# Towards Reliable Cloud Microservices with Intelligent Operations

**YANG, Tianyi**

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
September 2022

## Thesis Assessment Committee

Professor LEE Ho Man (Chair)

Professor LYU Rung Tsong Michael (Thesis Supervisor)

Professor LEE Pak Ching (Committee Member)

Professor CAO Jiannong (External Examiner)

Abstract of thesis entitled:

Towards Reliable Cloud Microservices with Intelligent Operations

Submitted by YANG, Tianyi

for the degree of Doctor of Philosophy

at The Chinese University of Hong Kong in September 2022

The reliability of cloud services is crucial to both customers and cloud providers. However, the increasing scale and complexity of modern cloud systems pose great challenges for service reliability engineering. Specifically, since online cloud services are often composed of a complex hierarchy of small-grained microservices, unplanned service failures may cause severe cascading impacts, deteriorating customer satisfaction. The massive amount of logs and alerts also obstructs engineers' prompt intervention upon the occurrence of cloud anomalies. In this thesis, I propose intelligent operations based on microservice runtime data including metrics, traces, logs, and alerts. I focus on three closely-related tasks for reliable microservices, i.e., predicting the intensity of microservice dependencies, testing microservice systems, and improving the quality of alerts and logs.

Firstly, I study the outages and the procedure for failure diagnosis in two cloud providers to motivate the definition of the intensity of dependency. I define the intensity of dependency between two services as how much the status of the callee service influences the caller service. Then I propose AID, the first

approach to predict the Aggregated Intensity of Dependencies between cloud services. The experimental results show that AID can efficiently and accurately predict the intensity of dependencies. I further demonstrate the usefulness of our method in a leading public cloud provider.

Secondly, for the resilience testing of microservice systems, I identify the scalability and adaptivity issues of current human-dependent practice for resilience testing. Then I empirically compare the manifestations of common failures in resilient and unresilient microservice systems. The empirical study demonstrates the feasibility of self-adaptive resilience testing. Observing that the resilience of a service is related to the degradation propagation from system performance metrics to business metrics, I propose AVERT, the first framework for self-AdaptiVE Resilience Testing that can automatically index the resilience of a microservice system to different failures. AVERT measures the degradation propagation from system performance metrics to business metrics. The higher the propagation, the lower the resilience. The evaluation of two open-source benchmark microservice systems indicates that AVERT can effectively and efficiently test the resilience of microservice systems.

Thirdly, I empirically study how to improve the quality of alerts and logs. For alerts, I observe on-call engineers being hindered from quickly locating and fixing faulty cloud services because of the vast existence of misleading, non-informative, non-actionable alerts. I call the ineffectiveness of alerts "anti-patterns of alerts". I conduct the first empirical study on the practices of mitigating anti-patterns of alerts in an industrial cloud system. I summarized six anti-patterns of alerts, four current reactions to mitigate the anti-patterns of alerts, as well as

the general preventative guidelines for the configuration of alert strategy. For logs, I survey 33 papers in the last 23 years across a variety of topics in the practice of logging, including *where-to-log*, *what-to-log*, and *how-to-log*. Based on our empirical study, I summarize the best practices of logging and propose bridging the gap between manual alert strategies and cloud service upgrades by automatically evaluating the Quality of Alerts (QoA).

In summary, this thesis targets to assist software engineers with intelligent operations to improve reliability in cloud microservices. Comprehensive experiments on industrial and public datasets confirm the effectiveness and efficiency of our proposed methods.

論文題目：面嚮微服務可靠性的智能運維
作者　　：楊天益
學校　　：香港中文大學
學系　　：計算機科學與工程學系
修讀學位：哲學博士
摘要　　：

雲服務的可靠性對客戶和雲服務提供者都至關重要。然而，現代雲系統日益增加的規模和複雜性對雲服務的可靠性工程增加了巨大挑戰。具體來説，由於雲服務通常由多層次的小細微性微服務組成，計畫外的故障可能會造成嚴重的級聯影響，從而降低客戶滿意度。大量的日誌和警報也阻礙了工程師在異常發生時的及時干預。在本文中，我提出了基於微服務運行時資料（包括監控指標、調用鏈、日誌和警報）的智慧化系統運維。我專注於保證微服務可靠性的三項密切相關的任務，即預測微服務之間的依賴強度、測試微服務系統、以及提高警報和日誌的品質。

第一，我調研了兩個雲服務提供者的故障記錄。基於此調研，我提出了聚合依賴強度的定義。我將兩個服務之間的聚合依賴強度定義為被調用方服務的狀態對調用方服務的影響程度。然後，我提出了AID，這是第一種預測雲服務之間的聚合依賴強度的方法。在類比的和真實的雲系統上的實驗結果表明，AID可以高效、準確地預測依賴強度。我在領先的公共雲服務提供者的系統中進一步展示了我所提出的方法的實用性。

第二，對於微服務系統的韌性測試，我指出了當前依賴人工審視的韌性測試實踐的擴展性和適應性問題。然後，我

實驗對比了韌性高的和韌性低的微服務系統在常見故障下的表現。我觀察到微服務的韌性與監控指標從系統性能指標降級傳播到業務指標降級的程度有關，這表明了自適應的韌性測試的可行性。我提出了AVERT，第一個自適應的韌性測試框架。AVERT可以自動索引微服務系統對不同故障的韌性。AVERT測量從系統性能指標降級到業務指標降級的傳播程度。傳播程度越高，韌性越低。對兩個開源基準微服務系統的評估表明，AVERT可以有效地測試微服務系統的韌性。

