to identify faulty services that have little or no impact on front-end services, implying they cannot adapt to the heterogeneity of microservices.

In this chapter, we present an application-agnostic coarse-grained cause localization method for cloud microservices, named MicroRCA[1], which locates the faulty service in real-time without requiring instrumenting to the application source code. MicroRCA constructs an attributed graph with services and hosts to model the anomaly propagation among services. This graph does not only include the service call paths but also include service collocated on the same (virtual) machines. MicroRCA correlates anomaly symptoms of communicating services with relevant resource utilization to infer the potential abnormal services and ranks the potential root causes. With the correlation of service anomalies and resource utilization, MicroRCA can identify abnormal non-compute intensive services with non-obvious service anomaly symptoms, mitigate the effect of false alarms to root cause localization and adapt to the heterogeneity of microservices.

In summary, our contributions are threefold:

1 Parts of this chapter are published in [209].

- We propose an attributed graph with service and host nodes to model the anomaly propagation in cloud microservices. Our approach is purely based on metrics collected at application and system levels and requires no application instrumentation.
- We provide a method to identify the anomalous services by correlating service performance symptoms with corresponding resource utilization, which adapts well to the heterogeneity of microservices.
- We evaluate MicroRCA by locating root causes from different types of faults and different kinds of faulty services. On average, of 95 test scenarios, it achieves a 13% improvement in precision over the state-of-the-art methods.

5.1 *microrca*: FAULTY SERVICE LOCALIZATION IN CLOUD MICROSERVICES

5.1.1 *Overview of MicroRCA*

Figure 11 presents the three main components, namely data collection, anomaly detection, and cause analysis engine, for locating root causes in cloud microservices. The data collection module continuously collects metrics from application and system layers. The application-layer metrics, particularly the end-to-end service response times, are used to detect performance anomalies, and metrics from both layers are used to locate root causes. Once performance anomalies are detected, the cause analysis engine is triggered to construct an attributed graph G with service and host nodes to represent the anomaly propagation in the system. Next, the engine extracts an anomalous subgraph SG by pruning the graph based on the detected anomalies and inferring which service is the most likely to cause the anomalies. The output of the cause analysis engine is a ranked list of faulty services, and the top service has the highest probability of being the root cause.

5.1.2 *Data collection*

MicroRCA is designed to be application-agnostic. It collects application and system levels metrics from a service mesh [210] and monitoring system separately and stores them in a time-series database. For cloud microservices, system-level metrics include container and host resource utilization, as illustrated by *container* and *host* in Figure 11. Application-level metrics include

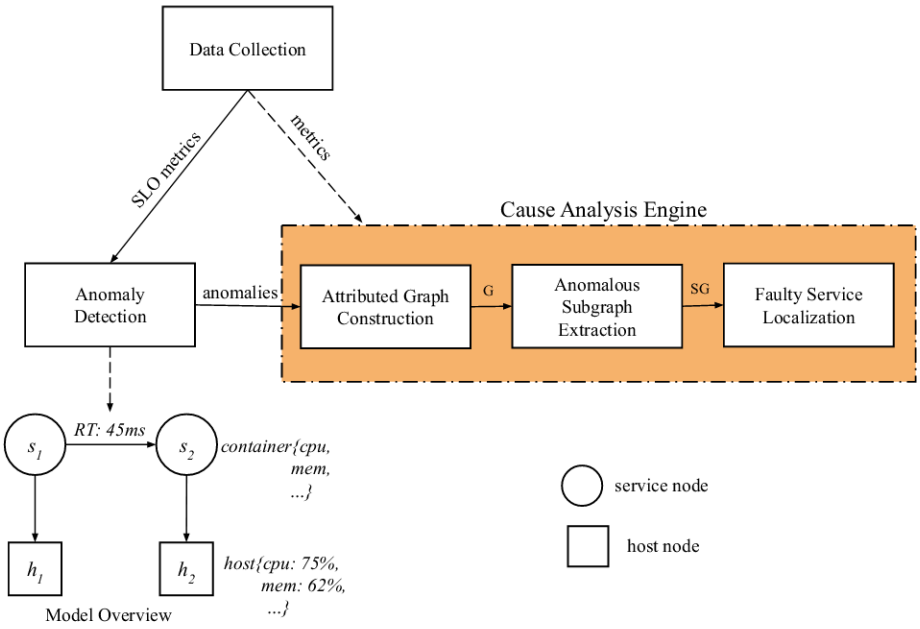


Figure 11: Overview of MicroRCA main components and root cause localization workflow.

response times between two communicating services, etc. (e.g., response time from service s_1 to service s_2 in Figure 11).

5.1.3 *Anomaly detection*

Anomaly detection is the starting point of root cause localization. Instead of detecting anomalies on the response times at the entry point of services [113], MicroRCA detects anomalies on end-to-end response times of communicating services, which not only proactively detect the user experience degradation but also avoids a severe outage due to prolonged anomaly propagation.

Anomaly detection is generally required to detect unexpected behaviors online in order to notify operators for a fast response. In addition, due to the challenges of large-scale and dynamic cloud microservices, the detection phase is required to handle a large number of data instances without labeling. To meet these requirements, an unsupervised learning algorithm named BIRCH [211], which is an online clustering algorithm for large datasets, is used to detect performance anomalies. In anomaly detection, end-to-end response times of microservices in a window size are collected and applied to the BIRCH algorithm. When the data points of a response time, a time series, are clustered into multiple microclusters, it is considered as an anomaly and vice versa.

5.1.4 *Cause analysis engine*

Once performance anomalies in end-to-end service response times are detected, the cause analysis engine starts to locate the root cause in services. The engine first constructs an attributed graph to capture the potential anomaly propagation through services and hosts in the cluster. Next, it extracts an anomalous subgraph from the attributed graph based on the detected anomalous services, thus narrowing down the service searching space. Lastly, it assigns attributes to the anomalous subgraph by correlating the service performance metrics and corresponding resource utilization and locates the faulty services with a graph centrality algorithm named Personalized PageRank [212]. The details about the engine are introduced in the next section.

5.2 THE MICRORCA METHOD

In this section, we describe the three procedures for identifying the faulty service that initiates the performance anomaly, namely attributed graph construction (Chapter 5.2.1), anomalous subgraph extraction (Chapter 5.2.2) and faulty services localization (Chapter 5.2.3).

5.2.1 *Attributed graph construction*

In the first step, MicroRCA constructs an attributed graph to represent the potential anomaly propagation paths in cloud microservices, which is based on the observation that anomalies propagate not only to services along the service call paths but also the services collocated on the same (virtual) machines [41, 213].

Our attributed graph consists of a set of service nodes $S = \{s_1, s_2, ..., s_{n_s}\}$ and host nodes $H = \{h_1, h_2, ..., h_{n_h}\}$ as shown in Figure 11. The service nodes are the services in the microservice-based application, and the host nodes are the (virtual) machines that services are deployed on. For each service node s_i, we add edges to all other service s_j it communicates with, and all hosts h_k it runs on.

We discover the graph nodes and their dynamic relationships by enumerating and parsing the metrics monitored at application and system levels. The graph nodes are interconnected as follows. (1) For the links between services, we parse the metrics collected from the service mesh, including the source and destination of a request. By enumerating and parsing these metrics, we get the dependencies among services. For example, we add a directed edge

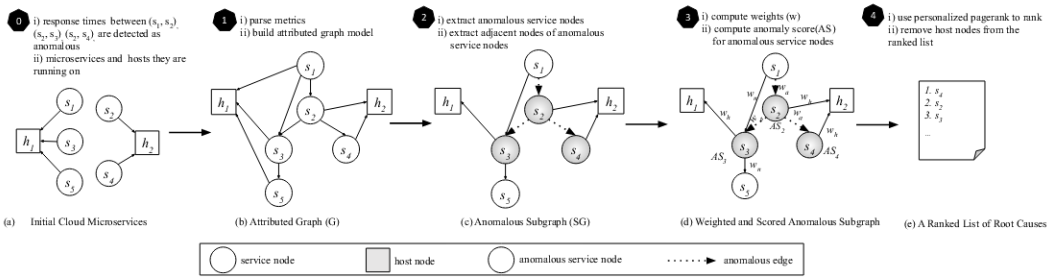


Figure 12: MicroRCA root cause localization procedures.

from s_i to s_j if service s_i sends requests to service s_j. (2) For the links between service and host, we parse the metrics collected from containers, including the service it containerizes and the host it runs on. If a containerized service s_i is allocated in host h_j, we add a directed edge from s_i to h_j. To the end, we get an attributed graph, where links between service nodes indicate the service invocations and links between service and host indicate the deployment information.

As the attributed graph shows the potential anomaly propagation graph, we use the metrics of a specific time frame before the anomaly was detected. We assume all microservices run reliably during this time frame, and the collected metrics can precisely provide relationships among all communicating services and located hosts. Figure 12(b) shows an example of a constructed attributed graph from the initial microservices environment.

5.2.2 *Anomalous subgraph extraction*

After constructing the attributed graph, we extract the anomalous subgraph based on detected anomalies and assign anomaly-related attributes to the nodes, to reduce the computation overhead and complexity of processing a large number of unaffected services. An anomalous subgraph SG is a connected subgraph representing anomaly propagation through anomalous services and hosts. Note that we use the response times between communicating services, which are the edges between service nodes in the attributed graph, as the anomaly detection targets. Once the response time of an edge is determined to be an anomaly, we consider the edge as an *anomalous edge* and the destination of the edge as an *anomalous node*.

To obtain the anomalous subgraph, we first extract the anomalous service nodes based on anomaly detection. In Figure 12(a), the response times between (s_1, s_2), (s_2, s_3), (s_2, s_4) are anomalous. Therefore, we extract the anomalous edges and take the destinations of the edges (s_2, s_3, s_4 in the example),

as the anomalous service nodes. Figure 12(c) depicts the anomalous edges as dashed lines and anomalous service nodes as shadowed nodes. Next, we compute the average of anomalous response times for each anomalous service node, denoted as $\mathtt{rt\_a}$, to a node attribute. Finally, we add nodes and edges that are connected to anomalous service nodes. Figure 12(c) shows the extracted anomalous subgraph, where adjacent services nodes s_1 and s_5 and adjacent host nodes h_1 and h_2 probably are impacted.

5.2.3 *Faulty services localization*

Once an anomalous subgraph is extracted, we start to locate the most likely faulty services. We first compute the similarity between connected nodes by weighing the subgraph, then assign anomaly scores to anomalous service nodes, and finally locate root causes based on a graph centrality algorithm.

5.2.3.1 *Anomalous subgraph weighing*

Edge weights represent the similarity between pairs of nodes. Similar to previous work [113, 114, 116, 117], we use the Pearson correlation function to measure similarity, henceforth denoted as $\mathtt{corr}(i, j)$. In particular, weights are computed in the following three ways:

- Weight of an anomalous edge w^a. From anomalous subgraph extraction (Chapter 5.2.2), we get a list of anomalous edges. To each anomalous edge, we assign a constant value $\alpha \in (0, 1]$, which denotes the anomaly detection confidence. If the real anomalies are correctly detected, we would set α higher, and vice versa. In Figure 12(d), weights $w_{(s_1,s_2)}$, $w_{(s_2,s_3)}$, $w_{(s_2,s_4)}$ are assigned α.
- Weight between an anomalous service node and a normal service node w^n. The weight is assigned with the correlation between two response times: the response time between an anomalous service node and a normal service node, and the average anomalous response time $\mathtt{rt\_a}$ of an anomalous service node. In Figure 12(d), the weight $w_{(s_1,s_3)}$ between the normal service node s_1 and the anomalous node s_3 is the correlation between the response time $\mathtt{rt}_{(s_1,s_3)}$ between s_1 and s_3, and the response time $\mathtt{rt}_{(s_2,s_3)}$ which is the average anomalous response time of s_3.
- Weight between an anomalous service node and a normal host node w^h. We use the maximum correlation coefficient between the average anomalous response time $\mathtt{rt\_a}$ of an anomalous service node, and the host

resource utilization metrics U_h, including CPU, memory, I/O, network, to represent the similarity between service anomaly symptoms and the corresponding resource utilization. Given that the response time of both normal and abnormal services have strong correlations with the host resource utilization, we take the average of in-edge weights w_{in} as a factor of the similarity. Thus, w_{ij}^{h} between service node i and host node j can be formulated as:

$$w_{ij}^{h} = \max_{k:u_k \in U_h(j)} corr(u_k, rt\_a(i)) \cdot \overline{w_I(i)}$$

Algorithm 1: Anomalous Subgraph Weighing

Input: Anomalous subgraph SG, anomalous edges, anomalous nodes, anomaly response time rt_a, host metrics U_h, response times of edges rt

Output: Weighted SG

```
for node v_j in anomaly nodes do
    for edge e_ij in in-edges of v_j do
        if rt_(i,j) in anomalous edges then
        Assign α to w_ij ;
        else
            Assign corr(rt_a(j), rt_(i,j)) to w_ij
        end
    end
    for edge e_jk in out-edges of v_j do
        if node v_k is service node then
        Assign corr(rt_a(j), rt_(j,k)) to w_jk ;
        else
            Assign avg(w_in(j)) × max(corr(rt_a(j), U_h(k))) to w_jk
        end
    end
end
return Weighted SG
```

To summarize, the weight w_{ij} between node i and node j is given by Equation 9, where $rt\_a(i)$ denotes the average anomalous response time of anoma-

lous service node i. Figure 12(d) shows these three types of weights along the edges.

$$w_{ij} = \begin{cases} \alpha, & \text{if } rt_{(i,j)} \text{ is anomalous,} \\ corr(rt_{(i,j)}, rt\_a(j)), & \text{if } i \in S \text{ and normal,} \\ corr(rt_{(i,j)}, rt\_a(i)), & \text{if } j \in S \text{ and normal,} \\ \text{as per Equation 5.2.3.1} & \text{if } j \in H \text{ and normal.} \end{cases} \tag{9}$$

The procedure of anomalous subgraph weighing is presented in Algorithm 1. In this algorithm, we iterate over the anomalous nodes and compute the weights for in-edges in lines L1-L8 and for out-edges in lines L9-L15.

5.2.3.2 *Assigning service anomaly score*

We calculate anomaly scores for anomalous service nodes and assign them to a node attribute, denoted as $AS \in [0, 1]$, to indicate its average effects on linked nodes and its abnormality.

To quantify the anomaly, we take an average weight of the service node $\overline{w(s_j)}$ that indicates the impact to linked nodes. Furthermore, In container-based microservices, we assume container resource utilization is correlated with service performance. We thus complement the service anomaly with the maximum correlation coefficient between the anomalous service node's average anomalous response time rt_a and its container resource utilization U_c. To summarize, given anomalous service node s_j, the anomaly score $AS(s_j)$ is defined as:

$$AS(s_j) = \overline{w(s_j)} \cdot \max_{k:u_k \in U_c(s_j)} corr(u_k, rt\_a(s_j)) \tag{10}$$

5.2.3.3 *Localizing faulty services*

We locate the faulty services from the anomalous subgraph with a graph centrality algorithm - Personalized PageRank [212], which proves a good performance in capturing anomaly propagation in previous work [114, 116, 117, 213]. In Personalized PageRank, the Personalized PageRank vector (PPV) $\boldsymbol{\nu}$ is regarded as the root cause score for each node, which shows the probability of being the root cause.

To compute PPV, we first define $\mathbf{P}$ as the transition probability matrix, where $P_{ij} = \frac{w_{ij}}{\sum_j w_{ij}}$ if node i links to node j, and $P_{ij} = 0$ otherwise. A preference vector $\mathbf{u}$ denotes the preference of nodes, which we assign the value

of anomaly scores of the nodes in our method. In this regard, the anomaly-related service nodes are visited more frequently when the random teleportation occurs. Formally, the personalized PageRank equation [212] is defined as:

$$\boldsymbol{v} = (1 - c)\mathbf{P}\boldsymbol{v} + c\mathbf{u} \tag{11}$$

Where $c \in (0, 1)$ is the teleportation probability, indicating that each step jumps back to a random node with probability c, and with probability $1 - c$ continues forth along with the graph. Typically $c = 0.15$ [212]. After ranking, we removed the host nodes from the ranked list as MicroRCA is designed to locate faulty services. Notably, as the link between service nodes in the anomalous subgraph represents the service call-callee relationship, we need to reverse the edges before running the localization algorithm. We give an example of the ranked list of root causes in Figure 12(e).

5.3 EVALUATION

In this section, we present the experimental setup, experimental results, a comparison with state-of-the-art methods, and discuss the characteristics of our approach.

5.3.1 *Experimental setup*

5.3.1.1 *Testbed*

We evaluate MicroRCA in a testbed established in Google Cloud Engine (GCE)[2] where we set up a Kubernetes cluster, deploy the monitoring system and service mesh, and run the benchmark named Sock-shop[3]. There are four worker nodes and one master node in the cluster; three of the worker nodes are dedicated to microservices and one for data collection. In addition, one server outside of the cluster runs the workload generator. Table 1 describes the detailed configuration of the hardware and software in the testbed.

5.3.1.2 *Benchmark*

Sock-shop[3] is a microservice demo application that simulates an e-commerce website that sells socks. It is a widely used microservice benchmark designed

2 Google Cloud Engine - https://cloud.google.com/compute/
3 Sock-shop - https://microservices-demo.github.io/

Table 1: Hardware and software configuration used in experiments.

Hardware Configuration			
Component	**Master node**	**Worker node(x4)**	**Workload generator**
Operating System	Container-Optimized OS	Container-Optimized OS	18.04.2 LTS
vCPU(s)	1	4	6
Memory(GB)	3.75	15	12
Software Version			
Kubernetes	**Istio**	**Prometheus**	**Node-exporter**
1.14.1	1.1.5	2.3.1	v0.15.2

to aid in demonstrating and testing microservices and cloud-native technologies. Figure 13 shows the architecture of sock-shop[3]. It consists of 13 microservices, which are implemented in heterogeneous technologies and intercommunicate using REST over HTTP. In particular, *front-end* serves as the entry point for user requests; *catalogue* provides a sock catalogue and product information; *carts* holds shopping carts; *user* provides the user authentication and store user accounts, including payment cards and addresses; *orders* places orders from *carts* after user log-in through the *user* service, then process the payment and shipping from the *payment* and *shipping* services separately. For each microservice, we limit the CPU resource to 1vCPU and memory to 1GB. The replication factor of each microservice is set with 1.

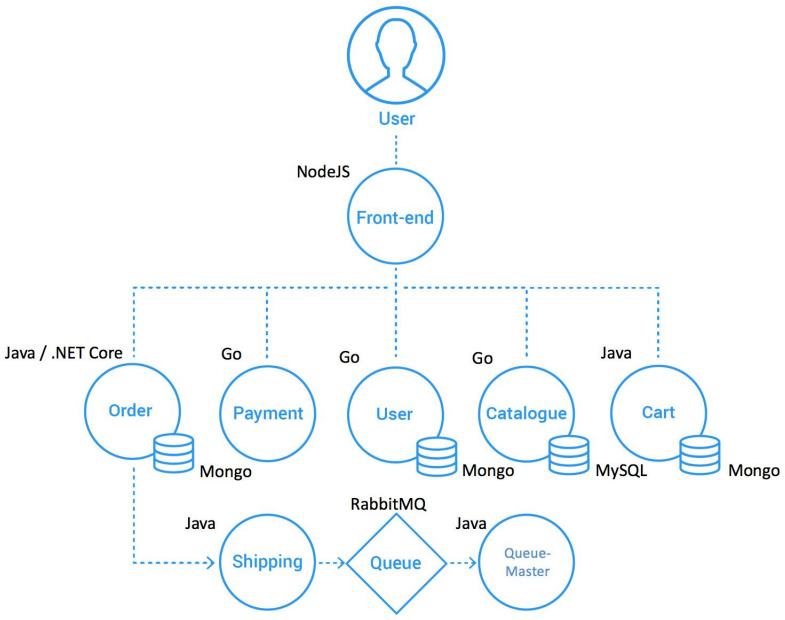


Figure 13: Sock-shop architecture.

Table 2: Request rates sent to microservices.

Microservices	front-end	catalogue	user	carts	orders
Concurrent users	100	100	100	100	100
Request rate(/s)	200	300	50	50	20

5.3.1.3 *Workload generator*

We develop a workload generator using Locust[4], a distributed, open-source load testing tool that simulates concurrent users in an application. The workload is selected to reflect real user behavior, e.g., more requests are sent to the entry points *front-end* and *catalogue*, and fewer to the shopping *carts*, *user* and *orders* services. We distribute requests to *front-end*, *orders*, *catalogue*, *user*, *carts* with five locust slaves, and provision 500 users that in total generate about 600 queries per second to sock-shop. The request rate of each microservice is listed in Table 2.

5.3.1.4 *Data collection*

We use the istio[3] service mesh, which in term uses Prometheus[1], to collect service-level and container-level metrics and node-exporter[4] to collect host-level metrics. Prometheus is configured to collect metrics every 5 seconds and sends the collected data to MicroRCA. At the service level, we collect response time between each pair of services. In both container-level and host-level, we collect CPU usage, memory usage, and the size of the total sent bytes.

5.3.1.5 *Faults injection*

Our method is applicable to any type of anomaly that manifests itself as increased microservice response time. In this evaluation, we inject three types of faults commonly used in the evaluation of the state-of-the-art approaches [113, 116, 152] to sock-shop microservices to simulate the performance anomaly. (1) Latency, we use the *tc* [5] to delay the network packets; (2) CPU hog, we use *stress-ng* [6], a tool to load and stress compute system, to exhaust CPU resources. As microservice *payment* is non-compute intensive whose CPU usage is only 50mHz, we exhaust its CPU heavily with 99% usage. (3) Memory leak: we use *stress-ng* to allocate memory continuously. As microservice *carts* and *orders*

4 Locust - https://locust.io/
5 tc - https://linux.die.net/man/8/tc
6 stress-ng - https://kernel.ubuntu.com/ cking/stress-ng/

Table 3: The detail of injected faults.

Microservices	front-end	catalogue	user	carts	orders	payment	shipping
Latency(ms)	200	200	200	200	200	200	200
CPU Hog(vcpu*%)	-	2*95	2*95	2*95	2*95	2*99	2*95
Memory Leak(vm*MB)	-	2*1024	2*1024	1*2048	1*2048	2*1024	2*1024

are CPU and memory-intensive services, and memory leak causes CPU overhead [211], we only provision 1 virtual machine. The details of the injected faults are described in Table 3.

To inject performance anomalies in microservices, we customize the existing sock-shop docker images by installing the above faults injection tools. Each fault lasts 1 minute. To increase the generality, we repeat the injection process five times for each fault. It produces a total of 95 experiment cases.

5.3.1.6 *Evaluation metrics*

To quantify the performance of each algorithm on a set of anomalies A, we use the following metrics:

- *Precision at top k* denotes the probability that the top k results given by an algorithm include the real root cause, denoted as PR@k. A higher PR@k score, especially for small values of k, represents the algorithm correctly identifying the root cause. Let $R[i]$ be the rank of each cause and v_{rc} be the set of root causes. More formally, PR@k is defined on a set of given anomalies A as:

$$PR@k = \frac{1}{|A|} \sum_{a \in A} \frac{\sum_{i<k}(R[i] \in v_{rc})}{(min(k, |v_{rc}|))} \tag{12}$$

- *Mean Average Precision* (MAP) quantifies the overall performance of an algorithm, where N is the number of microservices:

$$MAP = \frac{1}{|A|} \sum_{a \in A} \sum_{1 \leqslant k \leqslant N} PR@k. \tag{13}$$

5.3.1.7 *Baseline methods*

We compare MicroRCA with some baseline methods as follows:

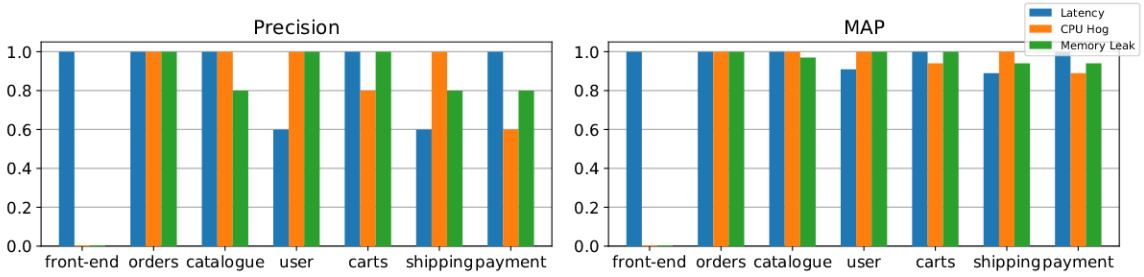


Figure 14: The result of PR@1 and MAP.

- Random selection(RS): Random selection is a way that an operating team uses without specific domain knowledge of the system. Every time, the operating team randomly selects one microservice from the uninvestigated microservices to investigate until they find the root cause.
- MonitorRank [117]: MonitorRank uses a customized random walk algorithm to identify the root cause. It is similar to the personalized PageRank[212] we adopt in MicroRCA. As we only consider the internal faults in microservices, we do not implement the pseudo-anomaly clustering algorithm to identify the external factors. To implement MonitorRank, we use their customized random walker algorithm to identify root causes in the extracted anomalous subgraph.
- Microscope [113]: Microscope is another graph-based method to identify faulty services in the microservices environment, and it also takes sock-shop as the benchmark. To implement the Microscope, we construct the services causality graph by paring the service-level metrics, then use their cause inference, which traverses the causality graph with the detected anomalous services nodes, to locate root causes.

The anomaly detection methods in MonitorRank and Microscope fail to detect anomalies, especially for non-obvious anomalies, like *payment* service. We aim to compare the root cause localization performance in our experiments. Thus, we use the same results from our anomaly detection module.

5.3.2 *Experimental results*

Figure 14 shows the results of our proposed coarse-grained cause localization method for different types of faults and sock-shop microservices. We observe that all services achieve a MAP in the range of 80%-100% and an average PR@1 over 80% except *shipping* and *payment*.

Table 4: Performance of MicroRCA.

microservics name	front-end	orders	catalogue	user	carts	shipping	payment	average
Latency								
PR@1	1.0	1.0	1.0	0.6	1.0	0.6	1.0	0.89
PR@3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
MAP	1.0	1.0	1.0	0.91	1.0	0.89	1.0	0.97
CPU Hog								
PR@1	-	1.0	1.0	1.0	0.8	1.0	0.6	0.9
PR@3	-	1.0	1.0	1.0	1.0	1.0	1.0	1.0
MAP	-	1.0	1.0	1.0	0.94	1.0	0.89	0.97
Memory Leak								
PR@1	-	1.0	0.8	1.0	1.0	0.8	0.8	0.9
PR@3	-	1.0	1.0	1.0	1.0	1.0	1.0	1.0
MAP	-	1.0	0.97	1.0	1.0	0.94	0.94	0.975

The average precision of *shipping* and *payment* that is lower than other services is likely for three reasons. First, *payment* is a non-compute intensive service; even though we exhaust the resources of CPU and memory, its response time is scarcely impacted. Second, there are few requests to *shipping* and *payment*, and they do not request any other services, which makes the response time increase less obvious than other faulty services. Third, to detect anomalies in their experimental cases, we use a small threshold in the anomaly detection module, which causes more false alarms.

Table 4 demonstrates the performance of MicroRCA in different types of faults and microservices. It shows that MicroRCA can achieve almost 90% in terms of PR@1 and effectively locate all root causes in the top three faulty services.

5.3.3 *Comparisons*

To evaluate the performance of MicroRCA further, we apply it and the baseline methods on all experimental cases.

We compare the overall performance of all methods and their performances in identifying different types of faults. Table 5 shows the performance, in terms of PR@1, PR@3, and MAP, for all methods. We can observe that MicroRCA outperforms the baseline methods overall. In particular, MicroRCA achieves a precision of 89% and MAP of 97%, which are at least 13% and 15% higher than the baseline methods separately. In general, Microscope per-

Table 5: Performance of each algorithm.

Metric	RS	MonitorRank	Microscope	MicroRCA	Improvement to MonitorRank (%)	Improvement to Microscope(%)
Overall						
PR@1	0.21	0.41	0.79	0.89	118	13.3
PR@3	0.46	0.65	0.86	1.0	53.2	15.9
MAP	0.58	0.73	0.85	0.97	33.7	14.7
Latency						
PR@1	0.17	0.23	0.66	0.89	287	34.8
PR@3	0.46	0.66	0.71	1.0	51.5	40.8
MAP	0.58	0.73	0.7	0.97	32.9	38.6
CPU Hog						
PR@1	0.23	0.6	0.87	0.9	50	3.5
PR@3	0.5	0.67	0.93	1.0	49.3	7.5
MAP	0.61	0.77	0.92	0.97	26	5.4
Memory Leak						
PR@1	0.23	0.43	0.87	0.9	109	3.5
PR@3	0.37	0.63	0.97	1.0	58.7	3
MAP	0.57	0.68	0.95	0.98	44.1	3.2

forms well in CPU hog and memory leak when the anomalies are detected correctly but worse in fault latency when more false alarms are detected. However, MicroRCA performs well in all types of faults and achieves an average improvement of 24% in MAP compared to MonitorRank and Microscope.

Next, we compare the performance of each method on different microservices. Figure 15 shows the comparison results, in terms of PR@1, PR@3, and MAP, on different services. We can see that MonitorRank performs well in identifying dominating nodes that have large degrees, like microservice *orders*. However, it fails to identify leaf nodes, like microservice *payment*. This is because MonitorRank calculates the similarity based on the correlation between front-end services and back-end services. The anomaly of microservice *payment* decreases the correlation during propagation, and thus the root cause localization fails. On the contrary, Microscope performs better in identifying leaf nodes, such as the microservice *payment*, but worse in identifying dominating nodes, like microservice *orders*. This is because the Microscope traverses the graph based on the detected anomalies and puts the anomalous child nodes into a list of potential causes, which makes it fail to identify the dominating nodes when alarms are reported from child nodes. Compared to

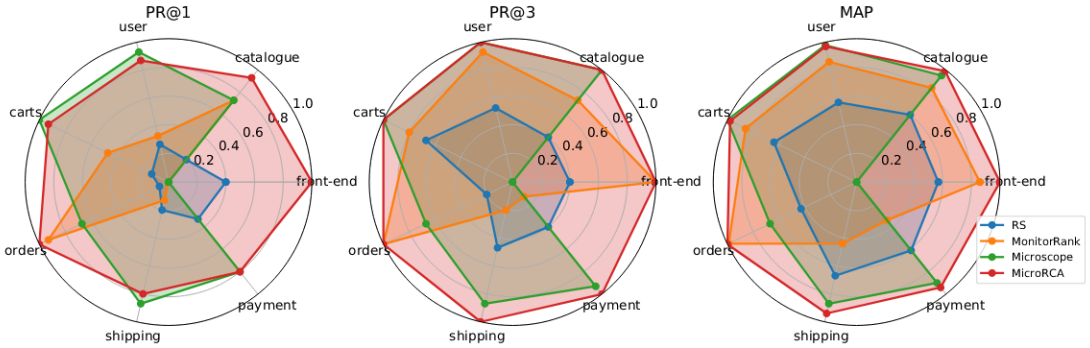


Figure 15: The comparison results of PR@1,PR@3 and MAP for different microservices.

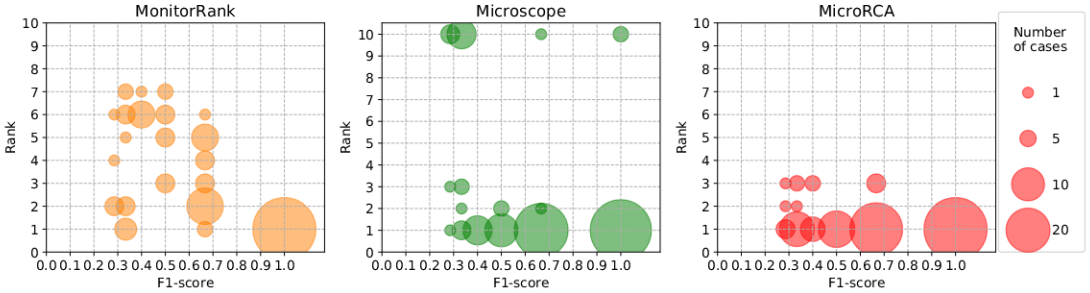


Figure 16: Root cause rank against anomaly detection F1-Score.

MonitorRank and Microscope, MicroRCA generally has a good performance for all the services, no matter dominating nodes or leaf nodes.

Finally, we compare each method's root cause localization performance against the anomaly detection results to evaluate their sensitivity to the performance of anomaly detection. We compute the F1-score of each experimental case and count the number of different ranks of all experiment cases to different F1-scores. F1-score is computed as the harmonic mean of precision and recall: $2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$. In our experiment, we only inject one fault to one microservice, which means only one root cause in a case. Thus in each case, the recall is 1 if the root cause is in the ranked list, otherwise 0. Precision is the number of detected root causes divided by the number of detected anomalies. As RS(random selection) is not related to anomaly detection results, we only analyze the MonitorRank, Microscope, and our MicroRCA.

Figure 16 shows the number of different ranks against F1-score for MonotorRank, Microscope, and MicroRCA separately. To show the results compactly, we set rank with 10, which is higher than the total number of microservices when the root cause is failed to locate. We can see all of them have a good performance when the faulty services are correctly detected. However, when false alarms are frequent, Microscope cannot identify root causes in some

cases; MonitorRank can identify root causes, but root causes are always low-ranked; MicroRCA identifies the root cause correctly in scenarios with high and low F1-score. In particular, MicroRCA identifies 13 out of 18 faults in the top 1 when F1-score is less than 0.4, which is higher than the other two methods.

5.3.4 *Discussion*

Here we discuss the overhead and sensitivity of MicroRCA.

5.3.4.1 *Overhead*

The overhead of MicroRCA on cloud microservices is mostly caused by the data collection module, which continuously collects the application-level and system-level metrics. Table 6 shows the overhead of data collection and the execution time of modules in MicroRCA. We can see that the execution time of MicroRCA is short enough to run in real-time, given a data collection interval of 5 seconds. However, the overhead of the data collection module is a little high. In the future, we would like to explore lightweight monitoring tools to reduce this overhead.

Table 6: The overhead of MicroRCA.

Modules	**Cost**
Data collection	0.6 vCPU and 1511MB RAM
Anomaly Detection	0.01s (8 cores)
Attributed Graph Construction	3.3s (8 cores)
Root Cause Localization	0.03 (8 cores)

5.3.4.2 *Sensitivity*

To evaluate the sensitivity of MicroRCA to the anomaly detection confidence(α), which is assigned to the anomalous edges in the weighing anomalous subgraph (Chapter 5.2.3). We analyze the performance of MicroRCA with different values of α. Figure 17 shows the performance of MicroRCA, in terms of overall PR@k(k=1,2,3) and MAP of different types of faults over all the test cases, when α ranges from 0.15 to 1.0. We can see that the performance of MicroRCA changes with the weight assigned to the anomalous nodes. PR@1 increases to the maximum when $\alpha = 0.55$ and drops when $\alpha \geqslant 0.7$; PR@3 is always 1 when α is less than 0.55, and drops after that. MAP of all types of faults is relatively stable and keeps over 96%. In this case, we choose 0.55

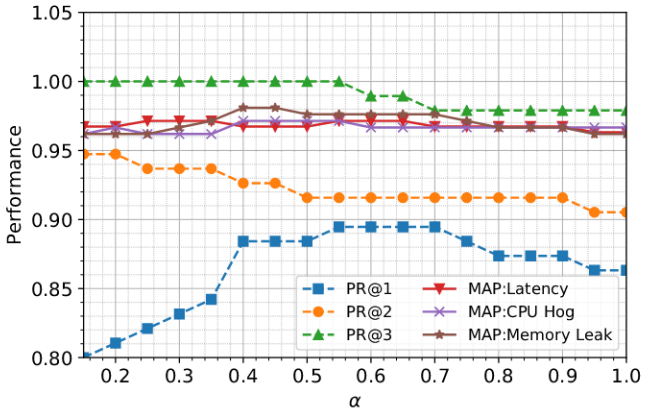


Figure 17: Performance against anomaly detection confidence(α).

as the weight of anomalous edges where the PR@1 and PR@3 are the maxima. In practice, as different anomaly detection methods have different performances, we need to tune the anomaly detection confidence(α) according to the anomaly detection module.

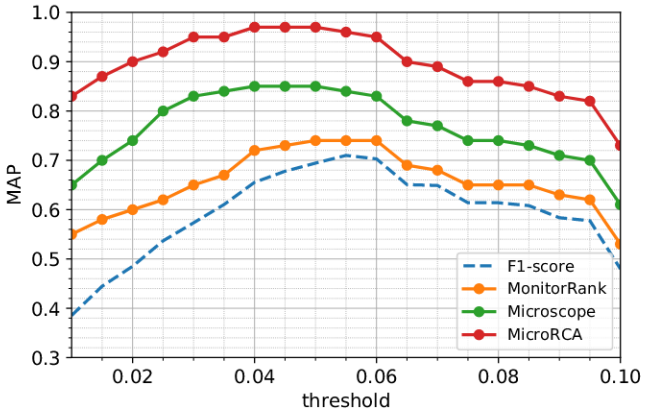


Figure 18: Performance against anomaly detection threshold.

Furthermore, we analyze the sensitivity to anomaly detection results with different thresholds in the anomaly detection module. Figure 18 shows the MAP of MicroRCA and other two baseline methods in solid lines and F1-score in dashed line when the threshold ranges from 0.01 to 0.1. Note that we set α to the constant 0.55 when MicroRCA regularly performs well. We can see that the three methods share a similar pattern with F1-score, and MicroRCA performs well in a range from 0.02 to 0.065, which is wider than the other two methods. As all three methods perform well when the threshold is 0.045, we use it in the anomaly detection module for all the above analyses.

5.4 CHAPTER SUMMARY

Locating the source of a performance anomaly is critical for cloud microservices recovery, but it is also challenging due to the complex dependencies, a large number of services, polyglot service design, and frequent updates. We presented MicroRCA, an application-agnostic method for identifying the faulty service that initiates a performance degradation in cloud microservices to mitigate these challenges. MicroRCA locates the bottleneck service in real-time by constructing an attributed graph to show the anomaly propagation among services and hosts and correlating service anomalous performance symptoms with corresponding resource utilization to infer the faulty service. We developed a prototype of our method and evaluated it on a microservice benchmark. The evaluation shows MicroRCA achieves 89% in precision and 97% in mean average precision (MAP). In particular, our method improves the precision in identifying root causes from both dominating services with large degrees and leaf services with non-obvious anomaly symptoms, where the state-of-the-art methods fall short. Furthermore, we discussed the sensitivity of the diagnosis performance against the anomaly detection results and the computation overhead of our method.

Nevertheless, it is still time-consuming and error-prone with the identified faulty service to determine the corrective actions for resolving the performance anomaly. Therefore, in the next chapter, we provide a fine-grained cause localization method that dives into the observational metrics to identify the evidence to explain why the performance anomaly occurs, aiming to reduce the number of candidate recovery actions.

6

FINE-GRAINED CAUSE LOCALIZATION

Contents

In this chapter, we aim to identify the fine-grained causes of performance anomalies in cloud microservices, which explains *why* a performance anomaly occurs or at least provides some insights. Apart from the challenges of the scale, dynamics, and complex dependencies between services in coarse-grained cause localization, fine-grained cause diagnosis is additionally challenged by the complexity of metrics (e.g., the massive number of metrics and their diverse anomaly patterns).

To pinpoint fine-grained causes, various approaches based on deep learning, pattern recognition, and causal inference have been proposed in the literature. However, approaches that use deep learning, such as Seer [118], require instrumentation of the application. Approaches based on pattern recognition suffer a high computation complexity along with the increased number of failure patterns and can only identify root causes of known issues; The third class of approaches based on causal inference constructs a causality graph with metrics from multiple components and may fail to localize root causes that manifest in diverse anomaly symptoms.

In this chapter, we present two methods, MicroRCA+ and MicroDiag, for fine-grained cause localization in cloud microservices. In the first method, MicroRCA+, we reduce the computation overhead and the inference of diverse anomaly symptoms from other services by leveraging the coarse-grained causes identified by MicroRCA (presented in Chapter 5). For a detected faulty service, MicroRCA+ applies autoencoders to its relevant metrics to locate the culprit metric that contributes to its abnormality. In the second method, MicroDiag, we apply Spatio-temporal cause inference into the metrics collected from all components and model the anomaly propagation across metrics

with a metric causality graph. A mixture of causal inference techniques is employed to capture the heterogeneous anomaly patterns, which differentiates the anomaly properties of metrics from interactive layers. This method is based on our findings from an experimental evaluation of CI techniques on locating root causes of performance anomalies.

Furthermore, we summarize the contributions in this chapter as follows:

- We propose a method named MicroRCA+, to locate the bottleneck metrics for a faulty service using deep learning, without requiring historical failure data and application instrumentation. The evaluation shows that this method can identify the culprit metric with 85.5% in precision.
- We evaluate the overall performance of CI techniques on diagnosing performance anomalies in cloud microservices through applying six representative CI techniques to locate root causes of a range of performance anomalies injected into two well-known microservices benchmarks.
- Based on the experimental findings of CI evaluation, we propose a fine-grained causal localization method named MicroDiag, using Spatio-temporal causal inference. The evaluation shows that MicroDiag can rank 97% of the culprit metrics in one of the top 3 most likely causes, outperforming at least 31.1% of several state-of-the-art methods.

6.1 *microrca+*: CULPRIT METRIC LOCALIZATION WITH DEEP LEARNING

This section presents MicroRCA+[1], which employs deep learning to identify the culprit metrics that explain why the performance anomaly occurs. This method is an extension of our MicroRCA, which identifies the culprit services. It applies a deep learning algorithm into the relevant metrics of the detected faulty service to infer the bottlenecks that cause the service performance degradation.

Figure 19 shows the overall structure of our method. Before triggering the culprit metric localization (CML) procedure, the performance anomaly is detected, and the possible faulty services are pinpointed in culprit service localization (CSL), using the metrics data collected from microservices and the underlying cloud infrastructure. For each service in the ranked list of faulty services, MicroRCA+ applies autoencoders to identify the most likely metrics that contribute to the service abnormality. In the end, a list of (service, metrics list) pairs is output as the root causes.

1 Parts of this section are published in [214].

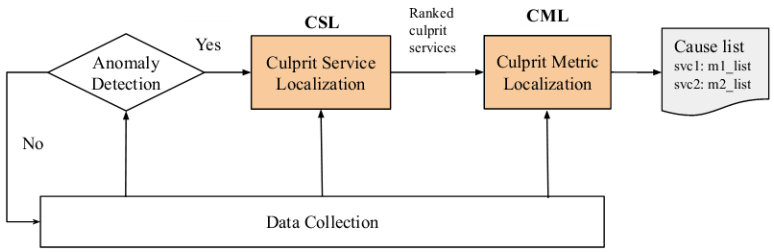


Figure 19: The workflow of MicroRCA+.

The localization of the culprit metric in MicroRCA+ assumes that the most crucial symptom of the faulty service has a significant deviation from its normal behavior. For example, the CPU exhaustion issue causes a significant increase in CPU utilization compared to normal operations. Thus, measuring the contribution of individual metrics to the abnormal symptoms that lead to the discrepancy between observed and normal values allows locating the most likely cause for the performance anomalies.

6.1.1 *Culprit metric localization with autoencoders*

In MicroRCA+, we adopt the autoencoder algorithm [51] to discover the culprit metric responsible for performance degradation of the faulty service. An autoencoder is beneficial to deal with an arbitrary number of metrics as the input so that it can include many potential symptoms at once. Moreover, it is able to correlate arbitrary relationships and high complexity within the observed metrics by tuning the depth and applied non-linearity of the neural networks to capture diverse anomaly patterns.

The overall process of culprit metric localization is depicted in Figure 20. It consists of three units: the autoencoder, anomaly detection, and root cause localization, with two phrases included: offline training and online (detection and) localization. The offline training phase is to learn the normal patterns of faulty services with metrics collected from normal operations. This training process can be performed following the services updates or routinely (e.g., once per day). During the training phases, the parameters of autoencoders are obtained with aiming to minimize the reconstruction loss. Next, the trained model is used to perform online detection and localization. For a metric m at time t, we can obtain a predicted value from the autoencoder and compute a reconstruction error that measures the difference between the observed and predicted values. If the error is over a threshold, it is declared anomalous, and the metric that contributes the most is identified as the culprit metric. In

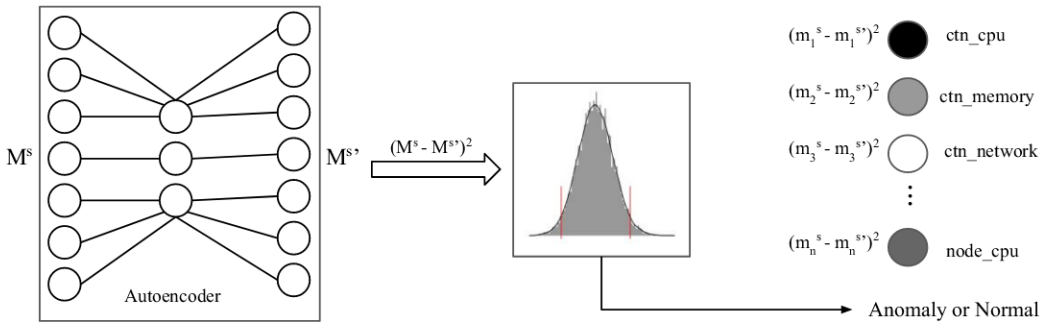


Figure 20: The inner diagram of culprit metric localization in MicroRCA+.

the end, an ordered list of the most likely causes is generated based on the reconstruction errors.

In MicroRCA+, anomaly detection is based on the assumption that the reconstruction error of an anomaly sample is expected to be large as the model is trained to reconstruct normal behaviors. The decision about anomaly is based on a threshold of the reconstruction error. Instead of setting a fixed error threshold, we employ a validation set (part of the training data) for threshold selection. For each window of time series points in the validation set, we apply the model trained by the training set and calculate the mean squared error (MSE) between the predicted value and the actual sample. At every time step, the MSEs of the validation group are modeled by a Gaussian distribution. We choose the Gaussian distribution to model the MSEs as per the central limit theorem that such samples follow the normal distribution. In the anomaly detection block, the error is considered normal if it is within a high confidence interval of the above Gaussian distribution; otherwise, it is considered abnormal. In addition, the L_1 regularization technique is used to penalize the autoencoder for enforcing sparse weights and preventing the propagation of information that is not relevant. This prevents the modeling of irrelevant dependencies between metrics.

After the anomaly is detected, the most likely symptom causing the service performance degradation is localized by computing the reconstruction error of an individual metric and then ranking them. In Figure 20, the most likely culprit metric is the ctn_cpu, which indicates the CPU utilization of a container causes the service performance degradation. Hence, we report the metric that contributes most to the reconstruction error as the culprit for a given performance anomaly.

In addition to identifying the culprit metric of the faulty service, we note that the autoencoder can also be used to calibrate the false positives in faulty service localization. MicroRCA+ is to localize culprit metrics for the abnormal-

ity of a service, which provides a local property of the performance anomaly. By combining the global search based on dependency graph (in coarse-grained cause localization) and the local diagnosis on metrics, the results of locating the faulty service can be improved.

6.1.2 *Evaluation*

We evaluate MicroRCA+ on a testbed as described in Chapter 5.3.1, and inject two types of anomalies, namely CPU hog and memory leak, into four services of sock-shop using the same method. For each anomaly, we repeated it six times with a duration of at least 3 minutes. In addition, we collect metrics for two hours when microservices operate normally with the same workload, which is used to train the autoencoders. The results are analyzed with evaluation metrics PR@k and MAP, where the root cause is the collected observable metrics instead of services.