第三，我實證研究了提高告警和日誌品質的方法。對於告警品質，我觀察到，由於大量存在誤導性、非資訊性、不可操作的警報，使運維工程師無法快速定位和修復故障雲服務。我將告警的無效性稱為"告警的反模式"。我首次對應對雲系統鐘告警的反模式的工業實踐進行了實證研究。我研究了華為的告警策略和告警處理流程。我總結了六種告警的反模式，四種應對告警反模式的方法，以及告警策略配置的一般預防指南。對於日誌品質，我調研了過去23年中的33篇相關論文。這些論文涉及日誌中的各種主題，包括在哪裡記錄日誌、記錄什麼日誌內容以及記錄日誌的方法。基於我的實證研究，我總結了記錄日誌的最佳實踐，並提出通過自動評估警報品質的方法來提高告警和日誌的品質。

總之，本文旨在幫助軟體工程師進行智慧操作，以提高雲微服務的可靠性。在工業和公共數據集上的實驗證實了我提出的方法的有效性和高效性。

# Acknowledgement

First and foremost, I would like to thank my supervisor, Prof. Michael R. Lyu, for his excellent supervision during my Ph.D. study at CUHK. From choosing the research topic to technical writing, his inspiring guidance and patience help me conduct challenging research work. During the long Ph.D. study period, I have learned so much from his knowledge and attitude in doing research and being a nice person.

I am grateful to my thesis assessment committee members, Prof. LEE Ho Man Jimmy and Prof. Lee Pak Ching Patrick, for their constructive comments and valuable suggestions for this thesis and all my term presentations. Many thanks to Prof. CAO Jiannong from The Hong Kong Polytechnic University, who kindly serves as the external examiner for this thesis.

Thank Prof. Cuiyun Gao from Harbin Institute of Technology (Shenzhen), Prof. Yuxin Su from Sun Yat-Sen University, Dr. Yu Kang and Dr. Si Qin from Microsoft Research Asia, and Mr. Guangsheng Wei from Huawei Cloud for their insightful discussions about the research topics included in this thesis.

Thank my fantastic collaborators, Jiacheng Shen and Baitong Li for their precious suggestions and collaboration during my Ph.D. study.

I am also thankful to my excellent group fellows, Jian Li, Pengpeng Liu, Xiaotian Yu, Wang Chen, Yue Wang, Han Shao,

Shilin He, Haoli Bai, Wenxiang Jiao, Yifan Gao, Jingjing Li, Weibin Wu, Xiaoxue Ren, Zhuangbin Chen, Wenchao Gu, Jen-Tse Huang, Yun Peng, Jianping Zhang, Jinyang Liu, Yintong Huo, Wenxuan Wang, Yichen Li, Shuqing Li, Wenwei Gu, Yizhan Huang, and Renyi Zhong.

Lastly, I want to thank my family. Their deep love and constant support are the driving force for me to pursue my doctorate.

To my family.

# Contents

**Abstract** **i**

**Acknowledgement** **vi**

**1 Introduction** **1**

1.1 Overview . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Thesis Contributions . . . . . . . . . . . . . . . . . 8

1.3 Thesis Organization . . . . . . . . . . . . . . . . . 10

**2 Research Foundations** **13**

2.1 Research Background . . . . . . . . . . . . . . . . 13

2.1.1 Microservice Systems . . . . . . . . . . . . . 13

2.1.2 Monitoring Microservice Systems . . . . . 15

2.2 Research Problems . . . . . . . . . . . . . . . . . 24

2.2.1 Intensity of Microservice Dependency . . . 24

2.2.2 Microservice Resilience Testing . . . . . . 26

2.2.3 Anti-patterns of Alerts . . . . . . . . . . . 28

**3 Predicting the Aggregated Intensity of Dependency** **31**

3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . 31

3.2 Motivation . . . . . . . . . . . . . . . . . . . . . . 34

3.2.1 A Survey of the Outages in AWS . . . . . 35

3.2.2 Drawbacks of Current Failure Diagnosis Methods . . . . . . . . . . . . . . . . . . . . 37
3.2.3 Intensity of Service Dependency . . . . . . 39
3.3 Problem Definition . . . . . . . . . . . . . . . . . . 41
3.4 Methodology . . . . . . . . . . . . . . . . . . . . . 41
3.4.1 Overview . . . . . . . . . . . . . . . . . . . . 42
3.4.2 Candidate Selection . . . . . . . . . . . . . . 43
3.4.3 Service Status Generation . . . . . . . . . . 44
3.4.4 Intensity Prediction . . . . . . . . . . . . . . 46
3.5 Evaluation . . . . . . . . . . . . . . . . . . . . . . . 49
3.5.1 Experimental Setup . . . . . . . . . . . . . . 49
3.5.2 RQ1: Effectiveness . . . . . . . . . . . . . . 53
3.5.3 RQ2: Impact of Different Parameter Settings . . . . . . . . . . . . . . . . . . . . . . . 54
3.5.4 RQ3: Impact of Different Similarity Measures . . . . . . . . . . . . . . . . . . . . . . . 55
3.5.5 RQ4: Efficiency . . . . . . . . . . . . . . . . 56
3.6 Use Cases . . . . . . . . . . . . . . . . . . . . . . . 57
3.6.1 Optimization of Dependencies . . . . . . . . 58
3.6.2 Mitigation of Cascading Failures . . . . . . 58
3.7 Discussion . . . . . . . . . . . . . . . . . . . . . . . 59
3.7.1 Practical Usage and Perceived Limitations 59
3.7.2 Threat to Validity . . . . . . . . . . . . . . . 60
3.8 Related Work . . . . . . . . . . . . . . . . . . . . . 61