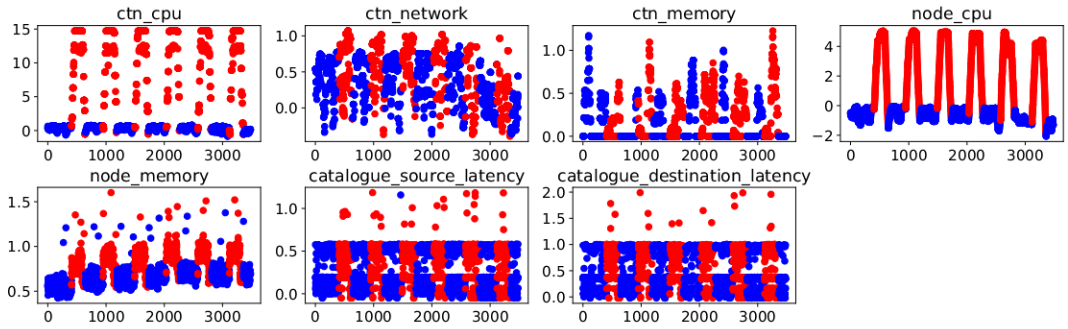


Figure 21: Metrics collected when a CPU hog is injected to the microservice catalogue.

For each anomaly case, we collect performance metrics from the application (suffixed with latency) and system, including containers (prefixed with ctn) and worker nodes (prefixed with node). Figure 21 gives an example of the collected metrics when a CPU hog fault is injected into the microservice catalogue, repeated six times within an hour. The data collected during the fault injection is marked in red. The CPU hog fault is expected to be reflected by the metric ctn_cpu. We can see that (1) there are obvious peaks in metrics ctn_cpu and node_cpu. The peak of ctn_cpu causes the peak of node_cpu as the container resource usage is correlated to node resource usage; (2) metrics ctn_memory and node_memory also have some deviations; (3) the CPU hog fault causes peaks in the service latency. Therefore, we can conclude that the fault injected into the service manifests a significant deviation from normal status. Meanwhile, it also affects some other metrics.

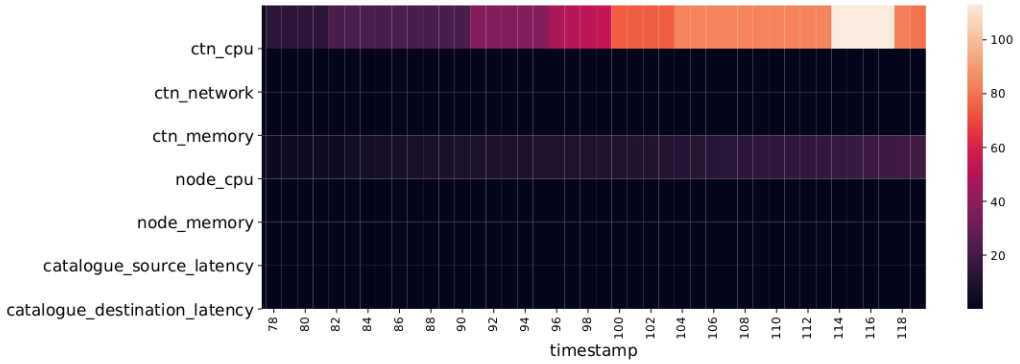


Figure 22: Reconstruction errors of metrics.

We train the autoencoder with normal data for each fault-injected service and test it with the anomalous data. Figure 22 shows the reconstruction errors of the autoencoder for each metric. We can see that the metric ctn_cpu has a large error compared to other metrics, indicating that it is more likely to be the cause of the anomaly of the microservice catalogue. The second highest reconstruction error is in the metric node_cpu, due to its strong correlation with the container resource usage. Hence, we conclude that ctn_cpu is the culprit metric.

Table 7 demonstrates the results of MicroRCA+ on different microservices and faults in terms of PR@1, PR@3, and MAP. We observe that MicroRCA+ performs well on various services and faults with 100 in PR@1, except for the service orders and carts with the fault memory leak. This is because (1) orders and carts are computation-intensive services; (2) we heavily exhaust their resource memory in the fault injection; (3) fault memory leak issues

Table 7: Performance of MicroRCA+ on identifying culprit metrics.

service	**orders**	**catalogue**	**carts**	**user**	**average**
CPU Hog					
PR@1	1.0	1.0	1.0	1.0	1.0
PR@3	1.0	1.0	1.0	1.0	1.0
MAP	1.0	1.0	1.0	1.0	1.0
Memory Leak					
PR@1	0.83	1.0	0	1.0	0.71
PR@3	0.83	1.0	1.0	1.0	0.96
MAP	0.88	1.0	0.83	1.0	0.93

manifest as both high memory usage and high CPU usage. As our method targets the root cause that manifests itself with a significant deviation of the causal metric, the precision decreases when the root cause manifests itself in multiple metrics. On average, our method achieves 85.5% in precision and 96.5% in MAP.

6.2 CAUSAL INFERENCE TECHNIQUES FOR PERFORMANCE DIAGNOSIS

One of the predominant approaches for diagnosing performance anomalies in cloud microservices is to formulate it as a causal inference problem. It derives the causal relations between components or metrics represented with a causality graph to capture the anomaly propagation, and then it identifies the potential root causes by ranking the nodes in the graph. For example, Microscope [113] applies the PC algorithm, and Loud [116] uses Granger causality to obtain the causal graphs. Although many state-of-the-art approaches employ CI techniques to locate the root causes of performance anomalies, the overall performance of this class of methods for root cause analysis is not well understood. To this end, this section presents the results of a comprehensive evaluation of six representative CI methods from three categories, including Granger causality, causal network learning algorithms, and structural equation models (SCMs), for locating root causes at both coarse and fine granularity for cloud microservices[2] (the background of CI techniques is introduced in Chapter 2.3.3). The root cause localization is based on observable metrics from a set of fault injection experiments, where three types of anomalies are injected into heterogeneous services of two widely used microservice benchmarks. In addition, we developed a non-CI method that leverages the domain knowledge of service dependencies for comparison. To the best of our knowledge, this work is the first such comprehensive study to be reported on evaluating CI techniques for performance diagnosis in cloud microservices. The experimental results presented in this work are useful for understanding how well CI methods perform on root cause localization in order to make an appropriate decision when applying CI methods for performance diagnosis.

6.2.1 *Experimental design*

To understand the performance of CI techniques on locating root causes in cloud microservices, we first formulate three research questions, which demonstrate several dimensions of the challenges (i.e., numerous metrics and hetero-

2 Parts of this section are published in [215].

geneous anomaly patterns) in microservice performance diagnosis. To cover the challenge of a wide range of root causes, we conduct a set of experiments by injecting three types of anomalies into multiple different services. We then evaluate the performance of CI methods according to the three research questions with the metrics data collected from microservice benchmarks. Lastly, we study the computation overhead and scalability of selected CI methods. The details of our experiment design are shown in Table 8. The three research questions are formulated as follows:

RQ1: *How do CI techniques perform on pinpointing the faulty service that initiates the performance anomaly* (coarse-grained diagnosis)?

To localize the faulty service that initiates the performance anomaly, we apply CI methods to the response times of all services to construct the anomaly propagation among services. The number of metrics is equal to the number of services, and the heterogeneity of anomaly patterns in response times is relatively low. To understand the performance of CI methods against different numbers of metrics in a low-level heterogeneity, we compare the performance of CI methods under three types of feature reduction, including no feature reduction, feature reduction, and ideal feature reduction. Moreover, we compare CI methods to a non-CI method that uses the domain knowledge of service dependency graph.

RQ2: *How do CI techniques perform on locating the culprit metric that contributes to the faulty service abnormality, given the faulty service* (fine-grained diagnosis with giving the coarse-grained cause)?

We assume the faulty service is known, which can be satisfied with our previous work [209] and other faulty service localization methods [111, 113], then apply CI methods on metrics exposed by the faulty service to identify the culprit metric that leads to the service's abnormality. The number of metrics is relatively low, but their heterogeneity increases as different types of metrics are likely to have different anomaly patterns.

RQ3: *How do CI techniques perform on locating the culprit metric from all available metrics* (fine-grained diagnosis)?

We use all observable metrics to pinpoint the culprit metric that initiates the anomaly. The number of metrics is n times larger than the number of metrics in *RQ2*, where n is the number of services. Meanwhile, the heterogeneity of metrics is at a high level as diverse anomaly patterns exist in different types of metrics and heterogeneous services. Given the high heterogeneity, we also compare the performance of CI methods against feature reduction.

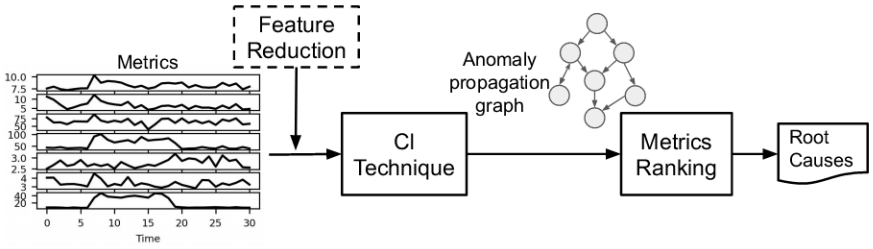


Figure 23: Performance diagnosis framework with CI techniques.

6.2.2 *Performance diagnosis framework*

Figure 23 shows the framework for performance diagnosis with CI techniques. The input of the framework is metrics corresponding to the research questions (metrics for research questions are listed in Table 8) with an optional feature reduction. Next, each CI method is employed to construct an anomaly propagation graph. Finally, it ranks the nodes in the graph and outputs a ranked list of potential root causes, where the top items have the highest probability to be the root causes. Based on the input metrics, the output root causes can be coarse-grained and fine-grained.

In our evaluation, we implement feature reduction with an unsupervised machine learning algorithm called Birch Clustering [211] for identifying abnormal metrics, and utilize the PageRank graph centrality algorithm, which is commonly used in the state-of-the-art performance diagnosis approaches, to rank the root causes. Notably, we evaluate CI techniques with a primary ranking method as our focus is on comparing the CI techniques. For the actual application of such techniques, the performance can be improved with advanced ranking methods [129, 209].

Table 8: Research questions, metrics, benchmarks, feature reduction, non-CI comparison, and results.

Research question	Metrics	Benchmark	Feature reduction	Non-CI comparison	Results
RQ1	response times of all services	sock-shop	N	Y	Chapter 6.2.5.2
			Y		
			Ideal		
RQ2	metrics of the faulty service	sock-shop	N	N	Chapter 6.2.5.3
RQ3	metrics of all services	sock-shop	N	N	Chapter 6.2.5.4
			Y		
Overhead	metrics of RQ3	sock-shop	Y	N	Chapter 6.2.6.1
Scalability	metrics of RQ1	train-ticket	Y	Y	Chapter 6.2.6.2
	metrics of RQ3			N	

6.2.3 *Causal inference techniques*

This work aims to understand the performance of the class of CI techniques on microservices performance diagnosis. To achieve this goal, we select six representative CI methods from three categories, namely Granger causality, causal network learning algorithms, and structural equation models (SCMs), based on their popularity, assumptions, and availability of implementations. Notably, we use the pure form of these methods rather than the tailed ones used in existing approaches [111, 129] as we aim to understand the performance of different types of CI methods in diagnosing microservice performance in general rather than achieving high diagnostic accuracy. The chosen CI methods are as follows:

- GC: GC [58] is a popular CI method that has been adopted by many performance diagnosis methods [115, 116] to build the causal graph. We use χ^2 as the Granger causality test.
- PC algorithm with partial correlation (PC-corr): PC-corr is a version based on PC algorithm [216] that uses Fisher's z-transformation of the partial correlation to test the (conditional) independence.
- PC algorithm with kernel (PC-kernel): PC-kernel is a version of PC algorithm that uses kernel-based independence criteria [217] which can deal with non-linear causal-effect relationships and non-Gaussian noise. We use the Hilbert-Schmidt independence criterion to test independence.
- Greedy Equivalence Search (GES): GES [60] is a score-based CI method. We use "obs" which is an ℓ_0-penalized Gaussian maximum likelihood estimator based on BIC score in the implementation.
- Causal Additive models (CAM): CAM [218] is an SCM method that identifies causal relations by fitting an additive SEM with Gaussian error, where the causal-effect relationships can be non-linear.
- LiNGAM: LiNGAM [64] is an SCM method that assumes linear causal relationships but with non-Gaussian disturbance.

We conclude the chosen CI methods and their hyperparameters in Table 9. In addition, we implement and compare to a non-CI method named *Corr*, which uses the service dependency graph directly and incorporates the anomaly symptoms by setting weights with correlation coefficients.

Table 9: The chosen causal inference methods and their hyperparameters.

Method	Category	Assumptions	Hyperparameters	References
GC	Time-lag	Linearity	AIC criteria for lag selection; maxlag = 30s; χ^2 test for granger causality; p-value = 0.05.	package statsmodels [219]
PC-corr	Constraint-based	Gaussianity; Linearity	CItest = gaussian, alpha=0.01	Causal discovery toolbox [220]
PC-kernel	Constraint-based	Non-Gaussianity; Non-linearity	CItest = hsic_gamma, alpha=0.01	Causal discovery toolbox [220]
GES	Score-based	Gaussian; Linearity	score=obs	Causal discovery toolbox [220]
CAM	SCM	Gaussianity; Non-linearity	score=nonlinear, cutoff=0.001, selmethod=gamboost, pruning=False, prunmethod=gam	Causal discovery toolbox
LiNGAM	SCM	Non-Gaussianity; Linearity	nonparametric; max_iter=1000	ICALiNGAM in python package lingam [221]

6.2.4 *Benchmarks and evaluation metrics*

The evaluation is based on metrics data collected from fault injection experiments on microservice benchmarks.

6.2.4.1 *Benchmarks*

We select two representative microservice benchmarks named sock-shop[3] and train-ticket[3] (one of the largest microservice benchmarks), which simulate an e-commerce website for selling socks and a train ticket booking system, respectively. Sock-shop consists of thirteen microservices, and train-ticket has forty-one microservices. They are polyglot (e.g., Java, Golang, Node.js, etc.) and intercommunicate using REST over HTTP. Compared to sock-shop, train-ticket has longer propagation paths and exposes a larger number of metrics. Figure 24 shows the key services in train-ticket and the potential anomaly propagation paths of the *station* microservice. Anomalies from this service probably affect nine microservices, and the longest propagation path includes six microservices, which would result in a large number of anomalous metrics.

3 Train-ticket - https://github.com/FudanSELab/train-ticket

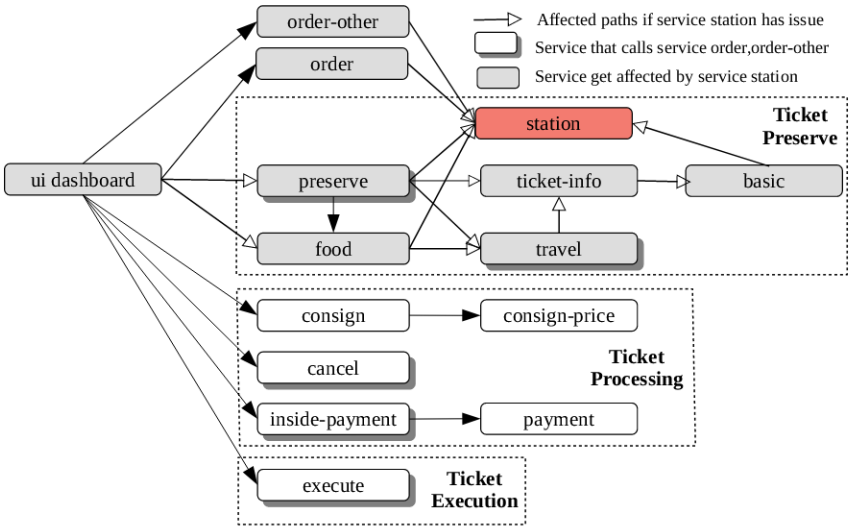


Figure 24: The key microservices in benchmark train-ticket.

To these two microservices benchmarks, three types of faults described in Chapter 5.3.1.5 are injected into multiple services. Each anomaly lasts 1 minute, and the system has 5 minutes to cool down before another injection. We repeat the injection process 3-5 times for each fault and service to increase generality.

6.2.4.2 *Evaluation metrics*

We quantify the performance of CI methods on diagnosing the root cause with two metrics: precision at top k (PR@k) and average precision at top k (AP@k). These two evaluation metrics are defined as follows:

- *Precision at top k* is same as Equation 12, where the root causes v_{rc} are services in coarse-grained diagnosis, and metrics in fine-grained diagnosis.
- *Average Precision at k* (AP@k) quantifies the overall performance of an algorithm with an average of PR@k:

$$AP@k = \frac{1}{k} \sum_{1 \leqslant j \leqslant k} PR@j. \tag{14}$$

6.2.5 *Experiment results*

This section presents the fault injection experiments setup, the results of the three research questions, and the discussion about the computation overhead and scalability of the selected CI techniques.

6.2.5.1 *Testbed*

We create a testbed using Kubernetes where we deploy a microservice benchmark and a set of monitoring tools, as shown in Figure 25. In our Kubernetes cluster, there are one master node and multiple worker nodes (4 for sock-shop and 6 for train-ticket), including one worker node dedicated for data collection and the rest for microservices. The hardware and software configuration is the same as Table 1, and we use the same load generator and data collection described in Chapter 5.3.

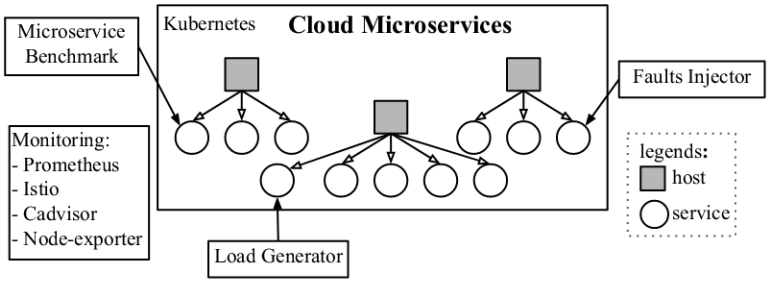
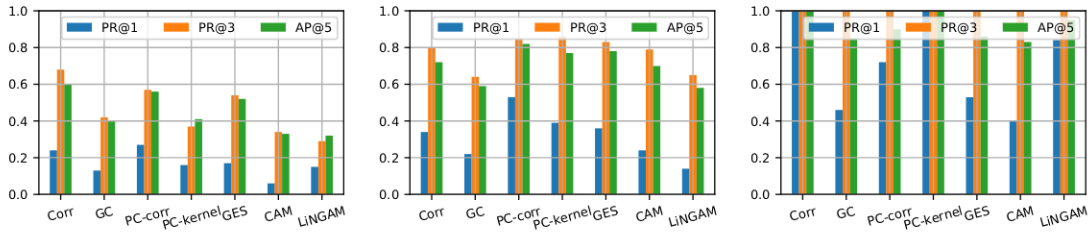


Figure 25: An overview of our experimental testbed.

6.2.5.2 *RQ1: How do CI techniques perform on coarse-grained diagnosis?*

We evaluate the performance of CI techniques on locating the faulty service that initiates the performance anomalies based on the response times of all



(a) No feature reduction (b) Feature reduction (c) Ideal feature reduction

Figure 26: The results of PR@1, PR@3, and AP@5 for coarse-grained diagnosis against feature selections.

services. We first take all the metrics without any feature reduction as the input and obtain the diagnosis result in terms of PR@1, PR@3, and AP@5 shown in Figure 26(a). Overall, all the methods cannot identify the faulty services well (with a maximum of 27% in PR@1). This is because services are interdependent not only in abnormal status but also in normal status. These normal dependencies in metrics introduce extra causal links in the graph, which renders the graph centrality algorithm unable to get the precise root cause.

Next, we apply a feature reduction which results in an F1-score of 0.6 in anomaly detection. The diagnosis result is shown in Figure 26(b) shows that the CI technique with feature reduction improves on average 77.4% in PR@3, outperforming this scenario with no feature reduction by 31.6%. In addition, we can see that constraint-based methods achieve a higher performance than non-CI method *Corr*, like *PC-corr* achieves an improvement of 19% in terms of PR@1 over *Corr*. This is because feature reduction with an F1-score of 0.6 includes false positives, resulting in spurious propagation paths in *Corr*. In contrast, CI methods can eliminate parts of the spurious paths, thus increasing the accuracy.

Finally, we evaluate the performance of CI methods with an ideal feature reduction, where only the expected anomalies, including the faulty service and its upstream services, are used to construct the anomaly propagation graph. The performance of all methods is shown in Figure 26(c). Overall, all the methods achieve higher performance when only true-positives are detected as anomalies, on average, achieving 70.7% in PR@1, improving 39% over feature selection with an F1-score of 0.6. Particularly, both *Corr* and *PC-kernel* achieves 100% precision.

We note that the results reported here can be improved if advanced methods are applied according to the state-of-the-art methods, like using metrics from multiple layers to score the abnormality of services and assigning attributes to edges and nodes before ranking that has been done in our previous work MicroRCA [209].

Summary: Neither CI methods nor the non-CI method based on service dependency graphs can identify the faulty services well without feature reduction, as causal relations exist among both normal and abnormal metrics. However, their performance can be improved if feature selection can be applied before causal inference. In particular, *PC-kernel* and the non-CI method *Corr* achieve 100% in precision when only true-positives are detected as anomalies. More importantly, constraint-based CI methods, like *PC-corr*, are more robust than the knowledge-based non-CI method to false positives.

Table 10: The performance of CI methods on fine-grained diagnosis with giving faulty service against different types of anomaly.

Methods	GC	PC-corr	PC-kernel	GES	CAM	LiNGAM	average
Latency							
PR@1	**0.24**	0.15	0.15	0.21	0.09	0.21	0.175
PR@3	**0.56**	0.41	0.44	0.41	0.38	0.41	0.435
AP@5	**0.55**	0.45	0.49	0.44	0.36	0.44	0.455
CPU Hog							
PR@1	0.23	**0.63**	**0.73**	0.17	0.2	0.17	0.355
PR@3	0.70	**0.80**	**0.83**	0.73	0.5	0.73	0.715
AP@5	0.66	**0.85**	**0.87**	0.63	0.45	0.63	0.68
Memory Leak							
PR@1	**0.32**	0.11	0.11	0.18	0.21	0.18	0.19
PR@3	0.57	0.57	0.50	**0.61**	0.39	**0.61**	0.54
AP@5	**0.56**	0.51	0.49	**0.54**	0.46	**0.54**	0.517

6.2.5.3 *RQ2: How do CI techniques perform on fine-grained diagnosis with giving the faulty service?*

To address this research question, we apply CI methods to the faulty service's relevant metrics and evaluate their performance on locating the culprit metrics contributing to the service abnormality. Table 10 shows the performance of each CI method on locating the culprit metric on different types of anomalies in terms of PR@1, PR@3, and AP@5 when no feature reduction is applied. We can see that Granger causality (GC) performs better than other methods on latency issues. This is because latency anomalies propagate to other resources through service performance degradation, which can be reflected as time-lags among metrics, which satisfies the assumption of GC.

Regarding CPU hog issues, most of the CI methods identify the root cause with a higher performance except the CAM method. This is because CPU issues reflect themselves as strong correlations between cause and effect metrics and weak correlations with non-causal metrics. Figure 27 shows the maximum and minimum coefficient of determination r^2 of linear regression (r_max, r_min) and Pearson correlation coefficients (p_max, p_min) between the culprit metric and other metrics. We can see that there is a significant difference between the maximum and minimum values (95.7% of p_max and 12%

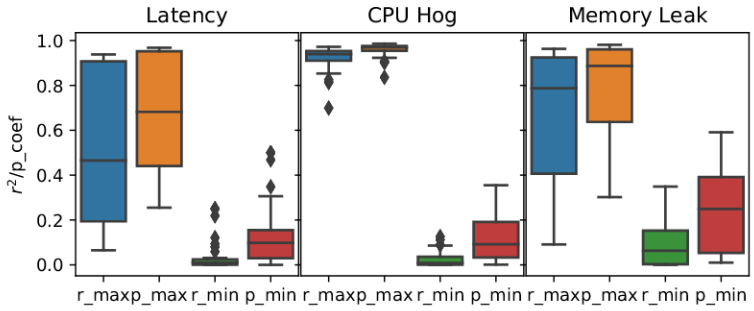


Figure 27: R-squared and Pearson correlation coefficient between culprit metric and other metrics of different types of anomaly.