**4 Self-adaptive Microservice Resilience Testing 66**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . 66
4.2 Motivation . . . . . . . . . . . . . . . . . . . . . . . 70
4.2.1 RQ1: Issues of Current Practice . . . . . . 71
4.2.2 RQ2: Failures and Their Impact . . . . . . 73

4.3 Methodology . . . . . . . . . . . . . . . . . . . . . . 76
4.3.1 Design Objectives . . . . . . . . . . . . . . . . 77
4.3.2 Overview . . . . . . . . . . . . . . . . . . . . . 78
4.3.3 Failure Execution . . . . . . . . . . . . . . . . 79
4.3.4 Degradation-based Metric Lattice Search . 80
4.3.5 Resilience Indexing . . . . . . . . . . . . . . . 85
4.4 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . 85
4.4.1 Experiment Settings . . . . . . . . . . . . . . 86
4.4.2 RQ3: Effectiveness . . . . . . . . . . . . . . . 90
4.4.3 RQ4: Ablation Study . . . . . . . . . . . . . . 91
4.4.4 RQ5: Efficiency . . . . . . . . . . . . . . . . . 92
4.5 Discussion . . . . . . . . . . . . . . . . . . . . . . . . 92
4.5.1 Threats to Validity . . . . . . . . . . . . . . . 92
4.6 Related Work . . . . . . . . . . . . . . . . . . . . . . 94

**5 Empirical Study on Alerting and Logging 97**

5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . 97
5.2 The Anti-patterns of Alerts . . . . . . . . . . . . . . 102
5.2.1 RQ1: Anti-patterns in Alerts . . . . . . . . 103
5.2.2 RQ2: Standard Alert Processing Procedure 110
5.2.3 RQ3: Reactions to Anti-patterns . . . . . 111
5.2.4 RQ4: Avoidance of Anti-patterns . . . . . 113
5.3 The Practice of Logging . . . . . . . . . . . . . . . . 114
5.3.1 RQ5: Logging Mechanism and Libraries . 115
5.3.2 RQ6: Challenges for Logging . . . . . . . . 117
5.3.3 RQ7: Logging Approaches . . . . . . . . . . 119
5.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . 126
5.4.1 Detecting Anti-patterns of Alerts . . . . . 126
5.4.2 Best Practices for Logging . . . . . . . . . . 128
5.5 Related Work . . . . . . . . . . . . . . . . . . . . . . 129

**6 Conclusion and Future Work** **132**

6.1 Conclusion . . . . . . . . . . . . . . . . . . . . . . . 132

6.2 Future Directions . . . . . . . . . . . . . . . . . . . . 134

6.2.1 Trace Compression based on Service Topology . . . . . . . . . . . . . . . . . . . . . . . . 135

6.2.2 Analysis-Oriented Logging . . . . . . . . . . 135

6.2.3 Automated Generation of Logging Statements . . . . . . . . . . . . . . . . . . . . . . 136

**7 Publications during Ph.D. Study** **137**

**Bibliography** **140**

# List of Figures


1.1 An illustration of the microservice architecture. . 2
1.2 Overview of the research in this thesis. . . . . . . 4
1.3 Categorization of the research in this thesis. . . . 5

2.1 The components of a microservice monitoring system. . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.2 A trace with six spans. . . . . . . . . . . . . . . . . 16
2.3 An example of logging statements by SLF4J and the generated logs. . . . . . . . . . . . . . . . . . . . 20
2.4 The significance of alerts for cloud reliability. . . . 21
2.5 The monitoring metrics during the normal period (the green area) and the failure injection period (the red area). . . . . . . . . . . . . . . . . . . . . 27

3.1 The statuses of service `A`, `B` and `C`. `A` invokes `B` and `C` but `B` has a greater effect on `A`. . . . . . . . . . 39
3.2 The overall workflow of AID. . . . . . . . . . . . . . 41
3.3 Prediction loss under different bin size $\tau$. . . . . . 54
3.4 The use case of AID. . . . . . . . . . . . . . . . . . 57

4.1 Overall framework of AVERT. . . . . . . . . . . . . 77

4.2 An example metric lattice constructed from $\mathcal{M} = \{m_1, \cdots, m_4\}$. I set number of monitoring metrics as a small value, 4, for a clear illustration. The path of all solid red edges forms a ranked list. 82

5.1 A survey about the current practice of mitigating the anti-patterns of alerts. . . . . . . . . . . . . . . 103
5.2 Repeating alerts in an alert storm. . . . . . . . . 108
5.3 An example Standard Operation Procedure. . . . 110
5.4 Answers to Q1 "Overall Helpfulness" regarding OCEs' working experience. . . . . . . . . . . . . . 110
5.5 An example of logging statements by SLF4J and the generated logs. . . . . . . . . . . . . . . . . . 116
5.6 Incorporating human knowledge and machine learning to detect anti-patterns of alerts. . . . . . . . . 127

# List of Tables


2.1 The attributes of a span. . . . . . . . . . . . . . . . 17
2.2 A span generated by the train-ticket benchmark. . 18
2.3 Sample reliability alerts in a cloud system. The names of microservices are omitted due to confidentiality. . . . . . . . . . . . . . . . . . . . . . . . 22

3.1 Summary of AWS outages related to service dependency. . . . . . . . . . . . . . . . . . . . . . . . 35
3.2 Dataset statistics. . . . . . . . . . . . . . . . . . . . . 50
3.3 Performance Comparison of Different Methods on Two Datasets . . . . . . . . . . . . . . . . . . . . . 53
3.4 The impact of different similarity measures . . . . 55

4.1 Dataset Statistics . . . . . . . . . . . . . . . . . . . . 88
4.2 Performance Comparison of AVERT on Two Datasets 90
4.3 Ablation Study of AVERT on Two Datasets . . . 91
4.4 Typical faults and the corresponding degradation with and without the resilience mechanisms mentioned in § 4.2.2 . . . . . . . . . . . . . . . . . . . 96

5.1 The Terminology Adopted in This Chapter. . . . 99
5.2 Summary of logging approaches. . . . . . . . . . . 120

# Chapter 1

# Introduction

## 1.1 Overview

Modern online services are moving towards the microservice architecture [108], where a monolithic online service is split into fine-grained, independently-managed microservices which collectively serve user requests. A *microservice* is a small independent program that communicate over well-defined APIs. Multiple microservices serve users' requests as a whole. Under this architecture, microservice runtime management frameworks like Kubernetes will be responsible for managing the life cycles of microservices. Figure 1.1 illustrates a microservice system comprised of three cloud services, each of which is composed of multiple microservices. Developers can focus on the application logic instead of the bothering tasks of resource management and failure recovery. All the component microservices of an online service and the additional components for the orchestration of microservices (e.g., load balancer, message queues, and service registries) constitute a *microservice system*.

The microservice architecture exhibits three prominent attributes. First, a microservice system is highly decoupled [7, 9]. Each

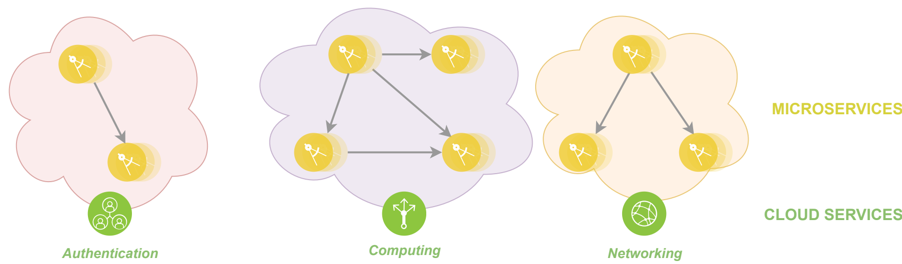



Figure 1.1: An illustration of the microservice architecture.

microservice in a microservices system can be developed, deployed, operated, and scaled without affecting the functioning of other services. Microservices do not need to share any of their code or implementation with other microservices. The individual microservices communicate with each other through well-defined APIs. Second, the microservice architecture is highly dynamic [7]. New features and updates are delivered continuously and frequently [55]. Last, microservices are specialized [7]. Different from other existing distributed systems (e.g., Hadoop, Spark, and Blockchain), each microservice is designed for a set of capabilities and focuses on serving a specific problem. If developers contribute more code to a service over time and the service becomes complex, it can be broken into smaller services. As a result, the microservice failures are usually cascaded due to the multi-layer deployment and inter-service dependencies architecture [91, 149, 159]. Such three attributes lead to the challenges that are specific to the microservice architecture.

Challenges to the reliability of cloud microservices originate from both internal and external aspects of microservices. Internal aspects are the flaws of the microservice itself, e.g., software bugs and software resilience issues, etc. On the one hand, a software bug is an error, flaw or fault in the design, development,

or operation of software. Bugs lead to erroneous behaviors of the microservice. On the other hand, service resilience [127], i.e., the ability to maintain performance at an acceptable level and recover the service back to normal under service failures, is essentially a desired ability of microservices. Resilience issues affect the availability of the microservice, which is harmful for cloud providers' revenue. I exemplify a resilience issue with Figure 2.5, which illustrates the request throughput of a service during the normal (green) and the failure (red) period. Intuitively, the resilience of the service is low because the failure causes service degradation, reflected by the throughput decrement. Since faults and failures are unavoidable [68, 91], test engineers conduct resilience tests on microservices to ensure service reliability. All new or updated microservices need to pass a lot of resilience tests before their deployments. Specifically, test engineers purposefully inject failures into the system to discover flaws [65,106]. Improvements on architectural design are then adopted according to the test results. External aspects indicate the threats from outside a microservice, such as cascading failures and low-quality logs and alerts. Cascading failures that result in service degradation are ubiquitous in a microservice system. Although microservice management frameworks provide automatic mechanisms for failure recovery, unplanned service failures may still cause severe cascading effects. For example, failures of critical services that provide basic request routing functions will impact the invocation of cloud services, slow down request processing, and deteriorate customer satisfaction. Therefore, evaluating the impact of service failures rapidly and accurately is critical to the operation and maintenance of cloud services. Knowing the scope of the impact, reliability engineers can put more emphasis

on services that have greater impacts on others. Low-quality logs and alerts are due to system-level misconfigurations. As On-Call Engineers (OCEs) usually inspect the logs and alerts to locate and diagnose failures when failures occur, if the logs and alerts are low-quality or misleading, the process of manual diagnosis will be hindered.