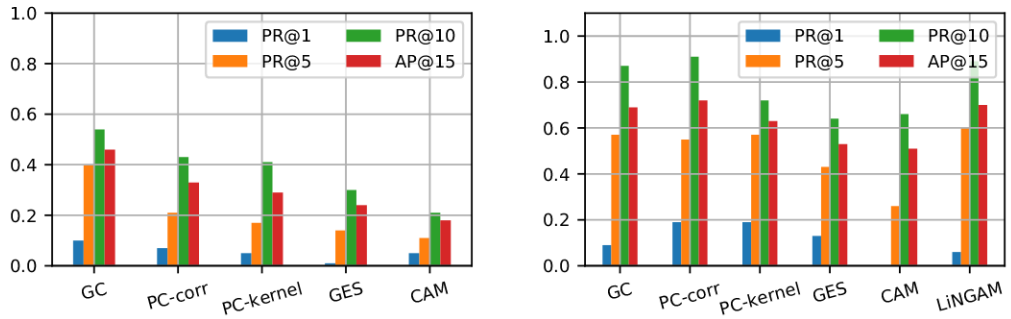
of p_min). Due to these obvious differences between causal and non-causal metrics, it is easier to diagnose the root cause. Meanwhile, due to this linearity between cause and effect metrics in CPU hog issues, the CAM method performs poorly for this type of anomalies as it models non-linear causal relations. Conversely, we can see that latency and memory leak issues yield smaller differences between causal and non-causal metrics, resulting in lower performance for those methods that assume linearity.

For memory leak issues, we can see that LiNGAM and GES perform better than other methods in terms of PR@3. This is because memory leak issues manifest themselves in multiple resource metrics (e.g., memory utilization and CPU utilization), which introduces contemporaneous effects that can be difficult to identify for other CI methods. LiNGAM is a model that can capture such effects; thus, it performs better for memory leak issues.

Summary: Anomaly symptoms in service relevant metrics vary with the type of anomaly, which violates or satisfies the assumptions of a specific CI method. To exemplify, Granger causality performs well on latency issues, SCM method LiNGAM performs well on instantaneous effects in memory leak issues, and most of the CI methods, particularly the PC-algorithm, perform well for correlated anomaly symptoms caused by CPU hogging. On average, the PC algorithm has a better performance on the three types of anomalies.

6.2.5.4 *RQ3: How do CI techniques perform on fine-grained diagnosis with all observational metrics?*

We address this research question by evaluating the performance of CI on identifying the culprit metrics from all observed metrics. Figure 28(a) shows the performance of culprit metric diagnosis of each CI method in terms of PR@1, PR@5, PR@10, and AP@15, where no feature reduction is applied. In



(a) No feature reduction

(b) Feature reduction

Figure 28: The results of PR@1, PR@5, PR@10, and AP@15 for fine-grained diagnosis against feature reductions.

this experiment, we use a larger $k = \{1, 5, 10, 15\}$ in the evaluation metrics compared to RQ1 and RQ2 ($k = \{1, 3, 5\}$). This is because the number of potential root causes is much larger, and a small k cannot differentiate the performance of CI methods well. We note that LiNGAM is not included when the number of metrics exceeds the sample size, for which the method is not defined.

We can see that most of the methods cannot locate the culprit metric well, with an average of 5.6% in PR@1, and 20.6% in PR@5. This is because (1) higher-dimensional features exaggerate the difficulty to infer a precise anomaly propagation graph; (2) anomalies propagate across services and also their metrics, resulting in many instantaneous effects that are difficult to discover; (3) due to the heterogeneity of services, their anomalous metrics commonly manifest diverse symptoms. It is hard to capture various causal relations with one CI method; (4) normal patterns of metrics also introduce unexpected inference.

Next, we evaluate the performance of CI methods with feature reduction. Figure 28(b) shows the performance of CI methods on reduced metrics. Overall, all the CI methods improve after the feature reduction with an average of 5.4% in PR@1 and 29.1% in PR@5, more than no feature selection. Particularly, GC, PC-corr, PC-kernel, and LiNGAM achieve an average of 57% in terms of PR@5. With the feature reduction, some unexpected and uncorrelated patterns can be removed from metrics, thus reducing the inference to causal graph learning. Additionally, we notice that LiNGAM achieves the highest rank in PR@5 with 60%, which indicates that LiNGAM has a higher performance in identifying fine-grained causes in cloud microservices.

Lastly, we compare the performance of CI methods on different groups of metrics corresponding to our three research questions. Table 11 shows the performance in terms of PR@3 of each CI method. We can see that the perfor-

Table 11: The performance of CI methods on different ranges of metrics in terms of PR@3.

RQ	Metrics	GC	PC-corr	PC-kernel	GES	CAM	LiNGAM	Average
RQ1	response times of all services	0.47	**0.62**	0.47	0.25	0.41	0.25	0.45
RQ2	metrics of culprit service	**0.61**	0.59	**0.59**	**0.58**	**0.42**	**0.58**	**0.56**
RQ3	metrics of all services with filtering	0.45	0.4	0.43	0.23	0.15	0.26	0.32

mance of CI methods on dealing with metrics of all services (RQ3) is lower than others, as metrics in RQ3 include diverse anomaly patterns not only from service-relevant metrics but also from heterogeneous services. In comparison, most CI methods achieve a higher performance in identifying the culprit metric from metrics exposed by faulty service.

Summary: It is difficult to identify fine-grained causes from all metrics, which is a mixture of many normal and abnormal metrics (20.6% in PR@5). The performance of CI methods can be improved with the feature reduction, but it does not rank the root cause in the top 5 of the list well (49.7% in PR@5). Instead, a drill-down approach that first identifies the faulty service pinpoints the culprit metric that attributes to the faulty service is more promising to obtain the fine-grained causes.

6.2.6 *Computation overhead and scalability*

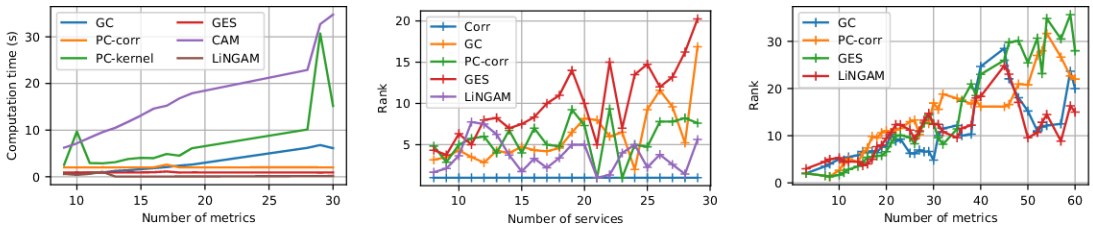
We discuss the overhead of CI methods and their scalability to the number of microservices and metrics.

6.2.6.1 *Overhead*

We analyze the computation overhead of each CI method, varying with the number of metrics it handles. Figure 29(a) shows the computation time of the six evaluated CI methods (running on the same Intel Core i7 version server with 8-core CPU and 16 GB memory) against the number of metrics (from 10 to 30, and with 5 minutes of data gathered from each metric, in a total of 60 data points). We can see that the computation times of CAM and PC-kernel increase dramatically with the number of features, and GC also becomes slightly slower when the number of metrics increases.

6.2.6.2 *Scalability*

We evaluate the scalability of CI methods on one of the largest benchmarks, named train-ticket, which has longer propagation paths and a larger number of metrics.



(a) Computation overhead (b) Coarse-grained diagnosis (c) Fine-grained diagnosis

Figure 29: Computation overhead, the rank of root causes against the number of metrics or services.

We conduct the same fault injection experiments on the service *station* in the benchmark train-ticket, which has the longest propagation paths. Then we apply four computation efficient CI methods, including GC, PC-corr, GES, and LiNGAM, on metrics corresponding to RQ1 and RQ3.

We evaluate each CI method's performance by measuring the root cause's rank against different numbers of metrics, which are produced by changing the threshold of the feature reduction. Figure 29(b) and (c) show the average rank of root cause (lower is better) against a different number of service-level metrics and metrics of all services. We can see from Figure 29(b) that the performance of *Corr* with the service dependency graph by far outperforms all CI methods. This is because the service *station* has a high in-degree (upstream services), which results in high ranking accuracy in *Corr* 's graph centrality algorithm. Meanwhile, we can see that LiNGAM identifies the culprit service in the top 5 as the number of metrics increases. However, from Figure 29(c), we can see that all CI methods have increasing ranks of root causes with the increasing number of metrics, which is due to the same issue of diverse anomaly symptoms mentioned in Chapter 6.2.5.4.

Summary: The computation overhead of some CI methods, such as PC-kernel and GES, increases dramatically with the number of metrics. When metrics have similar patterns, like service-level metrics, LiNGAM can identify the faulty service well even if the number of microservices is high. However, identifying the root causes from all metrics in the system is difficult due to the high-level heterogeneity.

6.2.7 *Conclusions*

The above evaluation based on a set of fault injection experiments shows that:

- CI methods need to incorporate anomaly symptoms through anomaly detection or anomaly scores to achieve high performance. In addition, some CI methods are more robust to false positives in anomaly detection than the domain knowledge-based non-CI method.
- The performance of CI techniques varies with the types of anomalies.
- The performance of CI techniques is sensitive to the heterogeneous anomaly patterns manifested in the metrics. Therefore, to identify fine-grained root causes, a drill-down approach that identifies the coarse-grained causes first then looks into the detailed information from its relevant metrics is better than inferring from all metrics.
- With the increasing number of services or metrics, the runtime of some CI techniques tends to increase drastically, and the precision of fine-grained diagnosis decreases. However, some CI methods can still perform well in identifying the faulty service that initiates the performance anomaly.

6.3 *microdiag*: CULPRIT METRICS LOCALIZATION WITH CAUSAL INFERENCE

Based on our findings from the above experimental study, we propose a fine-grained cause localization method, named MicroDiag[4]. This method applies Spatio-temporal causal analysis towards the metrics to capture the anomaly propagation in the system represented as a metric causality graph, and further localizes the source of the propagation from the inferred causality graph as the root cause.

6.3.1 *Overview of MicroDiag*

Figure 30 shows the procedures of fine-grained cause localization with MicroDiag. The cause localization is based on metrics M^c exposed by components C from multiple layers, including services, containers, and server nodes. Once metrics are collected, the unvarying metrics are removed, and the rest are grouped by components. Meanwhile, the anomaly detection module continuously detects the unexpected service behaviors using Birch Clustering (introduced in Chapter 5.1.3).

After the service performance degradation is detected, MicroDiag initiates the cause localization by inferring the spatial propagation among components

4 Parts of this section are published in [222].

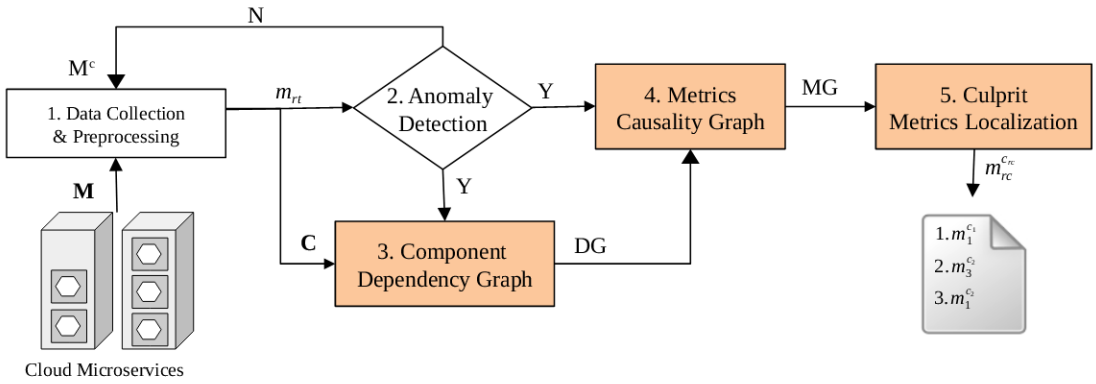


Figure 30: The workflow of fine-grained cause localization with MicroDiag.

as a component dependency graph DG, and then applies temporal causal inference on metrics of interactive components to capture the anomaly propagation among metrics, represented as a metric causality graph MG. Lastly, MicroDiag locates the source of the propagation represented by the metric causality graph, using a graph centrality algorithm, and outputs a ranked list of metrics, indicating both the faulty service and the type of metric. The top of the list is the most likely fine-grained causes.

6.3.2 *Culprit metric localization with causal inference*

The core idea of MicroDiag is to infer the anomaly propagation among metrics for locating the culprits. We represent the anomaly propagation among metrics with a metric causality graph, where each node represents an individual metric m_i^c of a component c and an edge represents a cause-effect relationship that is also an anomaly propagation path. To obtain the metric causality graph, we first analyze the spatial propagation across components, which eliminates the interference of similar anomalous patterns in metrics of different components, showing as a component dependency graph DG. Next, for each service in the DG, we apply temporal causal inference across layers adapting to their causal properties. In the end, a metric causality graph is constructed for culprit metric localization.

6.3.2.1 *Component dependency graph*

In the first step, MicroDiag infers the potential spatial propagation across components, including services, containers, and server nodes, which is represented with a component dependency graph.

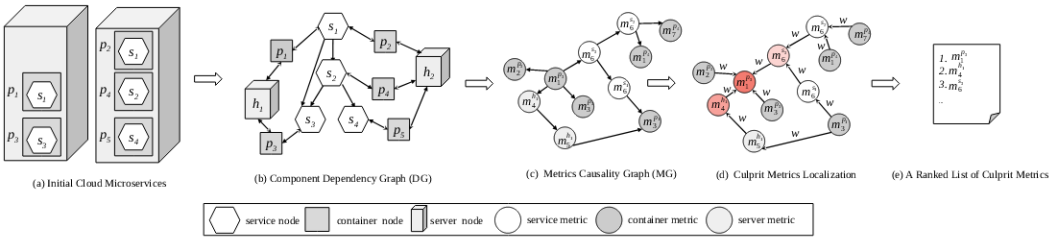


Figure 31: Culprit metric localization procedures in MicroDiag.

In cloud microservices, (e.g., Figure 31(a)), an anomaly can propagate across not only interdependent services along invocations but also containers and severs through resources sharing or contention (such as s_1 and s_3 in Figure 31(a)). Particularly, due to the isolated nature of containerization, the anomaly propagation between service and server nodes is across containers, and the fault in the container propagates to both service and sever nodes. Therefore, we construct a component dependency graph to show potential spatial propagations among services, containers, and servers.

The component dependency graph is an extension of the attributed graph in MicroRCA [209], where a new node type of container is added to service and host nodes. Similar to MicroRCA, for each service in a microservice-based application, we add edges to its communicating services and containers it runs inside; for a container running a service, we add edge to the service it containerized and the server node on which it runs. For a server node, we add edges to all the containers it runs. Thus, we get a directed graph where edges across service, container, and server layers are bi-directed.

We construct the component dependency graph by parsing the metrics collected by the cloud-native monitoring stack as MicroRCA. Specifically, we parse the service-level metrics collected by the service mesh istio[3] to obtain the service dependencies and the system metrics collected by Cadvisor[5] to get the deployment information. Figure 31(b) gives an example of the component dependency graph of the cloud microservice in Figure 31(a), including service dependencies and deployment relationships among services, containers, and servers. The spatial propagations shown in the component dependency graph vastly reduce the spurious causal relations between unrelated metrics caused by diverse anomaly patterns of metrics.

6.3.2.2 *Metric causality graph*

With the spatial propagation among components, MicroDiag next applies temporal causal inference to metrics of interdependent components, in order to obtain a metric causality graph showing the anomaly propagation among met-

rics. In the metric causality graph, each node represents an individual metric m_i^c of a component c and an edge represents a cause-effect relationship that is also an anomaly propagation path. For example, an edge from metric $m_1^{p_3}$ to metric $m_2^{p_3}$ in Figure 31(c), means that $m_1^{p_3}$ causes $m_2^{p_3}$ and the anomaly propagates from $m_1^{p_3}$ to $m_2^{p_3}$.

MicroDiag identifies the causal relations among metrics by differentiating the causal properties of metrics between different layers using different causal inference techniques. For each service in the component dependency graph, we construct a small causality graph with metrics of components in a path from service to the server that spans three layers (i.e., service, container, and host). We note that the temporal causal inference is applied to anomalous services. As the number of services in microservice-based applications is large, analyzing the anomaly propagation among anomalous services can reduce not only the computation overhead but also the interference of unrelated metrics. In our example in Figure 31, the anomalous services are s_1, s_2, and s_3.

In the temporal causal inference, three types of causal properties are considered: (1) propagation across services; (2) propagation across resource and service-related metrics of linked service and container nodes; (3) propagation across resource metrics of linked container and server nodes.

Regarding (1) propagation across services, as the anomaly propagates along with the service invocations, specifically, an anomaly in a downstream service would affect its upstream services. For example, the increased response time of service s_3 would increase the tail or medium latencies of service s_1 and s_2. Therefore, MicroDiag reverses the edges between services in the component dependency graph to represent the anomaly propagation paths among services. As shown in Figure 31(c), the edges between services s_1, s_2, and s_3 are opposite to the edges in Figure 31(b).

Regarding (2) propagation across metrics of linked container and service nodes, MicroDiag leverages a time-lag-based causal inference method named Granger causality to capture the anomaly propagation. This is based on the assumption that a causal metric affects other metrics in an accumulative manner, where a cause precedes an effect. Granger causality is helpful to determine whether a time series can be used to predict another time series.

Selecting the time lag is a critical problem in GC. The estimation of AutoRegression models requires the model order (time-lag) included. Too few lags can lead to poor representation of the data, whereas too many of them can lead to problems in the model estimation. MicroDiag adopts the Akaike Information Criterion (AIC) to estimate the model order. AIC is calculated for a set of model orders, and the order that yields the lowest value of AIC is selected as the model order of the AR model to determine Granger causality

between two time series. Once the time lag is determined, MicroDiag infers the causal relations by applying GC with the χ^2 test. As GC tests might introduce spurious causal relations among container resource metrics, we calibrate the causality with the results from the SCM model.

Regarding (3) anomaly propagation across resource metrics, MicroDiag employs a structural causal model (SCM) [63] to infer the causal relationships among the resource metrics. An anomaly from some resource metrics propagates to other resource metrics simultaneously. For example, a memory leak issue in a container manifests itself in both memory and CPU resource metrics and also in the corresponding resource metrics of the server node. This kind of propagation is known as *contemporary effects*, which are hard to identify through conditional independence tests, like the PC algorithm and time-lag-based methods used in the existing methods [111, 116]. Instead, we address the identification of the contemporary effects with a structural causal model (SCM). Considering that server resources are the sum of the container resources it runs, i.e., a linear function, and anomalous resource metrics follow a non-gaussian distribution [223], we employ the LiNGAM to infer the anomaly propagation paths across resource metrics independent containers and servers.

LiNGAM estimates a causal ordering of variables with no prior knowledge of the structure and assumes the observed data is generated from a process represented graphically by a directed acyclic graph (DAG). Let us represent this DAG with a $p \times p$ adjacency matrix $\mathbf{B} = \{b_{ij}\}$ where every b_{ij} represents the connection strength from a metric m_i to another m_j in the DAG. Instead of getting the causal order of the metrics, we use the adjacency matrix to construct the anomaly propagation graph. Therefore, for a metric $m_i^{c_x}$ of component c_x, it is defined as a linear function of the causal metrics $m_j^{c_y}$ from which the anomaly propagates, and an independent factor $e_i^{c_x}$, where the latter from the measurement errors or fluctuations of the metrics. Formally, $m_i^{c_x}$ can be defined as Equation 15.

$$m_i^{c_x} = \sum_{k(j)<k(i)} b_{ij} m_j^{c_y} + e_i^{c_x} \tag{15}$$

Furthermore, we rewrite the model in a matrix form as follows:

$$\mathbf{m} = \mathbf{B}\mathbf{m} + \boldsymbol{e}, \quad \mathbf{m} = \boldsymbol{A}\boldsymbol{e} \tag{16}$$

where $\mathbf{m}$ is the n-dimensional random vector (metric time series), and $\mathbf{B}$ is the direct causal effect and $\boldsymbol{A}$ is the total causal effect. For each b_{ij}, it represents the direct causal effect of m_j on m_i and each a_{ij}, the (i, j)-th element of the matrix $\boldsymbol{A} = (\mathbf{I} - \mathbf{B})^{-1}$, represents the total effect of m_j on m_i. The goal of

identifying the causal relations is to estimate the adjacency matrix $\mathbf{B}$ only by observing the data $\mathfrak{m}$.

Algorithm 2: DirectLiNGAM.

Input: p-dimensional random vector m, a set of its variable subscripts U and a $p \times n$ data matrix of the random vector as M

Output: Matrix $\mathbf{B}$

Initialize an ordered list of variable $L := \emptyset$ and $k := 1$

for *Repeat until* $p - 1$ *subscripts are appended to* L **do**

  Perform least square regression of m_i on m_j for all $i \in U \setminus L$ ($i \neq j$) and compute the residual vectors $r(j)$ and the residual data matrix $R(j)$ from the data matrix M fro all $j \in U \setminus L$. Find a variable m_k that is most independent of its residuals: $m_k = \mathrm{argmin}_{j \in U \setminus L} T_{pwling}(m_j; U \setminus L)$,

  where T_{pwling} is the independence measure defined in Equation 18.

  Append m to the end of L

  Update $m := r^k$, $M := R^k$

end

Append the remaining variable to the end of L.

Construct matrix B by following the order in L, and estimate the connection strengths b_{ij} by using least squares regression on the original random vector m and the original data matrix M.

return B

MicroDiag employs a method named DirectLiNGAM [224] to infer the causality from the LiNGAM model. DirectLiNGAM has been proven to give promising results, especially when the number of observed data points is small compared to the dimension of the data, and it also shows algorithmic advantages because of no requirement of gradient-based iterative methods.

DirectLiNGAM estimates a causal order of variables by successively subtracting the effect of each independent component from given data. It first identifies an exogenous variable based on the independent measures of pairwise variables. For each pair of m_i and m_j, it applies least square regression and gets the residual r_i and r_j, and then it applies a differential mutual information to measure the dependency between these two variables.

$$\mathrm{DMI} = I(m_i, r_i) - I(m_j, r_j) = H(m_i) + H(\frac{r_i}{\sigma_{r_i}}) - H(m_j) - H(\frac{r_j}{\sigma_{r_j}}) \tag{17}$$

Where H is the differential entropy which is computed with maximum entropy approximation. This criterion is equivalent to evaluating the independence of m_i vs r_i and m_j vs r_j using mutual information and choosing the direction in which the regressor is more independent of the residual.

We first evaluate pairwise independence between a variable and each of the residuals and next take the sum of the pairwise measures over the residuals. Let us denote by U the set of the subscripts of variables m_i, that is, U=1, ⋯, p. We use the following statistic to evaluate independence between a variable m_j and its residuals $r_i^j = m_i - \frac{cov(m_i,m_j)}{var(m_j)} m_j$ when m_i is regressed on m_j:

$$T_{pwling}(m_j; U) = \sum_{i \in U, i \neq j} \widehat{DMI}(m_i, r_i^j, m_j, r_j^i) \tag{18}$$

Next, we remove the effect of the exogenous variable from the other variables using least squares regression. Then, we can find the second variable in the causal ordering of the original observed variables by analyzing the residuals and their LiNGAM and finding an exogenous residual. The iteration of these effect removal and causal ordering estimates the causal order of the original variables. Formally, the algorithm is defined as Algorithm 2. The causality among resource metrics is shown as the container and server metrics in Figure 31(c).

After the above three steps, MicroDiag constructs a metric causality graph for all components in the cloud microservices. Figure 31(c) gives an example of the constructed metric causality graph from component dependency graph Figure 31(b).

6.3.2.3 *Culprit metric localization*

With the constructed metric causality graph, MicroDiag weighs the graph with the Pearson correlation coefficient between two linked metrics, showing the probability of the anomaly propagation, and ranks the culprit metrics with the PageRank algorithm. The transition probability matrix $\mathbf{P}$ in PageRank is computed as $P_{ij} = \frac{w_{ij}}{\sum_j w_{ij}}$ if node i links to node j, and $P_{ij} = 0$ otherwise, and the teleportation probability is set to $c = 0.15$ as recommended in [212]. MicroDiags returns a ranked list of potential root causes by ranking the nodes in the metric causality graph. We note that the metric causality graph needs to be reversed before conducting the localization procedure, as shown in Figure 31(d). An example of the ranked list of culprit metrics is shown in Figure 31(e), where $m_1^{p_3}$ has the highest probability to be the root cause.