Figure 1.2: Overview of the research in this thesis.

To tackle the challenges above, in this thesis, I resort to intelligent operations for improving the reliability of microservice systems. As illustrated in Figure 1.2, the intelligent operations are based on microservice runtime data including logs, metrics, traces, and alerts. Logs, metrics, and traces reflect the runtime status of the microservices. Specifically, traces record the status of each microservice invocation, including the return value, the duration of execution, etc. Logs are semi-structured text printed by logging statements (e.g., `logger.info()`) in the source code. Metrics are fixed-interval time series reflecting the statuses of the microservice system [39]. Alerts, in addition, are notifications sent to On-Call Engineers (OCEs), of the form defined by the alert strategy, of specific abnormal states of the cloud service or microservice. The monitoring data are collected and processed by the microservice monitoring system. When an abnormal state is detected and a pre-configured alert strategy is activated, the alerting module will send an alert to the OCEs.

In Figure 1.3, I further categorize the reliability measures discussed in this thesis into three types, i.e., proactive measures,

reactive measures, and retrospective measures. Proactive measures examine the microservice system to detect possible flaws of the system before the occurrence of a failure. Reactive measures assist OCEs to reduce the impact of a failure during failure mitigation. Retrospective measures aim at discovering the ineffectiveness of the failure mitigation process so that engineers can make optimizations accordingly.

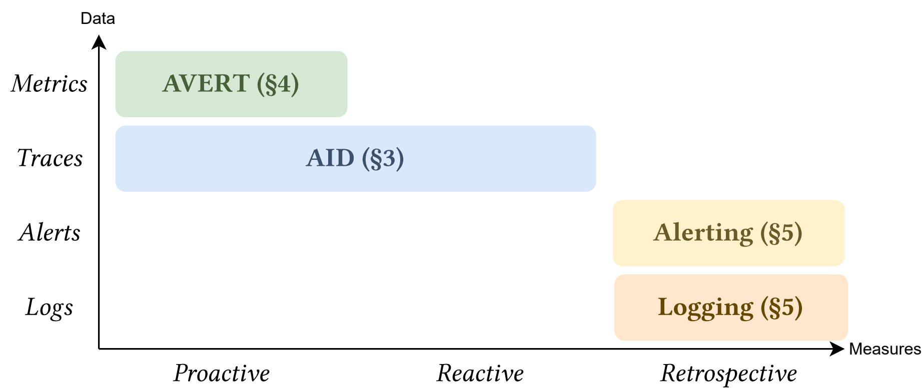



Figure 1.3: Categorization of the research in this thesis.

In this thesis, I focus on three closely-related tasks towards reliable microservices. The task details are listed as follows:

**Predicting the Intensity of Microservice Dependency**: Service invocations cause dependencies between services. Identifying whether one service depends on another (i.e., binary dependency) in online service systems can be well solved by industrial tracing frameworks like Dapper, Jaeger, and Zipkin. However, modeling the relations of services solely with binary dependencies is not precise enough. The callee microservice impact the caller microservice in different ways. Hence, the procedure of failure recovery can be sped up by skipping those unimportant services. Manual examination of different dependencies without any priority is inefficient, especially in microservice systems

where the microservice components could be highly decoupled and dynamic. Based on this observation, I argue that it will be helpful to measure the dependency as a continuous value that indicates the intensity of this dependency. Specifically, by checking microservices that are dependent on the failed microservice with large intensity values, OCEs can find the root cause of a failure with a higher probability. By recovering the services that are strongly dependent on the failed one, the whole system could be restored faster. To this end, I propose an effective and efficient approach to predict the intensity of microservice dependency based on traces.

**Resilience Testing of Microservice Systems**: The resilience of a microservice system refers to the ability to maintain the performance of services at an acceptable level and recover the service back to normal when a failure in one or more parts of the system causes the service degradation [127, 152]. Resilience testing [106] is one of the primary ways to measure the resilience of software. By purposefully introducing failures into the system, the test engineers can monitor how the microservice system performs and improve the architectural design according to the discovered flaws [65]. Automation of the resilience testing procedure is possible, but standardization of test parameters is still required. In practice, to standardize the resilience testing procedure, test engineers need to manually determine the set of rules for each failure type, which is burdensome and unscalable. This is due to microservice systems' decoupled and specialized nature. To solve this problem, I propose a self-adaptive framework for resilience testing of microservice systems.

**Improving the Quality of Alerts and Logs**: The quality issues of logs and alerts stem from the dynamic and special-

ized attribute of microservice systems. The configuration of alert strategies is empirical, which heavily depends on human expertise. There are no one-fits-all criteria for "*when to generate an alert*" and "*what attributes and descriptions an alert should have*". In practice, I observe on-call engineers being hindered from quickly locating and fixing faulty cloud services because of the vast existence of misleading, non-informative, non-actionable alerts. I call the ineffectiveness of alerts "anti-patterns of alerts". To better understand the anti-patterns of alerts and provide actionable measures to mitigate anti-patterns, in this thesis, I conduct the first empirical study on the practices of mitigating anti-patterns of alerts in an industrial cloud system. I study the alert strategies and the alert processing procedure at Huawei Cloud, a leading cloud provider. I also survey current reactions to mitigate the anti-patterns of alerts, and the general preventative guidelines for the configuration of alert strategy. Logs are also crucial for service reliability. The practice of logging has attracted attention from both academia [107, 123, 128, 146] and industry [14, 26, 47, 123, 154, 158] across a variety of application domains because logging is a fundamental step for all the subsequent log mining tasks. However, logging practice is scarcely documented or regulated by strict standard, such as the logging mechanism and APIs [14]. Essentially, logging is a subjective task that relies heavily on human expertise [47, 62, 118, 123, 132]. I review the challenges and existing solutions of the practice of logging. In the end, I propose the roadmap for improving the quality of alerts and summarize the best practices of logging for system reliability.