6.4 EVALUATION

In this section, we set up a testbed to evaluate the effectiveness of MicroDiag on locating culprit metrics of performance anomalies and compare our method to two baseline methods.

6.4.1 *Experimental setup*

We use the testbed described in Chapter 5.3.1 using the same cluster settings, benchmark, fault injection tools, and workload generator. Two types of performance anomalies: CPU hog and memory leak are injected into four microservices (catalogue, carts, orders, and users) in sock-shop. For each anomaly, we run the system in normal status for 5 minutes, then we inject one anomaly to one of four microservices for 1 minute and wait another 5 minutes for it to cold down in the system before performing the next fault injection. We repeat our experiments five times, and in total of 40 cases. To quantify the performance of each system, we use the following two evaluation metrics PR@k and AP@k (defined in Chapter 6.2.4.2).

6.4.1.1 *Baseline methods*

We compare our method to two state-of-the-art methods as follows:

- Loud [116]: Loud localizes the culprit metrics by constructing a propagation graph of anomalous metrics using the Granger causality test. To implement Loud, we use the anomalous metrics detected by our anomaly detection and construct the propagation graph using the Granger causality test, then rank the culprit metrics using PageRank after assigning weights to edges.
- CauseInfer [111]: CauseInfer identifies the culprit metrics by constructing a service dependency graph and metric causality graphs for each service using the PC algorithm. To implement CauseInfer, we use the dependency among services in our component dependency graph as the service dependency graph, then use the PC algorithm with partial correlation to get the metric causality graph and rank the culprit metrics with our localization method. We note that the independent test and ranking method in CauseInfer have poor performance in our dataset.

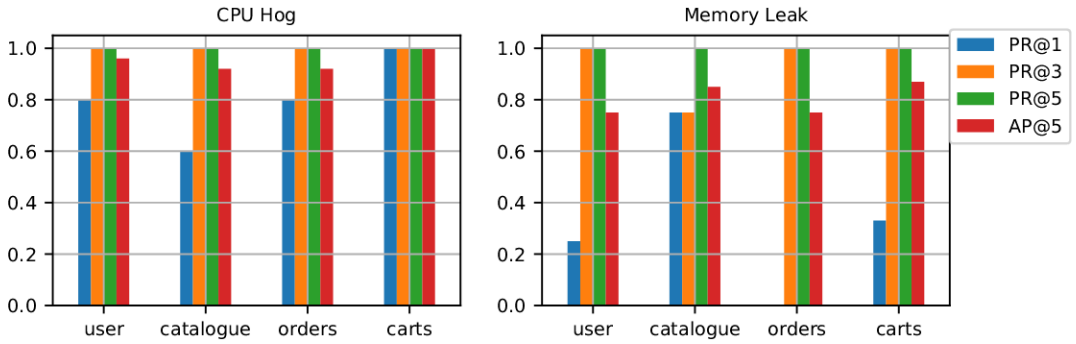


Figure 32: Performance of MicroDiag in terms of PR@1, PR@3, PR@5, and AP@5.

6.4.2 *Effectiveness evaluation*

Figure 32 shows the performance of MicroDiag in identifying the root causes from two types of anomalies injected to four microservices. Overall, we can see that MicroDiag achieves higher performance on CPU hog than memory leak in PR@1, with 80% in PR@1 of CPU hog and 33% in PR@1 for the memory leak. This is because (1) memory leak issue manifests in multiple resource metrics, which is more difficult to identify cause-effect relationships; (2) The monitoring interval is 5 seconds, and it might fail to capture the changes in time, thus failing causal inference. Besides, we can see that MicroDiag pinpoints root cause as one of the top 3 most likely causes, with an average of 100% and 93% for CPU hog and memory leak, respectively.

6.4.3 *Comparison*

Furthermore, we evaluate the performance of MicroDiag by comparing it with two baseline methods.

We apply MicroDiag and two baseline methods to all anomaly cases and get the average performance of each method in terms of PR@1, PR@3, PR@5, and AP@5, as shown in Table 12. We can see that all these methods cannot pinpoint the culprit metric in the top 1 of the ranked list. However, compared to the baseline methods, our MicroDiag achieves a significant improvement with at least 76.5% in precision, and it achieves 97% in PR@3 and 100% in PR@5. Besides, we can see that CauseInfer has a poor performance in identifying root causes, particularly with 9% in PR@1. This is because the PC algorithm has low performance in discovering contemporary effects resource metrics, thus failing to pinpoint root causes. In contrast, Loud uses Granger causality tests which add spurious causal relations among metrics. With the aid of PageRank,

it has a high chance to identify the root cause; therefore, it achieves higher performance in PR@5.

Table 12: Comparison of baseline methods and our MicroDiag.

Metric	Loud	CauseInfer	MicroDiag	Minimum improvement to baseline methods (%)
PR@1	34	9	60	76.5
PR@3	74	49	97	31.1
PR@5	94	69	100	6.4
AP@5	70	42	89	27.1

6.5 CHAPTER SUMMARY

Fine-grained cause localization is beneficial to decrease the scope of investigation and the number of potential recovery actions. Therefore, it is crucial to identify the fine-grained cause or derive some evidence additional to the faulty service for operators to diagnose and recover cloud microservices. However, locating the fine-grained cause from metrics is difficult in cloud microservices due to the increasingly complex dependencies, the scale, and the diversity in metrics.

We presented MicroRCA+ and MicroDiag to address the challenges by applying deep learning and causal inference to fine-grained performance diagnosis in cloud microservices. MicroRCA+ is an extension of coarse-grained diagnosis and applies autoencoders to detect bottleneck metrics, addressing the problem caused by diverse anomaly patterns among services. The evaluation on a microservice benchmark shows that MicroRCA+ can identify the culprit metric with 85.5% in precision.

As MicroRCA+ assumes, the most likely culprit metric deviates significantly to its normal state, which might be violated by issues that manifest themselves in multiple metrics simultaneously, such as the memory leak issue. To this end, we proposed MicroDiag, which is able to capture contemporary propagation in real-time. Before applying CI techniques into fine-grained cause localization, we conducted extensive experiments to investigate the overall performance of this class of methods on identifying root causes of performance anomalies in cloud microservices. MicroDiag identifies the root cause by modeling the anomaly propagation among metrics, using spatial and temporal causal inferences. Anomaly patterns among metrics from interactive

layers in cloud microservices are differentiated with a mixture of causal inference techniques. Experimental results show that MicroDiag can rank 97% of the culprit metric in one of the top 3 most likely causes, outperforming the state-of-the-art methods at least 31.1%.

With the identified fine-grained causes, the space of recovery actions can be reduced. In the next chapter, we present a method to select the best recovery action from the candidates to mitigate the performance anomaly in cloud microservices.

7

MODEL-DRIVEN RECOVERY ACTION SELECTION

Contents

To achieve resilient microservices, proposed solutions ranging from fault-tolerant service design and development, service resilience testing, and self-healing exist or have yet to be developed. Several investigations have focused on resiliency patterns in microservices design [225, 226] and testing [176, 227]. By contrast, less attention has been placed on automatic recovery techniques.

One of the key problems arising in the automatic recovery is to determine: given a detected service performance anomaly, which action(s) should be taken to mitigate it? However, the selection of recovery actions for mitigating an identified performance anomaly in cloud microservice is challenging to achieve due to the ample space of the system states and recovery actions, uncertainties in the system and applications, delusive corrective actions, and limited historical failure data.

In the literature, different approaches have been proposed to recover issues. For example, rule-based approaches select recovery actions by matching the user-defined rules [183]. However, the rules require frequent updates following the corresponding updates of microservices, which conflicts with the goal of automatic recovery. Case-based approaches identify recovery actions by matching previous failure cases [188]. However, their overhead and delay are high due to the numerous metrics in microservices. This also holds true for the learning-based approaches [182]. Further suggested methods select recovery actions by analyzing action properties [198, 199]. However, they are highly dependent on the probabilistic parameters learned from recovery history (e.g., the success rate of action to a given failure) and can be misled by delusive corrective actions. Notably, all the above approaches assume that his-

torical failure data is available for learning, which is not always the case for cloud microservices.

To overcome the shortcoming that the existing methods require historical failure data in the existing methods, we propose an automatic recovery selection method, MicroRAS [2], to mitigate performance anomalies. MicroRAS is a model-based method that can adapt to the frequent changes of microservices without requiring historical data of previous failures and can reduce the potentially destructive consequences of recovery actions by assessing their side effects. MicroRAS firstly models the system state with an attributed graph used to track the propagation of positive and negative effects of recovery actions. Next, it estimates the benefit (positive effects) and the risk (negative effects) associated with each action by predicting the future state, where the system would transit with the selected action. Lastly, it aggregates all these effects into an effectiveness value with fuzzy logic and selects the best possible action with a tradeoff between action effectiveness and the time of the action to mitigate the anomaly.

This chapter includes the following contributions [1].

- We propose a recovery action selection method based on real-time data collection and action properties observed during non-anomalous operation instead of historical failure data (Chapter 7.3).
- We propose an action effects estimation model to capture the positive and negative effects associated with a recovery action, which is adaptive to the anomalous context of the system (Chapter 7.3).
- We evaluate MicroRAS by mitigating different types, levels, and contexts of anomalies. Experimental results show that the actions selected by MicroRAS can mitigate the faulty service well, affecting other services 44.3% less, and complete at least four times faster than baseline recovery strategies (Chapter 7.4).

7.1 MOVITATING EXAMPLE

In this section, we use a concrete example to illustrate the motivation of our proposed method. For the sake of simplicity, we focus on a small part of a large-scale microservice-based application shown in Figure 33. This application consists of five microservices (MS) deployed on two hosts, where *MS 1*, *3* and *5* are co-located on *Host 1*, *MS 1*, *2* and *4* are on *Host 2*. Requests to *MS 1*

1 MicroRAS - Parts of this chapter are published in [228]

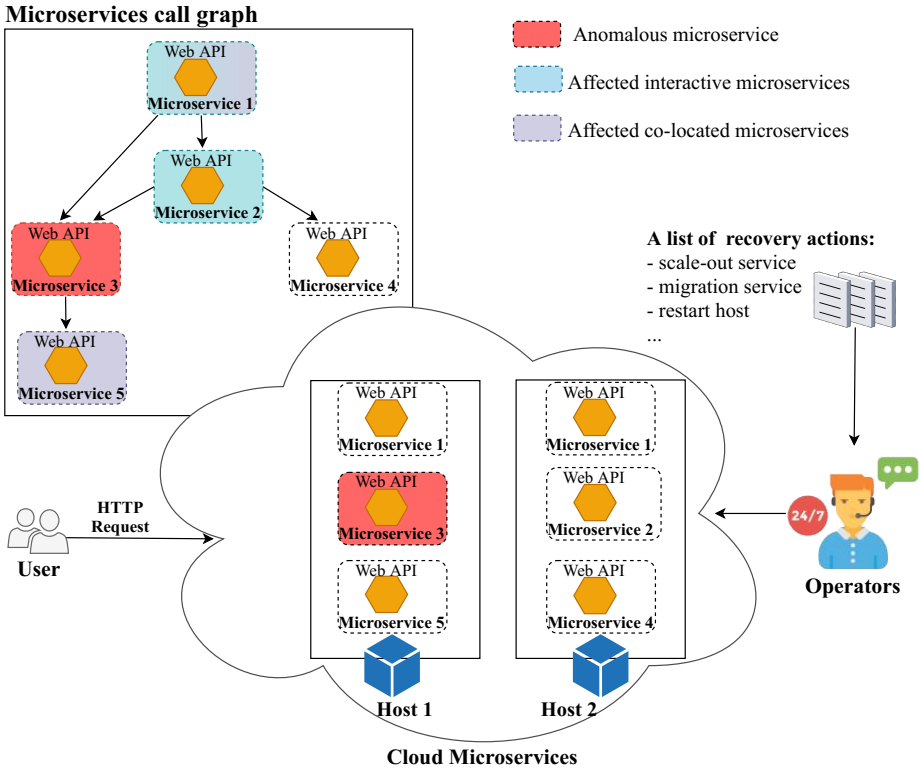


Figure 33: An illustration of the Motivating example.

are load-balanced across two replicas. The interactions among microservices are henceforth referred to as the *microservices call graph*.

Subsequently, slower response times of *MS 3* are observed and classified as a performance anomaly. The operators identify the root cause as *MS 3*, by manually debugging or root cause analysis tools. Meanwhile, they obtain a list of possible recovery actions based on their expert knowledge and previous experience, the latter commonly maintained as scripts or playbooks [37]. The recovery actions can be restart service, scale-out service, restart host, etc.

Let us take *scale-out service* as an example of recovery action. When *MS 3* scales out, the consequences of this action can be diverse. If the recovery time (the time it takes for the action to have an effect and the microservice to recover from the performance anomaly) of scale-out is very short and the available resources are sufficient, the performance anomaly would be mitigated. However, if the recovery time is too long, the performance anomaly could propagate to upstream microservices, i.e., *MS 1* and *2* in the blue box in the call graph, increasing the response times of *MS 1* and *2*. Even worse, as *Microservice 1* is a user-facing microservice, it could cause service disruption for end-users directly. Besides, if the available resources on the host are insuf-

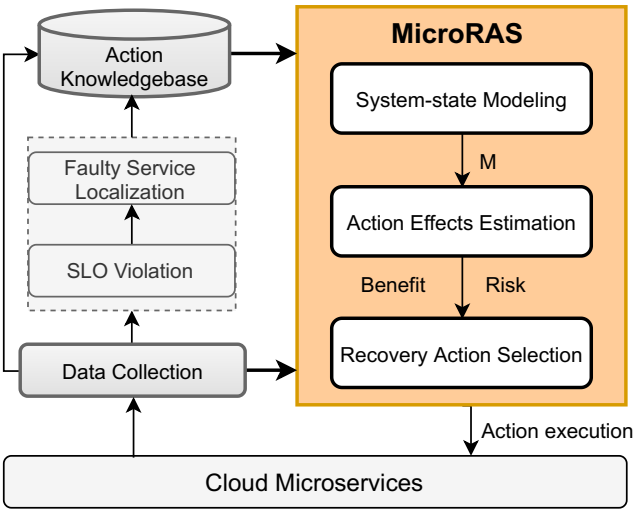


Figure 34: The workflow of recovery action selection.

ficient, the scale-out action might affect the co-located microservices, i.e., *MS 1* and *5* (purple in Figure 1), or even the entire application.

In order to mitigate performance anomalies in cloud microservices without causing significant downtime, it is crucial to identify the appropriate action that can recover the anomalous services while minimizing the side effects on other services and recovery time. We propose a model to assess potential recovery actions' positive and negative effects and select the best possible action with a tradeoff between the effects and the recovery time.

7.2 *microras*: RECOVERY ACTION SELECTION IN CLOUD MICROSERVICES

To adapt to the dynamics of the cloud microservices, where the recovery history is not always available, we propose MicroRAS, a recovery action selection method to automatically mitigate the performance anomalies based on real-time contextual information of the system. The main idea of our method is to understand the anomalous context of the system with a system-state model, using the data collected in real-time, then selecting recovery action by predicting the effects on the recovered system state if an action were to be applied. Recovery time is an essential factor in this decision, as unattended anomalies can propagate quickly, and it is a common objective in the literature [172, 182, 229]; MicroRAS takes both the action effects and recovery time as objectives and models the selection as an optimization problem.

Before the recovery process is initiated, detection of slower response times of microservices, location of the faulty service, and a set of possible recovery actions (a knowledge base) are required. The methods for anomaly detection

and faulty service localization have been well addressed in the literature [41, 132, 209, 230, 231], and the actions a knowledge base can be configured based on properties of the available recovery actions as observed in non-anomalous operations.

Figure 34 shows the workflow of recovery action selection. Once the selection process is triggered, MicroRAS selects the recovery action with the following steps: (1) it gathers the runtime contextual information of the system and models the system state with an attributed graph that can also track the propagation of action effects across services and hosts. (2) it predicts each potential recovery action's benefit and risk by estimating the future state the system would transit into by applying that action. (3) it aggregates the action benefit and risk into an effectiveness value and formulates the action selection as an optimization problem with effectiveness and recovery time as the objectives. After the action execution, the observed state change caused by the action and the recovery time is used to update the action knowledge base. We remark that the runtime complexity of MicroRAS is low, as the time complexity of the operations in our method is linear to the number of the services, nodes, and potential actions. Thus it scales well with the size of the cloud microservices environment.

7.3 THE MICRORAS METHOD

In this section, we describe the three key modules in MicroRAS to select the best appropriate recovery action without historical failure data, namely system-state model (Chapter 7.3.1), action effects estimation (Chapter 7.3.2), and recovery actions selection (Chapter 7.3.3).

7.3.1 *System-state modeling*

To estimate the potential effects of an action on the system, awareness of the system context in different states is necessary. We build a system-state model to capture the context, including the dependency among components in the system and their states of resources.

In a cloud microservices environment, services inter-communicate through lightweight protocols and are deployed across multiple hosts. An action applied to one service does thus influence not only the service itself but also other services, either through invocation paths or their co-located hosts. Understanding service influence is similar to the anomaly propagation problem [213]. Therefore, we model the system-state with an attributed graph that

Table 13: Notations used in MicroRAS.

Notation	Description
A	a list of n_a feasible actions, $A = \{a_i\}_{i=1}^{n_a}$
G	an attributed graph
S	a set of n_s services $S = \{s_i\}_{i=1}^{n_s}$
H	a set of n_h hosts $H = \{h_j\}_{j=1}^{n_h}$
s_i	a service with c pods, $s_i = \{s_{ij}\}_{j=1}^{c}$
s_{ij}	a pod of service s_i, the pod runs on host h_j
SM	a set of system state models, $SM = \{sm^i\}_{i\in\{N,A,R\}}$ each sm^i is represented by a unique $\{G, SV\}$
sm^N,sm^A, sm^R	normal, abnormal, and recovered system state
SV	a set of state variables of n_s services/pods and n_h hosts, $SV = \{sv_k\}_{k=1}^{n_s+n_h}$
sv_k	state variables of a pod/host, $sv_k = \{sv^{RU}, sv^{RA}\}$
sv^{RU}	resource usage (1 vCPU, 1GB memory, etc)
sv^{RA}	resource allocation such as host capacity, pod limits (2 vCPU, 2GB memory)
E_e, E_b, E_r	action effectiveness and its compositions: benefit, risk
$UT(s_{ij}), UT(h_j)$	resource utilization of pod s_{ij}, host h_j (%)
T	recovery time of an action

not only shows the dependencies among services and hosts but also tracks the propagation of action effects. In addition, we define the state of services and hosts in the system as a set of variables SV. As MicroRAS aims at performance anomalies caused by resource bottlenecks, we store in SV the resource usage sv^{RU} and resource allocation sv^{RA}, in terms of CPU, memory, etc. The major notations are summarized in Table 13. We define the system state model as follows:

7.3.1.1 *System-state model*

A set of system states SM, including normal sm^N, abnormal sm^A, and recovered sm^R state, is defined using an attributed graph G together with a set

of system state-variables SV, including resource usage sv^{RU} and resource allocation sv^{RA}. Notably, the normal and abnormal states are fully observable, whereas accurate prediction of sm^{R} is key to select the recovery action.

Once the response time between two services is slow and classified as a performance anomaly, MicroRAS constructs an attributed graph that holds the normal sm^{N} and abnormal sm^{A} system states, using the method proposed in our previous work [209]. In addition, the state variables SV are stored in the node attributes. The data for graph construction and state variables are gathered from the runtime monitoring of hosts and services, including a service mesh.

The attributed graph in Figure 35(a) corresponds to our motivating example in Figure 33. In Figure 35(a), the solid lines indicate service invocations, and the dashed lines show which host the service runs on. For each service and host, we collect resource usage and allocation in normal and abnormal states. In particular, for service s_i, which runs with multiple replicas (pods), we collect the resource data for each of the n_p pods s_{ij}. In Figure 35(a), $n_p = 2$ for service s_1, and 1 for the other services.

7.3.2 *Action effects estimation*

Based on the observed normal and abnormal system states, we estimate effects associated with each potential recovery action by predicting the future state that the system would transit into if applying the action.

Action effects in MicroRAS are composed of positive and negative effects. We define the positive effect as the benefit E_b that the identified anomalous service would achieve in terms of service performance and the negative effect as the risk E_r that the affected hosts would have in terms of resource contention, thus affecting the services that run on the hosts. Due to the uncertainty and complexity of cloud microservices, it is difficult to estimate service performance after recovery action execution accurately. We first estimate the resource utilization UT of service, then map the estimated UT into fuzzy sets of service performance using a fuzzy inference system described in Chapter 7.3.3. Hence, in order to estimate the action effects, we need to predict the recovered state, including the attributed graph and system state-variables after an action execution, in order to identify the potentially affected hosts and compute the resource utilization of the faulty service (action benefit E_b) and affected hosts (action risk E_r).

To predict the recovered state of an action, we need to know the current system state and the properties of the action, as the action affects the system state in different ways. Although there is a wide range of recovery actions, we

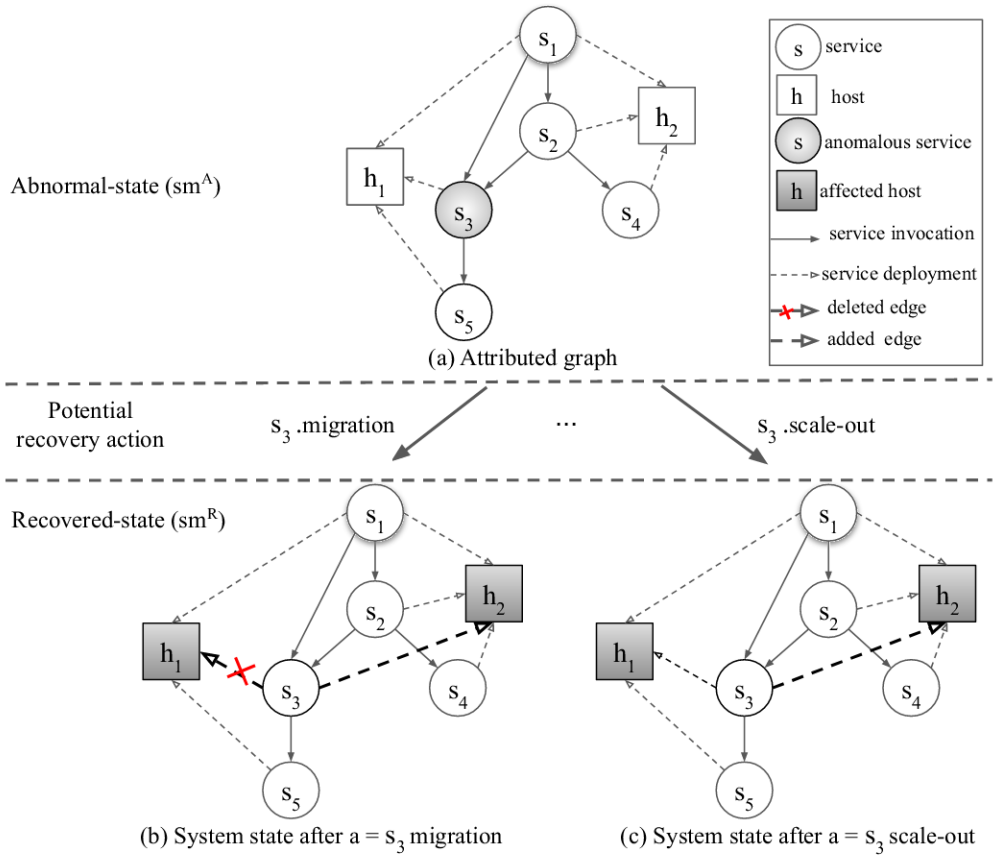


Figure 35: System states prediction.

only consider how the action modifies the system-state model. For a recovery action a_i in action set A, we include the following properties:

- Topology: Some actions change the service location, thus changing the topology of the attributed graph.
- Resource usage: Some actions change the resource consumption of the service or host. For example, *restart* can recover anomalies caused by memory leaks, thus reducing resource usage.
- Resource allocation: Some actions change the capacity of hosts or the resource limits of pods. Examples include *scale-up* and *scale-out* actions that increase the allocated resources of a host or a service.

Note that these properties, including the recovery time used in Chapter 7.3.3, are stored in the *action knowledge base* in Figure 34. All properties are obtained in non-anomalous operations and can be updated after the action is executed.

Based on the abnormal state sm^A and the *topology* property of a recovery action, MicroRAS predicts the graph changes in recovered state sm^R. Figures 35(b) and (c) show the recovered states after migrating and scaling out anomalous service s_3, where service migration removed the link between s_3 and h_1 and adds a new link between s_3 and h_2, whereas service scale-out adds a new link between s_3 and h_2. In these two recovery actions, the affected hosts are h_1 and h_2.

After the attributed graph is predicted, MicroRAS estimates the resource utilization of the faulty service and affected hosts. When an action applies to pod s_{ij} of faulty service s_i or affects host h_j, the resource utilization in recovered state sm^R is defined as the ratio between resource usage and allocation:

$$UT(h_j, sm^R) = \frac{sv^{RU}(h_j, sm^R)}{sv^{RA}(h_j, sm^R)} \tag{19}$$

$$UT(s_{ij}, sm^R) = \frac{sv^{RU}(s_{ij}, sm^R)}{sv^{RA}(s_{ij}, sm^R)} \tag{20}$$

Assuming an ideally equal load balancing between pods of a service, the utilization of service s_i is defined as:

$$UT(s_i, sm^R) = \frac{1}{n_p} \sum_{j=1}^{n_p} UT(s_{ij}, sm^R). \tag{21}$$

where n_p is the total number of pods. As some actions may under-provision the resource, the estimated UT can be over 1, thus its range is defined as $UT > 0$.

Based on the action properties and system state variables in normal and abnormal states, MicroRAS estimates the resource usage and resource allocation of pod s_{ij} and host h_j in the recovered state as follows.

The future resource usage of a pod in a recovered state varies with the recovery action. If the action modifies the pod, it is the configured resource usage Δsv^{RU}; If the action newly creates the pod, it is assigned with the service normal resource usage after load-balancing. Otherwise, the pod keeps the abnormal resource usage. Taking Figure 35 as an example, the resource usage of pod s_{32} in action *migration* in Figure 35(b) is Δsv^{RU}; The resource usage of pod s_{31} in action *scale-out* in Figure 35(c) keeps the abnormal resource usage $sv^{RU}(s_{31}, sm^A)$, and pod s_{32} is assigned with the load-balanced normal resource usage of service s_3, which is $sv^{RU}(s_3, sm^N)/2$. We summarize the pod resource usage in Equation 22.

$$sv^{RU}(s_{ij}, sm^{R}) = \begin{cases} \Delta sv^{RU}, & \text{if } s_{ij} \text{ is modified,} \\ sv^{RU}(s_{ij}, sm^{A}), & \text{if } s_{ij} \text{ is not modified,} \\ \frac{sv^{RU}(s_i, sm^{N})}{c}, & \text{if } s_{ij} \text{ is a new pod.} \end{cases} \tag{22}$$

Pod future resource allocation $sv^{RA}(s_{ij}, sm^{R})$ depends on the pod limits and the available resources of host h_j it runs on. If the available resources in h_j are sufficient (exceeds the pod limits), $sv^{RA}(s_{ij}, sm^{R})$ is equal to the pod limits and otherwise to the available resources in h_j. The pod limits can be modified by the action with Δsv^{RA} or kept in an abnormal state. The available resource of h_j is the host resource allocation $sv^{RA}(h_j, sm^{R})$ with a consumption of $sv^{RU}(h_j, sm^{A})$, where $sv^{RA}(h_j, sm^{R})$ can be modified by the action or remain the same as $sv^{RA}(h_j, sm^{A})$:

$$sv^{RA}(h_j, sm^{R}) = \begin{cases} \Delta sv^{RA}, & \text{if } h_j \text{ is modified,} \\ sv^{RA}(h_j, sm^{A}) & \text{otherwise.} \end{cases} \tag{23}$$

Once the future pod resource usage (Equation 22) and host resource allocation (Equation 23) are determined, we can estimate the future resource utilization of the affected host h_j where the pod s_{ij} runs on, as shown in Equation 24. Host resource usage $sv^{RU}(h_j, sm^{A})$ increases by pod resource usage $sv^{RU}(s_{ij}, sm^{A})$ if pod s_{ij} is migrated to h_j, or decreases by $sv^{RU}(s_{ij}, sm^{A})$ if s_{ij} is migrated from h_j to another host.

$$UT(h_j, sm^{R}) = \frac{sv^{RU}(h_j, sm^{A}) \pm sv^{RU}(s_{ij}, sm^{A})}{sv^{RA}(h_j, sm^{R})} \tag{24}$$

Future resource utilization of h_j is summarized in Equation 25. For each host that service s_i runs on, we calculate the resource utilization and use the maximum utilization as the risk of the action.

$$UT(h_j, sm^{R}) = \begin{cases} \frac{\Delta sv^{RU}}{sv^{RA}(h_j, sm^{R})}, & \text{if } h_j \text{ is modified,} \\ \text{as per Equation 24,} & \text{if } s_{ij} \text{ is migrated.} \end{cases} \tag{25}$$

7.3.3 *Recovery action selection*

After we estimate the benefit and risk associated with each recovery action in terms of resource utilization, we map these into fuzzy sets of service performance and aggregate them into a single crisp effectiveness value through

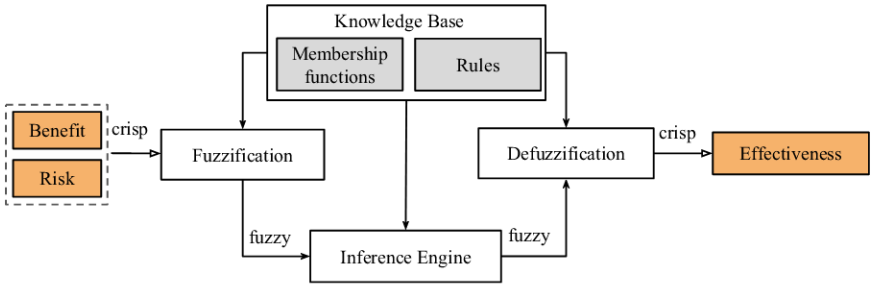
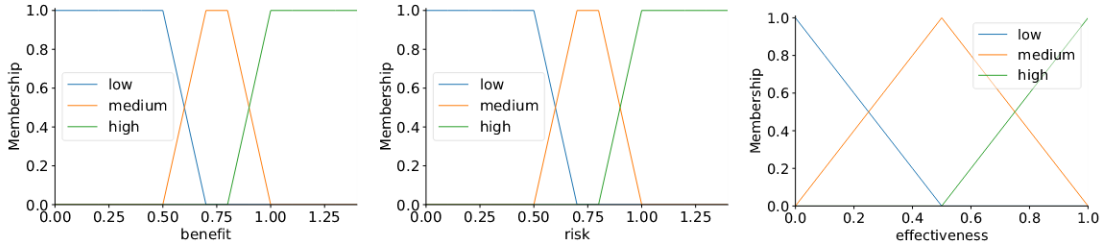


Figure 36: The structure of the fuzzy inference to combine action risk and benefit.



(a) benefit function (b) risk function (c) effectiveness function

Figure 37: Fuzzy membership functions.

a fuzzy inference system [232], illustrated in Figure 36. Finally, we formulate the selection problem as an optimization problem and select an appropriate action with a tradeoff between action effectiveness and recovery time.

In order to calculate the effectiveness value, the fuzzy inference system uses membership functions to determine the degree that its inputs belong to each of the relevant fuzzy sets. For this purpose, three overlapping fuzzy sets are created. For the action risk, host resource utilization values between 0 and 70% are in the Low range, values between 50% and 80% are in the Medium range, and values above 80% are in the High range.

A membership function defines how the input value is mapped to the membership degree between 0 and 1, where 0 means the input does not belong to the given fuzzy set, and 1 means the input ultimately belongs to it. Similar to [199, 233], the membership functions for the three fuzzy sets in inputs are respectively an R-function, a trapezoidal function, and an L-Functions, as shown in Figure 37(a). The membership function used in the output is three triangular functions, as shown in Figure 37(b). Taking the action risk 0.7 as an example, according to its membership function in Figure 37(a), it has membership degree 0.2 in the Low set, 0.7 in the Medium set, and 0 in the High set. These values are used for the fuzzy rules in the fuzzy reasoning. The

Table 14: Fuzzy rules for action effectiveness.

Benefit	Risk	Effectiveness
low	high	low
low	medium	low
low	low	medium
medium	high	low
medium	medium	medium
medium	low	medium
high	high	low
high	medium	medium
high	low	high

fuzzy rules for the inference system are defined based on the cloud microservices and their administrative policy. MicroRAS uses the fuzzy rules shown in Table 14.

Based on the inputs, some fuzzy rules are fired and integrated. The decisions are made according to the aggregation of the fired fuzzy rules. The aggregated fired fuzzy rules output a single fuzzy set which is the input of the defuzzification procedure. We use the centroid method for defuzzification to convert the fuzzy set into a crisp effectiveness value.

After obtaining the effectiveness values of the potential recovery actions, we formulate the action selection as an optimization problem, taking the effectiveness and action recovery time as the objectives. The recovery time of an action is measured as the time between the action initiation and completion in normal status, which initially was obtained by executing the recovery action in non-production environments. Once the action is executed actually to recover an anomaly, the recovery time is updated with the time between action initiation and action taking effect.

Given a set of potential recovery actions A, for each recovery action $a_i \in A$, its effectiveness is $E_e(a_i)$ and its recovery time is $T(a_i)$. For consistency purposes, we normalize the values of effectiveness and recovery time into the range $(0, 1)$ through Min-Max normalization. The performance of action a_i is quantified with a utility function $u(a_i)$:

$$u(a_i) = w_e E_e(a_i) - w_t T(a_i) \tag{26}$$

where w_e and w_t are user-defined weights for action effectiveness and recovery time ($w_e + w_t = 1, 0 < w_e, w_t < 1$). By setting the weights, users can prioritize the effectiveness and recovery time. We finally select the action that

has the highest utility value among the recovery action in A according to Equation 26.

7.4 EVALUATION

In this section, we evaluate the performance of MicroRAS through experiments on a cloud testbed. The experimental setup, evaluation results, and comparisons are presented.

7.4.1 *Experimental setup*

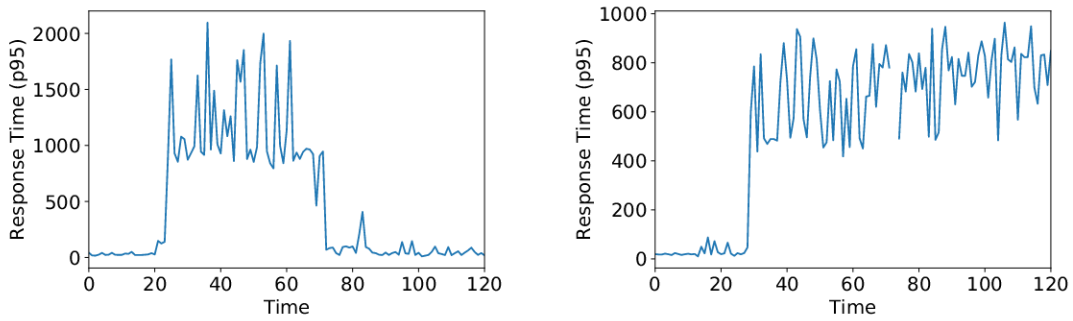
We evaluate MicroRAS in the same testbed presented in Chapter 5.3, using the same cluster settings, microservices benchmark, workload generator. However, we extend the fault injection from service to server host and use different evaluation metrics described as follows.

7.4.1.1 *Faults injection*

We evaluate our MicroRAS with two different types of anomalies (CPU hog and memory leak), different levels of anomalies (stressing services and hosts), and different contexts (with total cluster resources either sufficient or insufficient to resolve the anomaly). To inject the CPU hog and memory leak, we use stress-ng[6], a tool to load and stress computer systems to exhaust the CPU and memory resources continuously. We customize the existing sock-shop docker images to inject performance anomalies in microservices by installing the faults injection tool. The injected microservice is *catalogue*, and the injected host is the host *catalogue* runs on. In the cluster resources sufficient scenario, we only inject anomalies to service or host. In the cluster resources insufficient scenario, we also stress the other hosts. The details of the anomaly scenarios are shown in Table 15.

Table 15: Details of anomaly scenarios.

anomaly type		host-level	service-level
CPU Hog (vCPU * %)	cluster sufficient	4*95	3*95
	cluster insufficient (other hosts)	4*80	4*80
Memory Leak (vm * %)		1*73	2*50



(a) scale-out pod (b) restart pod

Figure 38: Two recovery actions for service performance anomaly caused by insufficient host resources: (a) scale-out pod recovers the anomaly, but (b) restart pod has no effect.

In each case, we run the microservices in normal status for 2 minutes with the workload generator running. We next introduce the anomaly and let it run for 3 minutes before MicroRAS is used to select and execute a recovery action. After executing an action, we collect another 5 minutes of data to measure the action consequences. To increase the generality, we repeat 3-5 times for each anomaly scenario. This produces a total of 23 experimental cases. In each anomaly scenario, we take 6 types of recovery actions, which are: *no action*, *restart pod* (in the same host), *migrate pod* (shutdown and start the pod again), *scale-out pod*, *scale-up pod* and *restart host*. Figure 38 gives two examples of data collected after applying scale-out pod and restart pod when the host CPU resource is insufficient, with pod scale-out (Figure 38(a)) having a positive effect while pod restart (Figure 38(b)) did not recover the anomaly.

7.4.1.2 *Evaluation metrics*

To quantify the performance of recovery action selection, we use the following metrics:

- Recovered Percentage (RP) quantifies the positive effects of a recovery action a on the anomalous service s_a. It is defined as the percentage of service performance recovered from abnormal state $perf(s_a, m^A)$ to recovered state $perf(s_a, m^R)$ with action a, to the abnormal deviation from normal state $perf(s_a, m^N)$.

$$RP(a) = \frac{perf(s_a, m^A) - perf(s_a, m^R|a)}{perf(s_a, m^A) - perf(s_a, m^N)} \tag{27}$$

- Affected Percentage (AP) quantifies the negative effects of a recovery action on affected services $\{s_i|i = 1, 2, ...N\}$, where N is the number of

affected services. AP is defined as the mean percentage of decreased performance for affected services from abnormal state $perf(s_i, m^A)$ to recovered state $perf(s_i, m^R|a)$ with action a, to the normal state $perf(s_i, m^N)$:

$$AP(a) = \frac{1}{N} \sum_{s_i} \frac{perf(s_i, m^R|a) - perf(s_i, m^A)}{perf(s_i, m^N)} \tag{28}$$

- Recovery Time (RT) quantifies time from initiating the mitigation action until the performance of the anomalous service and any affected services have stabilized.

7.4.1.3 *Baseline methods*

We compare MicroRAS with three recovery strategies which require no historical data and are commonly used in the comparisons in the literature:

- No Action: Here, the operation team just passively observes the system without taking any actions. This strategy shows the potential damages of the injected performance anomalies when left unattended.
- Random Selection: This strategy might be adopted when the operation team cannot determine the correct recovery action precisely but urgently is trying to fix the problem. The operating team randomly selects one action from the candidates and applies it [229].
- Restart: This is a prevalent recovery strategy, which can be applied at various levels. In a production environment, a significant fraction of failures can be cured by restarts [66]. We perform restarts at the host or pod level to resolve host and service level anomalies, respectively.

7.4.2 *Experimental results*

In our experiments, under normal workload, the 50th percentile (p50) of service response times is around 10 ms in normal status and is in range (35 ms, 300 ms) in abnormal status, depending on the anomaly types; the 95th percentile (p95) of response times is around 40 ms in normal status and ranges from 160 ms to 2000 ms in abnormal status.

Figure 39 shows the results of our proposed recovery action selection method for mitigating different anomaly scenarios. For each anomaly scenario, the bar charts show the mean recovered percentage (RP) and affected percentage (AP) in terms of p95 and p50 of response times, and the dashed line shows the mean recovery time.

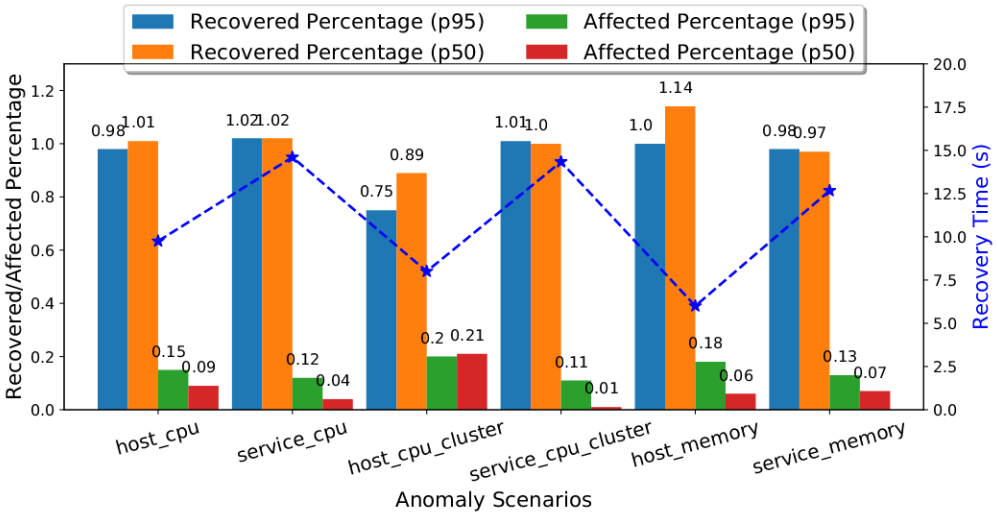


Figure 39: MicroRAS performance in terms of p95 and p50.

Table 16: Performance of MicroRAS in different types of anomaly scenarios.

Fault Scenario	CPU Hog	Memory Leak	Host-level	Service-level	Cluster Sufficient	Cluster Insufficient	Overall
Recovered Percentage	1.002	0.99	0.91	**1.007**	1.002	0.883	0.947
Affected Percentage	0.137	0.155	0.178	**0.12**	0.137	0.156	0.152
Recovery Time(s)	12.175	9.333	**7.917**	13.867	12.175	11.167	10.913

We aggregate RP p95 and AP p95 according to the type of anomaly scenarios in Table 16. We can see that MicroRAS overall can recover 0.947 of anomalous service degraded performance and mitigate the anomalies on average in 11 seconds. The performance of the service-level is better than host-level. This is because the service-level anomalies can be recovered entirely with the provided actions. However, the host-level anomalies can only be mitigated by most of the provided actions; further actions such as cluster scale-out are required to fix the anomalies entirely. For the same reasons, MicroRAS performs better in cluster sufficient than cluster insufficient cases.

7.4.3 *Comparisons*

To evaluate the performance of MicroRAS further by comparing it with the baseline methods. We compare the performance of each recovery strategy on different anomaly scenarios in terms of RP, AP, and recovery time in Figure 40. We observe that the performance anomalies cannot recover or even deteriorate

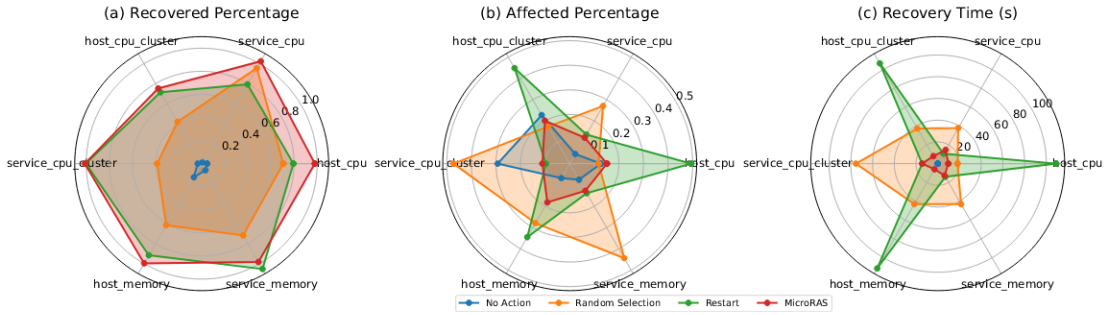


Figure 40: Performance summary for different strategies, performance metrics, and anomaly scenarios.

and affect other services if no action is taken. All the actions selected by the other three strategies can improve the performance of anomalous services. Notably, MicroRAS selects the best action in all scenarios but one, only slightly beaten by restart for memory leaks at the service level (Figure 40(a)) and has a shorter recovery time (Figure 40(c)) than others.

Both Restart and our MicroRAS have a good performance in terms of RP to all types of anomaly scenarios. However, Restart has a higher risk of affecting other services and longer recovery times when the anomaly exists at the host level. Figure 41 shows the AP and number of affected services in each anomaly case. The solid lines show the results of MicroRAS, and the dashed lines show the results of Restart. We observe that MicroRAS and Restart have similar AP and affected number for service-level anomalies. However, Restart has a higher AP and affected number for host-level anomalies. This is because, compared to service-level operations, restarting a host takes longer,

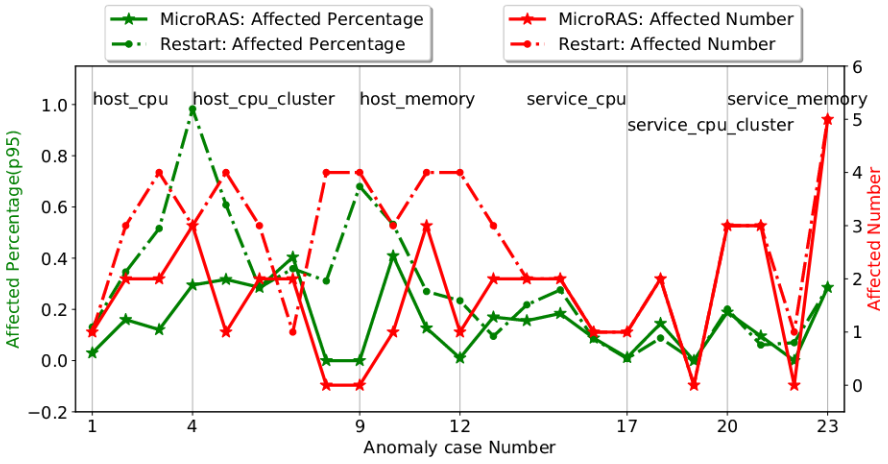


Figure 41: Affected percentage and number comparison.

and all the services running on the host would be restarted, which introduces fluctuations and uncertainty in the system.

Finally, we compare the overall performance of all strategies for all types of anomaly scenarios. Table 17 shows the performance, in terms of RP, AP, and recovery time (RT), for the four recovery strategies. We can observe that MicroRAS outperforms other strategies overall. In particular, MicroRAS achieves a recovered percentage of 94.7%, affecting other services at least 44.3% less and completing at least four times faster than other strategies.

Table 17: Overall performance of different strategies.

Metrics	**RP**	**AP**	**RT(s)**
No Action	0.037	0.138	-
Random Selection (RS)	0.646	0.273	40.565
Restart	0.897	0.295	62.652
MicroRAS	0.947	0.152	10.913
Improvement to RS(%)	46.6	44.3	73.1
Improvement to Restart (%)	5.5	48.5	82.6

7.5 CHAPTER SUMMARY

This chapter presents a method named MicroRAS to select the best possible recovery action based on an action effectiveness assessment model to mitigate performance degradation in cloud microservices. We estimated the positive and negative effects for each action and select the action with the best tradeoff between action effectiveness and recovery time, using data collected in real-time and knowledge obtained in normal status without the use of historical failure data. A system state model is used to estimate the effects, represented by an attributed graph that tracks the propagation of action effects across services and hosts. Experimental results show that MicroRAS can effectively recover the anomalous services by 94.7% of their degraded performance while affecting the performance of other services at least 44.3% less and mitigating the anomaly at least four times faster than several baseline strategies.

8 CONCLUSION

This thesis presents automatic performance diagnosis and recovery methods to improve the design, operation, and reliability of cloud microservices.