## 1.2 Thesis Contributions

In this thesis, I mainly focus on intelligent operations for improving the reliability of microservice systems. I build the foundation of intelligent operations on microservice runtime data including logs, metrics, traces, and alerts. Then, I focus on three closely-related tasks towards reliable microservices, as illustrated in Figure 1.3. For proactive and reactive reliability, I attribute the microservice dependency with the concept of aggregated intensity and propose an effective and efficient way to measure it. For microservice resilience testing, I identify the issues of current industrial practice and propose the first self-adaptive resilience testing framework that can automatically index the resilience of a microservice system to different failures. For improving the quality of alerts, I characterize the ineffective patterns in alerts, summarize the mitigating the ineffective patterns, and propose the criteria for evaluating and optimizing the quality of alerts. Besides, I conduct a survey across a variety of topics in logging practice and propose the best practices for logging. The contributions of this thesis are summarized as follows:

- For predicting the intensity of dependency, I conduct a comprehensive industrial survey and an empirical study to identify the inefficiency of using binary-valued dependency for failure diagnosis and failure recovery. I propose AID, the first method to quantify the intensity of dependencies between different services. The evaluation results on both simulated environment and industrial environment show the effectiveness and efficiency of the proposed method. I release a simulated dataset and an industrial dataset from a production cloud system to facilitate future studies. Additionally, Our

method have been successfully applied in a leading public cloud provider, and helped greatly reduce manual maintenance effort.

- For the resilience testing of microservice systems, I identify the scalability and adaptivity issues of current industrial practice for resilience testing. Then I conduct the first empirical study on the failures' manifestations on resilient and unresilient microservice systems. The empirical study demonstrates the feasibility for self-adaptive resilience testing. I propose AVERT, the first self-adaptive resilience testing framework that can automatically index the resilience of a microservice system to different failures. AVERT measures the degradation propagation from system performance metrics to business metrics. The higher the propagation, the lower the resilience. The evaluation on two open-source benchmark microservice systems indicates that AVERT can effectively and efficiently produce accurate test results.

- For improving the quality of alerts, I conduct the first empirical study on characterizing and mitigating anti-patterns of alerts in an industrial cloud system. I identify six anti-patterns of alerts in a production cloud system. Specifically, the six anti-patterns can be divided into two categories, namely individual anti-patterns and collective anti-patterns. Individual anti-patterns result from the ineffective patterns in one single alert strategy, including *Unclear Name or Description*, *Misleading Severity*, *Improper and Outdated Alert Strategy*, and *Transient and Toggling Alerts*. Collective anti-patterns are ineffective patterns that a bunch of alerts collectively exhibit, including *repeating* and *cascading alerts*.

I summarize the current industrial practices for mitigating the anti-patterns of alerts, including postmortem reactions to mitigate the effect of anti-patterns and the preventative guidelines to avoid the anti-patterns. The postmortem reactions include *rule-based alert blocking* and *alert aggregation*, *pattern-based alert correlation analysis*, and *emerging alert detection*. I also describe three aspects of designing preventative guidelines for alert strategies according to our experience in Huawei Cloud. Lastly, I share our thoughts on prospective directions to achieve automatic alert governance. I propose to bridge the gap between manual alert strategies and cloud service upgrades by automatically evaluating the Quality of Alerts (QoA) in terms of *indicativeness*, *impact*, and *handleability*.

- As for the practice of logging, I survey 33 papers in the last 23 years across a variety of topics in logging practice, including *where-to-log*, *what-to-log*, and *how-to-log*. The papers under exploration are mainly from top venues in three related fields: software engineering (*e.g.*, ICSE), system (*e.g.*, SOSP), and networking (*e.g.*, NSDI). Thus, the readers can obtain a deep understanding of the advantages and limitations of the SOTA approaches.

## 1.3 Thesis Organization

The remainder of this thesis is organized as follows.

- **Chapter 2**
  In this chapter, I provide a systematic review of the background knowledge about microservice systems that underpins

our approach. Firstly, I introduce the microservice architecture and sort out the monitoring data of the microservice system in Section § 2.1. Then, Section § 2.2.1 provides the problem definition of predicting the intensity of microservice dependency. Section § 2.2.2 gives a brief review of the resilience testing techniques. At last, Section § 2.2.3 describes the reliability measures for cloud services and the alerting mechanism in cloud systems., then explains why the ineffectiveness of alerts is a crucial problem.

- **Chapter 3**
  At first, Section § 3.1 provide a general introduction to this chapter. I describe our survey and empirical study on real outages that motivate our proposed method in Section § 3.2. I define the intensity prediction problem in Section § 3.3. Section § 3.4 elaborates on the proposed methodology, AID, for evaluating the intensity of dependency in detail. Section § 3.5 introduces the datasets, baselines and shows the experimental results. Successful use cases of the proposed method in a production cloud system are demonstrated in Section § 3.6. I discuss the practical usage, the perceived limitations, and the possible threats to validity in Section § 3.7. Section § 3.8 reviews existing work on service dependency analysis.