To diagnose the origin of a performance anomaly, we presented an application-agnostic method named MicroRCA, which aims to identify the bottleneck service in real-time. MicroRCA constructs an attributed graph to model the anomaly propagation among services and correlates service anomalous performance symptoms with the corresponding resource utilization to infer the anomalous services. Experimental evaluations on a microservice benchmark show that MicroRCA achieves 89% in precision and 97% in MAP, outperforming 13% in precision over several baseline methods. In addition, MicroRCA adapts well to the heterogeneity of microservices and is robust to the results of performance anomaly detection.

This thesis further investigates methods for locating the reason for the occurrence of a performance anomaly, which aim to localize the culprit metrics that provide evidence to explain why the performance anomaly occurs. We presented two methods, MicroRCA+ and MicroDiag, which employ deep learning and Spatio-temporal CI, respectively. The MicroRCA+ method applies autoencoders to the relevant performance metrics of a faulty service and leverages reconstruction errors to rank the anomalous metrics. Experimental evaluation shows that MicroRCA+ can identify the culprit metrics well with 85.5% in precision. MicroDiag models the anomaly propagation across metrics with a metric causality graph, using Spatio-temporal causal inference. It addresses the challenge of diverse anomaly patterns in metrics with a mixture of causal inference. Experimental results show that MicroDiag can rank 97% of the culprit metrics in one of the top 3 most likely causes, outperforming at least 31.1% of the state-of-the-art methods. Moreover, to fully understand the overall performance of causal inference techniques on microservices performance diagnosis, we conducted a comprehensive evaluation by applying six representative CI techniques to locate root causes of a range of performance anomalies injected into microservices benchmarks.

Lastly, with the identified root causes, we presented a method named MicroRAS to select the most appropriate recovery action to mitigate the performance anomaly. This decision-making process is based on an action effectiveness assessment model that estimates the positive and negative effects of each

candidate recovery action. The best possible recovery action is selected with a tradeoff between the effectiveness of an action and its recovery time, thus minimizing the MTTR. Experimental results show that MicroRAS can effectively recover the anomalous services by 94.7% of their degraded performance while affecting the performance of other services at least 44.3% less and mitigating the anomaly at least four times faster than several baseline recovery strategies.

Although this thesis has addressed the central aspects of automatic performance diagnosis and recovery of performance anomalies in cloud microservices, there are several interesting directions for further investigations. These directions can be derived from the key limitations of our current methods. The first key limitation exists in the performance diagnosis. We would like to combine all three observability of cloud microservices (i.e., logs, traces, and metrics) to diagnose a broader range of root causes in cloud microservices. The second key limitation lies in the recovery action selection. Our existing method considers a single-step ahead mitigation only, and some performance anomalies cannot recover completely. Therefore, we would like to investigate multi-step recovery strategies thus to resolve more types of anomalies. There is also a third key limitation in our monitoring tools. The involved tools (Kubernetes, Istio, etc.) are not designed to operate with sub-second monitoring intervals. This adds a limitation to how fast anomalies can be detected, rendering our methods unable to identify anomalous services for which the anomalies propagate faster than our monitoring interval. Therefore, a possible extension is to investigate or develop monitoring tools to collect fine-grained metrics for cloud microservices.

Nevertheless, the results presented in this thesis already show that our methods for locating root causes and recommending recovery actions are capable of supporting automatic performance diagnosis and recovery in cloud microservices, improving service reliability, and relieving operators from troublesome and tedious work.

BIBLIOGRAPHY

[1] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. "Cloud migration research: a systematic review." In: *IEEE transactions on cloud computing* 1.2 (2013), pp. 142–157.

[2] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Migrating to cloud-native architectures using microservices: an experience report." In: *European Conference on Service-Oriented and Cloud Computing*. Springer. 2015, pp. 201–215.

[3] Dennis Gannon, Roger Barga, and Neel Sundaresan. "Cloud-Native Applications." In: *IEEE Cloud Computing* 4.5 (2017), pp. 16–21.

[4] Björn Butzin, Frank Golatowski, and Dirk Timmermann. "Microservices approach for the internet of things." In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2016, pp. 1–6.

[5] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. "Research on architecting microservices: Trends, focus, and potential for industrial adoption." In: *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE. 2017, pp. 21–30.

[6] *The Future of Application Development and Delivery Is Now: Containers and Microservices Are Hitting the Mainstream*. URL: https://www.nginx.com/resources/library/app-dev-survey/. (accessed: 01.09.2021).

[7] *Microservices Adoption in 2020*. URL: https://www.oreilly.com/radar/microservices-adoption-in-2020/. (accessed: 01.09.2021).

[8] Sam Newman. *Building microservices*. " O'Reilly Media, Inc.", 2021.

[9] Johannes Thönes. "Microservices." In: *IEEE software* 32.1 (2015), pp. 116–116.

[10] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices architecture enables devops: Migration to a cloud-native architecture." In: *Ieee Software* 33.3 (2016), pp. 42–52.

[11] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. "Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned." In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 970–973.

[12] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. "Microservices: The journey so far and challenges ahead." In: *IEEE Software* 35.3 (2018), pp. 24–35.

[13] *Website performance: how to optimize website speed and avoid downtime*. URL: https://datadome.co/bot-management-protection/website-performance-how-to-increase-your-business-by-blocking-bots/. (accessed: 01.09.2021).

[14] *The hour the internet broke' cost retailers £1bn according to experts*. URL: https://www.chargedretail.co.uk/2021/06/09/the-hour-the-internet-broke-cost-retailers-1bn-according-to-experts/. (accessed: 01.09.2021).

[15] *The Story of Netflix and Microservices*. URL: https://www.geeksforgeeks.org/the-story-of-netflix-and-microservices/. (accessed: 01.09.2021).

[16] *How Uber Is Monitoring 4,000 Microservices with Its Open Sourced Prometheus Platform*. URL: https://www.cncf.io/uber-case-study/. (accessed: 01.09.2021).

[17] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. "A view of cloud computing." In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[18] Peter Mell, Tim Grance, et al. "The NIST definition of cloud computing." In: (2011).

[19] *Best Infrastructure as a Service (IaaS) Providers*. URL: https://www.g2.com/categories/infrastructure-as-a-service-iaas. (accessed: 01.09.2021).

[20] *Best Cloud Platform as a Service (PaaS) Software*. URL: https://www.g2.com/categories/cloud-platform-as-a-service-paas. (accessed: 01.09.2021).

[21] *10 Popular Software as a Service (SaaS) Examples*. URL: https://getnerdio.com/academy/10-popular-software-service-examples/. (accessed: 01.09.2021).

[22] Adalberto R Sampaio, Harshavardhan Kadiyala, Bo Hu, John Steinbacher, Tony Erwin, Nelson Rosa, Ivan Beschastnikh, and Julia Rubin. "Supporting microservice evolution." In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2017, pp. 539–543.

[23] Aditya Bhardwaj and C Rama Krishna. "Virtualization in Cloud Computing: Moving from Hypervisor to Containerization—A Survey." In: *Arabian Journal for Science and Engineering* (2021), pp. 1–17.

[24] Claus Pahl and Brian Lee. "Containers and clusters for edge cloud architectures–a technology review." In: *2015 3rd international conference on future internet of things and cloud*. IEEE. 2015, pp. 379–386.

[25] Devki Nandan Jha, Saurabh Garg, Prem Prakash Jayaraman, Rajkumar Buyya, Zheng Li, and Rajiv Ranjan. "A holistic evaluation of docker containers for interfering microservices." In: *2018 IEEE International Conference on Services Computing (SCC)*. IEEE. 2018, pp. 33–40.

[26] Hui Kang, Michael Le, and Shu Tao. "Container and Microservice Driven Design for Cloud Infrastructure DevOps." In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*. 2016, pp. 202–211.

[27] Ouafa Bentaleb, Adam SZ Belloum, Abderrazak Sebaa, and Aouaouche El-Maouhab. "Containerization technologies: taxonomies, applications and challenges." In: *The Journal of Supercomputing* (2021), pp. 1–38.

[28] *Kubernetes*. URL: https://kubernetes.io/. (accessed: 01.09.2021).

[29] *Swarm mode overview*. URL: https://docs.docker.com/engine/swarm/. (accessed: 01.09.2021).

[30] *Mesos*. URL: http://mesos.apache.org/. (accessed: 01.09.2021).

[31] *Microservices - a definition of this new architectural term*. URL: https://martinfowler.com/articles/microservices.html. (accessed: 01.09.2021).

[32] A. Sill. "The Design and Architecture of Microservices." In: *IEEE Cloud Computing* 3.5 (2016), pp. 76–80.

[33] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. "Microservices: Yesterday, today, and tomorrow." In: *Book: Present and Ulterior Software Engineering* (2017), pp. 195–216.

[34] Amine El Malki and Uwe Zdun. "Guiding architectural decision making on service mesh based microservice architectures." In: *European Conference on Software Architecture*. Springer. 2019, pp. 3–19.

[35] Ozair Sheikh, Serjik Dikaleh, Dharmesh Mistry, Darren Pape, and Chris Felix. "Modernize digital applications with microservices management using the istio service mesh." In: *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*. 2018, pp. 359–360.

[36] *The RED Method: key metrics for microservices architecture*. URL: https://www.weave.works/blog/the-red-method-key-metrics-for-microservices-architecture/. (accessed: 01.09.2021).

[37] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site reliability engineering: How Google runs production systems*. " O'Reilly Media, Inc.", 2016.

[38] *The USE Method*. URL: https://www.brendangregg.com/usemethod.html. (accessed: 01.09.2021).

[39] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly detection: A survey." In: *ACM computing surveys (CSUR)* 41.3 (2009), pp. 1–58.

[40] Douglas M Hawkins. *Identification of outliers*. Vol. 11. Springer, 1980.

[41] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. "Performance anomaly detection and bottleneck identification." In: *ACM Computing Surveys (CSUR)* 48.1 (2015), pp. 1–35.

[42] Jean-Claude Laprie. "Dependable computing and fault-tolerance." In: *Digest of Papers FTCS-15* 10.2 (1985), p. 124.

[43] Markus C Huebscher and Julie A McCann. "A survey of autonomic computing—degrees, models, and applications." In: *ACM Computing Surveys (CSUR)* 40.3 (2008), pp. 1–28.

[44] *Dynatrace*. URL: https://www.dynatrace.com/. (accessed: 01.09.2021).

[45] *AppDynamics*. URL: https://www.appdynamics.com/. (accessed: 01.09.2021).

[46] HaiZhou Du and YongBin Li. "An Improved BIRCH Clustering Algorithm and Application in Thermal Power." In: *2010 International Conference on Web Information Systems and Mining*. Vol. 1. 2010, pp. 53–56. DOI: 10.1109/WISM.2010.123.

[47] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. "BIRCH: an efficient data clustering method for very large databases." In: *ACM sigmod record* 25.2 (1996), pp. 103–114.

[48] Yasi Wang, Hongxun Yao, and Sicheng Zhao. "Auto-encoder based dimensionality reduction." In: *Neurocomputing* 184 (2016), pp. 232–242.

[49] Yunchen Pu, Zhe Gan, Ricardo Henao, Xin Yuan, Chunyuan Li, Andrew Stevens, and Lawrence Carin. "Variational autoencoder for deep learning of images, labels and captions." In: *Advances in neural information processing systems* 29 (2016), pp. 2352–2360.

[50] Chong Zhou and Randy C Paffenroth. "Anomaly detection with robust deep autoencoders." In: *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 2017, pp. 665–674.

[51] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[52] Geoffrey E Hinton and Ruslan R Salakhutdinov. "Reducing the dimensionality of data with neural networks." In: *science* 313.5786 (2006), pp. 504–507.

[53] Kenneth J Rothman and Sander Greenland. "Causation and causal inference in epidemiology." In: *American journal of public health* 95.S1 (2005), S144–S150.

[54] Markus Gangl. "Causal inference in sociological research." In: *Annual review of sociology* 36 (2010), pp. 21–47.

[55] Judea Pearl. "The seven tools of causal inference, with reflections on machine learning." In: *Communications of the ACM* 62.3 (2019), pp. 54–60.

[56] Peter Spirtes, Clark N Glymour, Richard Scheines, and David Heckerman. *Causation, prediction, and search*. MIT press, 2000.

[57] Austin Nichols. "Causal inference with observational data." In: *The Stata Journal* 7.4 (2007), pp. 507–541.

[58] Clive WJ Granger. "Investigating causal relations by econometric models and cross-spectral methods." In: *Econometrica: journal of the Econometric Society* (1969), pp. 424–438.

[59] Peter Spirtes and Kun Zhang. "Causal discovery and inference: concepts and recent methodological advances." In: *Applied informatics*. Vol. 3. 1. Springer. 2016, p. 3.

[60] David Maxwell Chickering. "Optimal structure identification with greedy search." In: *Journal of machine learning research* 3.Nov (2002), pp. 507–554.

[61] Dan Geiger, Thomas Verma, and Judea Pearl. "d-separation: From theorems to algorithms." In: *Machine Intelligence and Pattern Recognition*. Vol. 10. Elsevier, 1990, pp. 139–148.

[62] Jakob Runge, Peer Nowack, Marlene Kretschmer, Seth Flaxman, and Dino Sejdinovic. "Detecting and quantifying causal associations in large nonlinear time series datasets." In: 5.11 ().

[63] Judea Pearl. "Causal inference in statistics: An overview." In: *Statistics surveys* 3 (2009), pp. 96–146.

[64] Shohei Shimizu, Patrik O Hoyer, Aapo Hyvärinen, and Antti Kerminen. "A linear non-Gaussian acyclic model for causal discovery." In: *Journal of Machine Learning Research* 7.Oct (2006), pp. 2003–2030.

[65] Patrik Hoyer, Dominik Janzing, Joris M Mooij, Jonas Peters, and Bernhard Schölkopf. "Nonlinear causal discovery with additive noise models." In: *Advances in neural information processing systems* 21 (2008), pp. 689–696.

[66] Chengwel Wang, Soila P Kavulya, Jiaqi Tan, Liting Hu, Mahendra Kutare, Mike Kasick, Karsten Schwan, Priya Narasimhan, and Rajeev Gandhi. "Performance troubleshooting in data centers: an annotated bibliography?" In: *ACM SIGOPS Operating Systems Review* 47.3 (2013), pp. 50–62.

[67] Ma łgorzata Steinder and Adarshpal S Sethi. "A survey of fault localization techniques in computer networks." In: *Science of computer programming* 53.2 (2004), pp. 165–194.

[68] Jacopo Soldani and Antonio Brogi. "Anomaly Detection and Failure Root Cause Analysis in (Micro) Service-Based Cloud Applications: A Survey." In: *arXiv preprint arXiv:2105.12378* (2021).

[69] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R Lyu. "A Survey on Automated Log Analysis for Reliability Engineering." In: *arXiv preprint arXiv:2009.07237* (2020).

[70] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. "Deeplog: Anomaly detection and diagnosis from system logs through deep learning." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1285–1298.

[71] Pengpeng Zhou, Yang Wang, Zhenyu Li, Xin Wang, Gareth Tyson, and Gaogang Xie. "LogSayer: Log Pattern-driven Cloud Component Anomaly Diagnosis with Machine Learning." In: *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. IEEE. 2020, pp. 1–10.

[72] Haibo Mi, Huaimin Wang, Gang Yin, Hua Cai, Qi Zhou, and Tingtao Sun. "Performance problems online detection in cloud computing systems via analyzing request execution paths." In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2011, pp. 135–139.

[73] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. "Log clustering based problem identification for online service systems." In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2016, pp. 102–111.

[74] Karthik Nagaraj, Charles Killian, and Jennifer Neville. "Structured comparative analysis of systems logs to diagnose performance problems." In: *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 2012, pp. 353–366.

[75] Tong Jia, Pengfei Chen, Lin Yang, Ying Li, Fanjing Meng, and Jingmin Xu. "An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services." In: *2017 IEEE International Conference on Web Services (ICWS)*. IEEE. 2017, pp. 25–32.

[76] Pooja Aggarwal, Ajay Gupta, Prateeti Mohapatra, Seema Nagar, Atri Mandal, Qing Wang, and Amit Paradkar. "Localization of Operational Faults in Cloud Applications by Mining Causal Dependencies in Logs using Golden Signals." In: *International Conference on Service-Oriented Computing*. Springer. 2020, pp. 137–149.

[77] Byung Chul Tak, Shu Tao, Lin Yang, Chao Zhu, and Yaoping Ruan. "LOGAN: Problem diagnosis in the cloud using log-based reference models." In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2016, pp. 62–67.

[78] De-Qing Zou, Hao Qin, and Hai Jin. "Uilog: Improving log-based fault diagnosis by log analysis." In: *Journal of computer science and technology* 31.5 (2016), pp. 1038–1052.

[79] Soila P Kavulya, Scott Daniels, Kaustubh Joshi, Matti Hiltunen, Rajeev Gandhi, and Priya Narasimhan. "Draco: Statistical diagnosis of chronic problems in large distributed systems." In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE. 2012, pp. 1–12.

[80] Xiwei Xu, Liming Zhu, Ingo Weber, Len Bass, and Daniel Sun. "POD-diagnosis: Error diagnosis of sporadic operations on cloud applications." In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE. 2014, pp. 252–263.

[81] Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. "Visual, Log-Based Causal Tracing for Performance Debugging of MapReduce Systems." In: *2010 IEEE 30th International Conference on Distributed Computing Systems*. 2010, pp. 795–806.

[82] Sasho Nedelkoski, Jasmin Bogatinovski, Alexander Acker, Jorge Cardoso, and Odej Kao. "Self-attentive classification-based anomaly detection in unstructured logs." In: *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE. 2020, pp. 1196–1201.

[83] Jingmin Xu, Pengfei Chen, Lin Yang, Fanjing Meng, and Ping Wang. "LogDC: Problem diagnosis for declartively-deployed cloud applications with log." In: *2017 IEEE 14th International Conference on e-Business Engineering (ICEBE)*. IEEE. 2017, pp. 282–287.

[84] Chetan Bansal, Sundararajan Renganathan, Ashima Asudani, Olivier Midy, and Mathru Janakiraman. "DeCaf: diagnosing and triaging performance issues in large-scale cloud services." In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 2020, pp. 201–210.

[85] Jiaqi Tan, Xinghao Pan, Eugene Marinelli, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. "Kahuna: Problem diagnosis for mapreduce-based cloud computing environments." In: *2010 IEEE Network Operations and Management Symposium-NOMS 2010*. IEEE. 2010, pp. 112–119.

[86] Yue Yuan, Wenchang Shi, Bin Liang, and Bo Qin. "An Approach to Cloud Execution Failure Diagnosis Based on Exception Logs in OpenStack." In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE. 2019, pp. 124–131.

[87] Siyang Lu, Xiang Wei, Bingbing Rao, Byungchul Tak, Long Wang, and Liqiang Wang. "LADRA: Log-based abnormal task detection and root-cause analysis in big data processing with Spark." In: *Future Generation Computer Systems* 95 (2019), pp. 392–403.

[88] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. "Sherlog: error diagnosis by connecting clues from run-time logs." In: *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*. 2010, pp. 143–154.

[89] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. "Using Magpie for request extraction and workload modelling." In: *OSDI*. Vol. 4. 2004, pp. 18–18.

[90] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. "X-trace: A pervasive network tracing framework." In: *4th {USENIX} Symposium on Networked Systems Design & Implementation ({NSDI} 07)*. 2007.

[91] Avishay Traeger, Ivan Deras, and Erez Zadok. "DARC: Dynamic analysis of root causes of latency distributions." In: *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 2008, pp. 277–288.

[92] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. "Pinpoint: Problem determination in large, dynamic internet services." In: *Proceedings International Conference on Dependable Systems and Networks*. IEEE. 2002, pp. 595–604.

[93] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. "The mystery machine: End-to-end performance analysis of large-scale internet services." In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 217–231.

[94] Hiep Nguyen, Daniel J Dean, Kamal Kc, and Xiaohui Gu. "Insight: In-situ online service failure path inference in production computing infrastructures." In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 269–280.

[95] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. "Pip: Detecting the Unexpected in Distributed Systems." In: *NSDI*. Vol. 6. 2006, pp. 9–9.

[96] Alexander V Mirgorodskiy, Naoya Maruyama, and Barton P Miller. "Problem diagnosis in large-scale computing environments." In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. 2006, 88–es.

[97] Naoya Maruyama and Satoshi Matsuoka. "Model-based fault localization in large-scale computing systems." In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE. 2008, pp. 1–12.

[98] Mona Attariyan, Michael Chow, and Jason Flinn. "X-ray: Automating root-cause diagnosis of performance anomalies in production software." In: *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 2012, pp. 307–320.

[99] Daniel J Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. "Perfscope: Practical online server performance bug inference in production cloud computing infrastructures." In: *Proceedings of the ACM Symposium on Cloud Computing*. 2014, pp. 1–13.

[100] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. "Dapper, a large-scale distributed systems tracing infrastructure." In: (2010).

[101] 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." In: *IEEE Transactions on Software Engineering* (2018), pp. 1–1. DOI: 10.1109/TSE.2018.2887384.

[102] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. "Failure diagnosis using decision trees." In: *International Conference on Autonomic Computing, 2004. Proceedings.* 2004, pp. 36–43.

[103] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. "Canopy: An end-to-end performance tracing and analysis system." In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 34–50.

[104] Dan Ardelean, Amer Diwan, and Chandra Erdman. "Performance analysis of cloud applications." In: *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 2018, pp. 405–417.

[105] Zhihong Zhang, Jianfeng Zhan, Yong Li, Lei Wang, Dan Meng, and Bo Sang. "Precise request tracing and performance debugging for multi-tier services of black boxes." In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE. 2009, pp. 337–346.

[106] Haibo Mi, Huaimin Wang, Gang Yin, Hua Cai, Qi Zhou, and Tingtao Sun. "Performance problems diagnosis in cloud computing systems by mining request trace logs." In: *2012 IEEE Network Operations and Management Symposium*. 2012, pp. 893–899.

[107] Raja R Sambasivan, Alice X Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R Ganger. "Diagnosing Performance Changes by Comparing Request Flows." In: *NSDI*. Vol. 5. 2011, pp. 1–1.

[108] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. "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*. 2020, pp. 1387–1397.

[109] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. "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*. 2019, pp. 683–694.

[110] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. "MicroRank: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments." In: *Proceedings of the Web Conference 2021*. 2021, pp. 3087–3098.

[111] P. Chen, Y. Qi, P. Zheng, and D. Hou. "CauseInfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems." In: *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. 2014, pp. 1887–1895.

[112] B. Sharma, P. Jayachandran, A. Verma, and C. R. Das. "CloudPD: Problem determination and diagnosis in shared dynamic clouds." In: *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2013, pp. 1–12.

[113] JinJin Lin, Pengfei Chen, and Zibin Zheng. "Microscope: Pinpoint performance issues with causal graphs in micro-service environments." In: *International Conference on Service-Oriented Computing*. Springer. 2018, pp. 3–20.

[114] Ping Wang, Jingmin Xu, Meng Ma, Weilan Lin, Disheng Pan, Yuan Wang, and Pengfei Chen. "Cloudranger: root cause identification for cloud native systems." In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE. 2018, pp. 492–502.

[115] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. "Sieve: Actionable insights from monitored metrics in distributed sys-

tems." In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 2017, pp. 14–27.

[116] Leonardo Mariani, Cristina Monni, Mauro Pezzé, Oliviero Riganelli, and Rui Xin. "Localizing faults in cloud systems." In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2018, pp. 262–273.

[117] Myunghwan Kim, Roshan Sumbaly, and Sam Shah. "Root cause detection in a service-oriented architecture." In: *ACM SIGMETRICS Performance Evaluation Review* 41.1 (2013), pp. 93–104.

[118] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. "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*. 2019, pp. 19–33.

[119] Xu Chen, Ming Zhang, Zhuoqing Morley Mao, and Paramvir Bahl. "Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions." In: *OSDI*. Vol. 8. 2008, pp. 117–130.

[120] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. "Performance debugging for distributed systems of black boxes." In: *ACM SIGOPS Operating Systems Review* 37.5 (2003), pp. 74–89.

[121] Sandip Agarwala, Fernando Alegre, Karsten Schwan, and Jegannathan Mehalingham. "E2eprof: Automated end-to-end performance management for enterprise systems." In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE. 2007, pp. 749–758.

[122] Shriram Rajagopalan and Hani Jamjoom. "App–bisect: autonomous healing for microservice-based apps." In: *7th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. 2015.

[123] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A Maltz, and Ming Zhang. "Towards highly reliable enterprise network services via inference of multi-level dependencies." In: *ACM SIGCOMM Computer Communication Review* 37.4 (2007), pp. 13–24.

[124] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. "Detailed diagnosis in enterprise networks." In: *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*. 2009, pp. 243–254.

[125] Jianping Weng, Jessie Hui Wang, Jiahai Yang, and Yang Yang. "Root cause analysis of anomalies of multitier services in public clouds." In: *IEEE/ACM Transactions on Networking* 26.4 (2018), pp. 1646–1659.

[126] Hanzhang Wang, Phuong Nguyen, Jun Li, Selcuk Kopru, Gene Zhang, Sanjeev Katariya, and Sami Ben-Romdhane. "GRANO: Interactive graph-based root cause analysis for cloud-native distributed data platform." In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 1942–1945.

[127] Weilan Lin, Meng Ma, Disheng Pan, and Ping Wang. "FacGraph: Frequent Anomaly Correlation Graph Mining for Root Cause Diagnose in Micro-Service Architecture." In: *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*. IEEE. 2018, pp. 1–8.

[128] Ping Liu, Shenglin Zhang, Yongqian Sun, Yuan Meng, Jiahai Yang, and Dan Pei. "FluxInfer: Automatic Diagnosis of Performance Anomaly for Online Database System." In: *2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC)*. 2020, pp. 1–8.

[129] Y. Meng, S. Zhang, Y. Sun, R. Zhang, Z. Hu, Y. Zhang, C. Jia, Z. Wang, and D. Pei. "Localizing Failure Root Causes in a Microservice through Causality Inference." In: *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. 2020, pp. 1–10.

[130] Meng Ma, Weilan Lin, Disheng Pan, and Ping Wang. "Self-Adaptive Root Cause Diagnosis for Large-Scale Microservice Architecture." In: *IEEE Transactions on Services Computing* (2020).

[131] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. "Sage: practical and scalable ML-driven performance debugging in microservices." In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 135–151.

[132] Meng Ma, Jingmin Xu, Yuan Wang, Pengfei Chen, Zonghua Zhang, and Ping Wang. "Automap: Diagnose your microservice-based web applications automatically." In: *Proceedings of The Web Conference 2020*. 2020, pp. 246–258.

[133] Lingyu Zhang, Jiabao Zhao, and Min Zhang. "Root Cause Analysis of Concurrent Alarms Based on Random Walk over Anomaly Propagation Graph." In: *2020 IEEE International Conference on Networking, Sensing and Control (ICNSC)*. 2020, pp. 1–6.

[134] Meng Ma, Weilan Lin, Disheng Pan, and Ping Wang. "ServiceRank: Root Cause Identification of Anomaly in Large-Scale Microservice Architecture." In: *IEEE Transactions on Dependable and Secure Computing* (2021), pp. 1–1.

[135] Dewei Liu, Chuan He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshun Wu. "MicroHECL: High-Efficient Root Cause Localization in Large-Scale Microservice Systems." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2021, pp. 338–347.

[136] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. "Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems." In: *Proceedings of the 9th international conference on Autonomic computing*. 2012, pp. 191–200.

[137] Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B Woodard, and Hans Andersen. "Fingerprinting the datacenter: automated classification of performance crises." In: *Proceedings of the 5th European conference on Computer systems*. 2010, pp. 111–124.

[138] Dominik Scheinert, Alexander Acker, Lauritz Thamsen, Morgan K Geldenhuys, and Odej Kao. "Learning Dependencies in Distributed Cloud Applications to Identify and Localize Anomalies." In: *arXiv preprint arXiv:2103.05245* (2021).

[139] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Deepal Jayasinghe, Toshihiro Shimizu, Masazumi Matsubara, Motoyuki Kawaba, and Calton Pu. "Detecting transient bottlenecks in n-tier applications through fine-grained analysis." In: *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE. 2013, pp. 31–40.

[140] Tao Wang, Wenbo Zhang, Jun Wei, and Hua Zhong. "Fault detection for cloud computing systems with correlation analysis." In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE. 2015, pp. 652–658.

[141] Tao Wang, Wenbo Zhang, Chunyang Ye, Jun Wei, Hua Zhong, and Tao Huang. "FD4C: Automatic Fault Diagnosis Framework for Web Applications in Cloud Computing." In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 46.1 (2016), pp. 61–75.

[142] Ira Cohen, Jeffrey S Chase, Moises Goldszmidt, Terence Kelly, and Julie Symons. "Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control." In: *OSDI*. Vol. 4. 2004, pp. 16–16.

[143] Manoj K Agarwal and Venkateswara R Madduri. "Correlating failures with asynchronous changes for root cause analysis in enterprise environments." In: *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2010, pp. 517–526.

[144] Songyun Duan, Shivnath Babu, and Kamesh Munagala. "Fa: A system for automating failure diagnosis." In: *2009 IEEE 25th International Conference on Data Engineering*. IEEE. 2009, pp. 1012–1023.

[145] Chen Luo, Jian-Guang Lou, Qingwei Lin, Qiang Fu, Rui Ding, Dongmei Zhang, and Zhe Wang. "Correlating events with time series for incident diagnosis." In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2014, pp. 1583–1592.

[146] Pengcheng Xiong, Calton Pu, Xiaoyun Zhu, and Rean Griffith. "vPerfGuard: an automated model-driven framework for application performance diagnosis in consolidated cloud environments." In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. 2013, pp. 271–282.

[147] Ignacio Laguna, Subrata Mitra, Fahad A Arshad, Nawanol Theera-Ampornpunt, Zongyang Zhu, Saurabh Bagchi, Samuel P Midkiff, Mike Kistler, and Ahmed Gheith. "Automatic problem localization via multi-dimensional metric profiling." In: *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*. IEEE. 2013, pp. 121–132.

[148] Rui Zhang, Steve Moyle, Steve McKeever, and Alan Bivens. "Performance problem localization in self-healing, service-oriented systems using bayesian networks." In: *Proceedings of the 2007 ACM symposium on Applied computing*. 2007, pp. 104–109.

[149] Hiep Nguyen, Zhiming Shen, Yongmin Tan, and Xiaohui Gu. "FChain: Toward black-box online fault localization for cloud systems." In: *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE. 2013, pp. 21–30.

[150] Hiep Nguyen, Yongmin Tan, and Xiaohui Gu. "Pal: P ropagation-aware a nomaly l ocalization for cloud hosted distributed applications." In: *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*. 2011, pp. 1–8.

[151] Daniel J. Dean, Hiep Nguyen, Peipei Wang, Xiaohui Gu, Anca Sailer, and Andrzej Kochut. "PerfCompass: Online Performance Anomaly Fault Localization and Inference in Infrastructure-as-a-Service Clouds." In: *IEEE Transactions on Parallel and Distributed Systems* 27.6 (2016), pp. 1742–1755.

[152] Hui Kang, Haifeng Chen, and Guofei Jiang. "PeerWatch: a fault detection and diagnosis tool for virtualized consolidation systems." In: *Proceedings of the 7th international conference on Autonomic computing*. 2010, pp. 119–128.

[153] Cuong Pham, Long Wang, Byung Chul Tak, Salman Baset, Chunqiang Tang, Zbigniew Kalbarczyk, and Ravishankar K Iyer. "Failure diagnosis for distributed systems using targeted fault injection." In: *IEEE Transactions on Parallel and Distributed Systems* 28.2 (2016), pp. 503–516.

[154] Lingzhi Wang, Nengwen Zhao, Junjie Chen, Pinnong Li, Wenchi Zhang, and Kaixin Sui. "Root-cause metric location for microservice systems via log anomaly detection." In: *2020 IEEE International Conference on Web Services (ICWS)*. IEEE. 2020, pp. 142–150.

[155] Álvaro Brandón, Marc Solé, Alberto Huélamo, David Solans, María S Pérez, and Victor Muntés-Mulero. "Graph-based root cause analysis for service-oriented and microservice architectures." In: *Journal of Systems and Software* 159 (2020), p. 110432. ISSN: 0164-1212.

[156] Patricia Takako Endo, Guto Leoni Santos, Daniel Rosendo, Demis Moacir Gomes, André Moreira, Judith Kelner, Djamel Sadok, Glauco Estácio Gonçalves, and Mozhgan Mahloo. "Minimizing and Managing Cloud Failures." In: *Computer* 50.11 (2017), pp. 86–90.

[157] Min Fu, Len Bass, and Anna Liu. "Towards a taxonomy of cloud recovery strategies." In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE. 2014, pp. 696–701.

[158] Carlos Colman-Meixner, Chris Develder, Massimo Tornatore, and Biswanath Mukherjee. "A Survey on Resiliency Techniques in Cloud Computing Infrastructures and Applications." In: *IEEE Communications Surveys Tutorials* 18.3 (2016), pp. 2244–2281.

[159] P. Garraghan, R. Yang, Z. Wen, A. Romanovsky, J. Xu, R. Buyya, and R. Ranjan. "Emergent Failures: Rethinking Cloud Reliability at Scale." In: *IEEE Cloud Computing* 5.5 (2018), pp. 12–21.

[160] I. Brandic. "Towards Self-Manageable Cloud Services." In: *2009 33rd Annual IEEE International Computer Software and Applications Conference*. Vol. 2. 2009, pp. 128–133.

[161] Chris Schneider, Adam Barker, and Simon Dobson. "A survey of self-healing systems frameworks." In: *Software: Practice and Experience* 45.10 (2015), pp. 1375–1398.

[162] Harald Psaier and Schahram Dustdar. "A survey on self-healing systems: approaches and systems." In: *Computing* 91.1 (2011), pp. 43–73.

[163] Debanjan Ghosh, Raj Sharman, H Raghav Rao, and Shambhu Upadhyaya. "Self-healing systems—survey and synthesis." In: *Decision support systems* 42.4 (2007), pp. 2164–2185.

[164] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. "Microreboot–a technique for cheap recovery." In: *arXiv preprint cs/0406005* (2004).

[165] Jeffrey O Kephart. "Research challenges of autonomic computing." In: *Proceedings of the 27th international conference on Software engineering*. 2005, pp. 15–22.

[166] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, et al. *Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies*. Tech. rep. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, 2002.

[167] R. Koo and S. Toueg. "Checkpointing and Rollback-Recovery for Distributed Systems." In: *IEEE Transactions on Software Engineering* SE-13.1 (1987), pp. 23–31.

[168] V. Nallur and R. Bahsoon. "A decentralized self-adaptation mechanism for service-based applications in the cloud." In: *IEEE Transactions on Software Engineering* 39.5 (2013), pp. 591–612.

[169] J. P. Magalhães and L. M. Silva. "A Framework for Self-Healing and Self-Adaptation of Cloud-Hosted Web-Based Applications." In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. Vol. 1. 2013, pp. 555–564.

[170] Muli Ben-Yehuda, David Breitgand, Michael Factor, Hillel Kolodner, Valentin Kravtsov, and Dan Pelleg. "NAP: a building block for remediating performance bottlenecks via black box network analysis." In: *Proceedings of the 6th international conference on Autonomic computing*. 2009, pp. 179–188.

[171] David Garlan and Bradley Schmerl. "Model-based adaptation for self-healing systems." In: *Proceedings of the first workshop on Self-healing systems*. 2002, pp. 27–32.

[172] F. Díaz-Sánchez, S. Al Zahr, and M. Gagnaire. "An Exact Placement Approach for Optimizing Cost and Recovery Time under Faulty Multi-cloud Environments." In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. Vol. 2. 2013, pp. 138–143.

[173] Ashima Garg and Sachin Bagga. "An autonomic approach for fault tolerance using scaling, replication and monitoring in cloud computing." In: *2015 IEEE 3rd International Conference on MOOCs, Innovation and Technology in Education (MITE)*. 2015, pp. 129–134.

[174] Areeg Samir and Claus Pahl. "Autoscaling recovery actions for container-based clusters." In: *Concurrency and Computation: Practice and Experience* (2020), e5955.

[175] Andres J Ramirez, David B Knoester, Betty HC Cheng, and Philip K McKinley. "Applying genetic algorithms to decision making in autonomic computing systems." In: *Proceedings of the 6th international conference on Autonomic computing*. 2009, pp. 97–106.

[176] Haryadi S Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M Hellerstein, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. "FATE and DESTINI: A framework for cloud recovery testing." In: *Proceedings of NSDI'11: 8th USENIX Symposium on Networked Systems Design and Implementation*. 2011, p. 239.

[177] Yuanshun Dai, Yanping Xiang, and Gewei Zhang. "Self-healing and hybrid diagnosis in cloud computing." In: *IEEE International Conference on Cloud Computing*. Springer. 2009, pp. 45–56.

[178] Ahmad Mosallanejad, Rodziah Atan, Masrah Azmi Murad, and Rusli Abdullah. "A hierarchical self-healing SLA for cloud computing." In: *International Journal of Digital Information and Wireless Communications (IJDIWC)* 4.1 (2014), pp. 43–52.

[179] Emad Albassam, Jason Porter, Hassan Gomaa, and Daniel A. Menascé. "DARE: A Distributed Adaptation and Failure Recovery Framework for Software Systems." In: *2017 IEEE International Conference on Autonomic Computing (ICAC)*. 2017, pp. 203–208.

[180] Kaustubh R Joshi, Matti A Hiltunen, William H Sanders, and Richard D Schlichting. "Automatic model-driven recovery in distributed systems." In: *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. IEEE. 2005, pp. 25–36.

[181] Michael L Littman, Nishkam Ravi, Eitan Fenson, and Rich Howard. "Reinforcement learning for autonomic network repair." In: *International Conference on Autonomic Computing, 2004.* IEEE. 2004, pp. 284–285.

[182] Qijun Zhu and Chun Yuan. "A reinforcement learning approach to automatic error recovery." In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE Computer Society. 2007, pp. 729–738.

[183] Harrison Mfula and Jukka K Nurminen. "Self-healing cloud services in private multi-clouds." In: *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2018, pp. 165–170.

[184] Guoqiang Li, Lejian Liao, Dandan Song, Jingang Wang, Fuzhen Sun, and Guangcheng Liang. "A self-healing framework for qos-aware web service composition via case-based reasoning." In: *Asia-Pacific Web Conference*. Springer. 2013, pp. 654–661.

[185] Nadia Tabassum, Muhammad Saleem Khan, Sagheer Abbas, Tahir Alyas, Atifa Athar, and Muhammad Adnan Khan. "Intelligent reliability management in hyper-convergence cloud infrastructure using fuzzy inference system." In: *EAI Endorsed Trans. Scalable Information Systems* 6 (2018), e1.

[186] Afef Mdhaffar, Riadh Ben Halima, Mohamed Jmaiel, and Bernd Freisleben. "CEP4Cloud: Complex Event Processing for Self-Healing Clouds." In: *2014 IEEE 23rd International WETICE Conference*. 2014, pp. 62–67.

[187] David Garlan, S-W Cheng, A-C Huang, Bradley Schmerl, and Peter Steenkiste. "Rainbow: Architecture-based self-adaptation with reusable infrastructure." In: *Computer* 37.10 (2004), pp. 46–54.

[188] Stefania Montani and Cosimo Anglano. "Case-based reasoning for autonomous service failure diagnosis and remediation in software systems." In: *European Conference on Case-Based Reasoning*. Springer. 2006, pp. 489–503.

[189] Saadia Nasir, Maria Taimoor, Hina Gul, Amina Ali, and Malik Jahan Khan. "Optimization of Decision Making in CBR Based Self-Healing Systems." In: *2012 10th International Conference on Frontiers of Information Technology*. 2012, pp. 68–72.

[190] David McSherry, David Bustard, et al. "Conversational case-based reasoning in self-healing and recovery." In: *European Conference on Case-Based Reasoning*. Springer. 2008, pp. 340–354.

[191] J. Moysen and L. Giupponi. "A Reinforcement Learning Based Solution for Self-Healing in LTE Networks." In: *2014 IEEE 80th Vehicular Technology Conference (VTC2014-Fall)*. 2014, pp. 1–6.

[192] Hiroki Ikeuchi, Akio Watanabe, Tsutomu Hirao, Makoto Morishita, Masaaki Nishino, Yoichi Matsuo, and Keishiro Watanabe. "Recovery command generation towards automatic recovery in ICT systems by Seq2Seq learning." In: *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2020, pp. 1–6.

[193] A. Samir and C. Pahl. "Self-Adaptive Healing for Containerized Cluster Architectures with Hidden Markov Models." In: *FMEC*. 2019, pp. 68–73.

[194] Kaustubh R Joshi, William H Sanders, Matti A Hiltunen, and Richard D Schlichting. "Automatic recovery using bounded partially observable markov decision processes." In: *International Conference on Dependable Systems and Networks (DSN'06)*. IEEE. 2006, pp. 445–456.

[195] Jyoti Shetty, B Sathish Babu, and G Shobha. "Proactive cloud service assurance framework for fault remediation in cloud environment." In: *International Journal of Electrical & Computer Engineering (2088-8708)* 10.1 (2020).

[196] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. "Using architecture models for runtime adaptability." In: *IEEE software* 23.2 (2006), pp. 62–70.

[197] Min Fu, Liming Zhu, Daniel Sun, Anna Liu, Len Bass, and Qinghua Lu. "Runtime recovery actions selection for sporadic operations on public cloud." In: *Software: Practice and Experience* 47.2 (2017), pp. 223–248.

[198] S. Ossenbühl et al. "Towards Automated Incident Handling: How to Select an Appropriate Response against a Network-Based Attack?" In: *IMF*. 2015, pp. 51–67.

[199] Jyoti Shetty, B Sathish Babu, and G Shobha. "Proactive cloud service assurance framework for fault remediation in cloud environment." In: *IJECE* 10.1 (Feb. 2020), p. 987.

[200] Hans-Peter Schwefel and Imad Antonios. "Performability models for multi-server systems with high-variance repair durations." In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE. 2007, pp. 770–779.

[201] *Amazon CloudWatch*. URL: https://aws.amazon.com/cloudwatch/. (accessed: 01.09.2021).

[202] *CloudMonix*. URL: https://cloudmonix.com/aw//. (accessed: 01.09.2021).

[203] *Why Netflix, Amazon, and Apple Care About Microservices*. URL: https://blog.leanix.net/en/why-netflix-amazon-and-apple-care-about-microservices. (accessed: 01.09.2021).

[204] Marc-Oliver Pahl and François-Xavier Aubet. "All eyes on you: Distributed Multi-Dimensional IoT microservice anomaly detection." In: *2018 14th International Conference on Network and Service Management (CNSM)*. IEEE. 2018, pp. 72–80.

[205] Qingfeng Du, Tiandi Xie, and Yu He. "Anomaly detection and diagnosis for container-based microservices with performance monitoring." In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2018, pp. 560–572.

[206] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, et al. "Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks." In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2020, pp. 48–58.

[207] Jasmin Bogatinovski, Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. "Self-Supervised Anomaly Detection from Distributed Traces." In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE. 2020, pp. 342–347.

[208] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. "X-trace: A Pervasive Network Tracing Framework." In: *NSDI'07*.

[209] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. "MicroRCA: Root cause localization of performance issues in microservices." In: *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2020, pp. 1–9.

[210] Karthikeyan Shanmugam. *What's a service mesh? And why do I need one?* URL: https://containerjournal.com/topics/container-ecosystems/what-is-service-mesh-and-why-do-we-need-it/. (accessed: 01.09.2021).

[211] Anton Gulenko, Florian Schmidt, Alexander Acker, Marcel Wallschläger, Odej Kao, and Feng Liu. "Detecting Anomalous Behavior of Black-Box Services Modeled with Distance-Based Online Clustering." In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 912–915.

[212] Glen Jeh and Jennifer Widom. "Scaling personalized web search." In: *Proceedings of the 12th international conference on World Wide Web*. 2003, pp. 271–279.

[213] J. Weng, J. H. Wang, J. Yang, and Y. Yang. "Root Cause Analysis of Anomalies of Multitier Services in Public Clouds." In: *IEEE/ACM Transactions on Networking* 26.4 (2018), pp. 1646–1659.

[214] Li Wu, Jasmin Bogatinovski, Sasho Nedelkoski, Johan Tordsson, and Odej Kao. "Performance diagnosis in cloud microservices using deep learning." In: *International Conference on Service-Oriented Computing*. Springer. 2020, pp. 85–96.

[215] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. "Causal Inference Techniques for Microservice Performance Diagnosis: Evaluation and Guiding Recommendations." In: *ACSOS 2021-2nd IEEE International Conference on Autonomic Computing and Self-Organizing Systems*. 2021.

[216] Markus Kalisch and Peter Bühlmann. "Estimating high-dimensional directed acyclic graphs with the PC-algorithm." In: *Journal of Machine Learning Research* 8 (2007), pp. 613–636.

[217] Arthur Gretton, Ralf Herbrich, Alexander Smola, Olivier Bousquet, and Bernhard Schölkopf. "Kernel methods for measuring independence." In: *Journal of Machine Learning Research* 6.Dec (2005), pp. 2075–2129.

[218] Peter Bühlmann, Jonas Peters, Jan Ernest, et al. "CAM: Causal additive models, high-dimensional order search and penalized regression." In: *The Annals of Statistics* 42.6 (2014), pp. 2526–2556.

[219] *Granger causality tests*. URL: https://www.statsmodels.org/stable/generated/statsmodels.tsa.stattools.grangercausalitytests.html. (accessed: 01.09.2021).

[220] Diviyan Kalainathan, Olivier Goudet, and Ritik Dutta. "Causal Discovery Toolbox: Uncovering causal relationships in Python." In: *Journal of Machine Learning Research* 21.37 (2020), pp. 1–5.

[221] *LiNGAM*. URL: https://github.com/cdt15/lingam. (accessed: 01.09.2021).

[222] Li Wu, Johan Tordsson, Jasmin Bogatinovski, Erik Elmroth, and Odej Kao. "MicroDiag: Fine-grained Performance Diagnosis for Microservice Systems." In: *ICSE21 Workshop on Cloud Intelligence*. 2021.

[223] Charles Loboz. "Cloud resource usage—heavy tailed distributions invalidating traditional capacity planning models." In: *Journal of grid computing* 10.1 (2012), pp. 85–108.

[224] Shohei Shimizu et al. "DirectLiNGAM: A direct method for learning a linear non-Gaussian structural equation model." In: *The Journal of Machine Learning Research* 12 (2011), pp. 1225–1248.

[225] Stefan Haselböck et al. "Decision Guidance Models for Microservices: Service Discovery and Fault Tolerance." In: 2017. ISBN: 9781450348430.

[226] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Josef Spillner, and Thomas Michael Bohnert. "Self-managing cloud-native applications: Design, implementation, and experience." In: *Future Generation Computer Systems* 72 (2017), pp. 165–179.

[227] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K Reiter, and Vyas Sekar. "Gremlin: Systematic Resilience Testing of Microservices." In: *ICDCS*. 2016, pp. 57–66.

[228] Li Wu, Johan Tordsson, Alexander Acker, and Odej Kao. "MicroRAS: Automatic Recovery in the Absence of Historical Failure Data for Microservice Systems." In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. 2020, pp. 227–236.

[229] S. Huang et al. "Differentiated Failure Remediation with Action Selection for Resilient Computing." In: *PRDC*. 2015, pp. 199–208.

[230] M. Pahl and F. Aubet. "All Eyes on You: Distributed Multi-Dimensional IoT Microservice Anomaly Detection." In: *2018 14th International Conference on Network and Service Management (CNSM)*. 2018, pp. 72–80.

[231] A. Samir and C. Pahl. "DLA: Detecting and Localizing Anomalies in Containerized Microservice Architectures Using Markov Models." In: *2019 7th International Conference on Future Internet of Things and Cloud (FiCloud)*. 2019, pp. 205–213.

[232] Stefan Frey, Claudia Lüthje, Christoph Reich, and Nathan Clarke. "Cloud QoS Scaling by Fuzzy Logic." In: *2014 IEEE International Conference on Cloud Engineering*. 2014, pp. 343–348.

[233] Hamid Arabnejad, Claus Pahl, Pooyan Jamshidi, and Giovani Estrada. "A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling." In: *CCGRID*. 2017, pp. 64–73.