- **Chapter 4**
  I give a general introduction of this chapter in Section § 4.1. I describe our empirical study on the current industrial practice and the failures' manifestations of microservices in Section § 4.2 as the motivation that underpin our approach. Section § 3.4 elaborates on the entire workflow of AVERT in detail. Section § 4.4 introduces the datasets, baseline methods,

and discusses the experimental results. I present a discussion about the threats to validity in Section § 4.5 and review the existing research on testing the ability of tolerate possible system failures in Section § 4.6.

- **Chapter 5**
  A general introduction of this chapter in presented Section § 5.1. Section § 5.2 presents our empirical study on the anti-patterns of alerts. Particularly, I first introduce the general procedure to process an alert, then present six anti-patterns. I summarize the current postmortem measures and preventative guidelines to mitigate the anti-patterns. Section § 5.3 introduces approaches that automate or improve logging practices, including *where-to-log*, *what-to-log*, and *how-to-log*. Section § 5.4 discusses the prospective directions towards the automatic evaluation of QoA and the best practices for logging. Section § 5.5 reviews existing alert management techniques.

- **Chapter 6**
  The last chapter first summarizes this thesis in Section § 6.1. Then in Section § 6.2, I discuss some potential future directions about reliability engineering in modern cloud systems, including both microservice systems and serverless computing platforms.

---

□ **End of chapter.**

# Chapter 2

# Research Foundations

## 2.1 Research Background

### 2.1.1 Microservice Systems

Microservices are small, independent, and loosely coupled software modules that can be deployed independently [108, 115]. Modern large-scale online services, such as Netflix, Facebook, Amazon store, Alibaba Cloud, etc., are often constructed from a complex and large-scale hierarchy of small-grained microservices [9]. Nowadays, the common practice is to develop and deploy multiple microservices that collectively comprise an online service [38]. Different microservices serve different responsibilities [38, 121, 129] like user authentication, resource allocation, virtual network management, billing, etc. Additional components (e.g., API gateways, message queues, service registries, etc.) are usually employed for service decoupling and orchestration. When an external request arrives, it will be routed through the system and served by dozens of microservices that rely on one another. The microservices communicate with each other through well-defined APIs and, therefore, can be refactored and scaled independently and dynamically to adapt to incidents like

surges of requests and service failures [140]. Such an architecture is called the microservice architecture (MSA) [108].

The microservice architecture becomes increasingly popular due to its high flexibility, reusability, and scalability [11]. It enables agile development and supports polyglot programming, i.e., microservices developed under different technical stacks can work together smoothly. However, the loosely coupled nature of microservices makes it difficult for engineers to conduct system maintenance. First, the dependencies between microservices are hard to grasp. Different microservices in a large cloud system are usually developed and managed by separate teams. Each team only has access to their own services as well as services that are closely related, which means they only have a local view of the whole system [144]. Second, the failures become hard to diagnose due to the multi-layered, virtualized, and inter-dependent architecture. Microservices are typically deployed using virtualized infrastructure such as virtual machines and containers. As a result, the failure diagnosis, fault localization, and performance debugging in a large microservice system become more complex than ever [30, 51, 143]. Despite various fault tolerance mechanisms introduced by modern microservice systems, it is still possible for minor anomalies to magnify their impact and escalate into system outages. As exemplified in Section § 3.2, when a cloud service enters an anomalous state and does not return results in a timely manner, other services that depend on it will also suffer from the increased request latency. Such anomalous states can propagate through the service-calling structure and eventually affect the entire system, resulting in a degraded user experience or even a service outage. The propagated impacts of system anomalies and failures are more complex compared with

monolithic applications.

### 2.1.2 Monitoring Microservice Systems

One of the major drawbacks of the microservice architecture is the difficulty in system maintenance [165, 174]. The highly decoupled nature of the microservice architecture makes the performance debugging, failure diagnosis, and fault localization in cloud systems more complex than ever [31, 51, 160, 162].

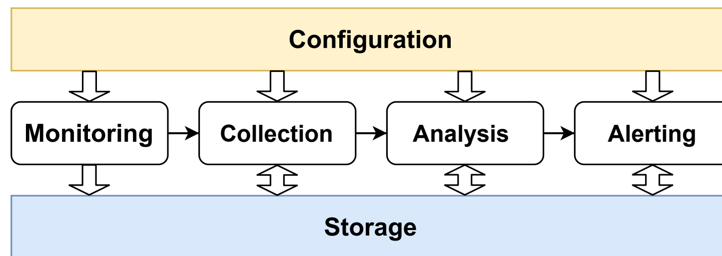



Figure 2.1: The components of a microservice monitoring system.

A common way to tackle the difficulties in system maintenance is to deploy monitoring systems. The microservice monitoring systems improve system observability [63,64,67,123,150] by tracing, logging, performance monitoring. Figure 2.1 illustrates the components of a typical microservice monitoring system. First, the monitoring modules generate monitoring data, i.e., logs, metrics, and traces. Second, the monitoring data are collected by the data collection module and processed by the analysis module. The analysis module continuously run anomaly detection on traces, logs and metrics. When an abnormal state is detected and a pre-configured alert strategy is activated, the alerting module will send an alert to the On-Call Engineers (OCEs).

### Distributed Tracing

For online service providers, it is crucial to troubleshoot and fix the failures in a timely manner because massive user applications may be affected even by a small failure [29]. Distributed tracing is a crucial technique for gaining insight and observability to cloud systems.

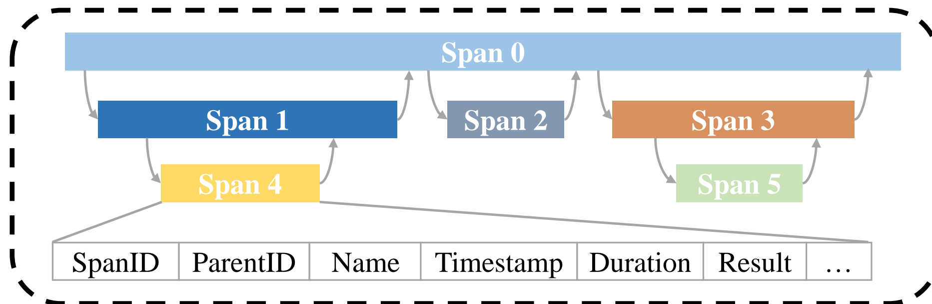



Figure 2.2: A trace with six spans.

In large-scale microservice systems, a request is usually handled by multiple chained microservice invocations. As clues to defective microservices are hidden in the intricate network, it is difficult for even knowledgeable On-Call Engineers (OCEs) to infer how a request is processed in the microservice system. Distributed tracing provides an approach to monitor the execution path of each request. For chained microservice invocations, e.g., service `A` invokes service `B`, and service `B` invokes service `C`, it is important to know the status of each microservice invocation, including the return value, the duration of execution, etc. By adding hooks to the microservices in the system, distributed tracing techniques [13, 46, 136] can record the contextual information of each service invocation. Such records are called `spans`. A span represents a logical unit of execution that is handled by a microservice in the microservice system. All the spans that serve for the same request collectively form a

directed graph of spans, as illustrated in Figure 2.2. Such a directed graph of spans generated by a request is called a `trace`. A trace represents an execution path through the microservice system. With a trace, engineers can track how the request propagates through the microservice system. Collectively analyzing the traces of the entire microservice system can help engineers obtain in-depth execution reports that could assist failure diagnosis, fault localization, and surface performance degradation in the microservice system.

Table 2.1: The attributes of a span.

| **Notation** | **Meaning** |
|---|---|
| $s_i^{id}$ | The ID of span $s_i$ |
| $s_i^{pid}$ | The ID of the parent span of $s_i$ |
| $s_i^{tid}$ | The ID of the trace that $s_i$ belongs to |
| $s_i^{name}$ | The name of service/microservice corresponding to $s_i$ |
| $s_i^{ts}$ | The time stamp of $s_i$ |
| $s_i^{d}$ | The duration of execution of $s_i$ |
| $s_i^{r}$ | The result of execution of $s_i$ |

Although the actual implementation of distributed tracing systems varies a lot, the types of information they record are similar. For clarity, I formally describe the attributes of spans as follows. Suppose I have a trace $T$ composed of spans $\{s_1, s_2, ..., s_n\}$, a span $s_i \in T$ contains the attributes[1] shown in Table 2.1.
Table 2.2 illustrates the contents of a span generated by the train-ticket benchmark [173]. It means that service `ts-preserve-service` was invoked at 04:58 on April 17, 2020. The duration of execution is 1126 $\mu s$, and the execution result is SUCCESS.

[1] Other additional contextual information [119] is omitted as I do not use them in the thesis.

| *Span ID* | e22f30bdbfd09134 |
|---|---|
| *Parent Span ID* | b42a04bf18997d5d |
| *Name* | ts-preserve-service |
| *Timestamp* ($\mu$s) | 1618589098705000 |
| *Duration* ($\mu$s) | 1126 |
| *Result* | SUCCESS |
| *Trace ID* | c0d17d481f47bdd9 |
| *Additional Logs* | ... |

Table 2.2: A span generated by the train-ticket benchmark.

**Performance Monitoring**

Performance monitoring techniques observe real-time statuses (i.e., monitoring metrics) of microservice systems [39]. The microservice architecture produces comprehensive monitoring metrics, i.e., float-valued or integer-valued time series. The types of monitoring metrics vary depending on the microservice system's architecture and implementation. Generally, monitoring metrics can be categorized into *system performance metrics* and *business metrics*.

System Performance Metrics

System performance metrics directly reflect the runtime status of the microservices and the underlying orchestration system. For example, Kubernetes[2], a popular microservice orchestration platform, employs a multi-level isolation architecture for container orchestration. Every microservice running in the Kubernetes environment executes in a container in an isolated virtual environment called "pod". Each pod resides in a "node" which can either be a virtual machine or a physical machine. Besides, Kubernetes also has many components for network management, proxy, and task scheduling. As any fail-

[2] http://kubernetes.io/

ure of these components may result in the degradation of service, all the pods, nodes, and other components will be monitored, producing various system performance metrics, e.g., CPU usage, memory usage, network throughput, network transmit (tx) and receive (rx) rate, disk I/O speed and error rate, number of TCP connections, etc. These system performance metrics are collected at both infrastructure-level (machines) and container-level (microservices).

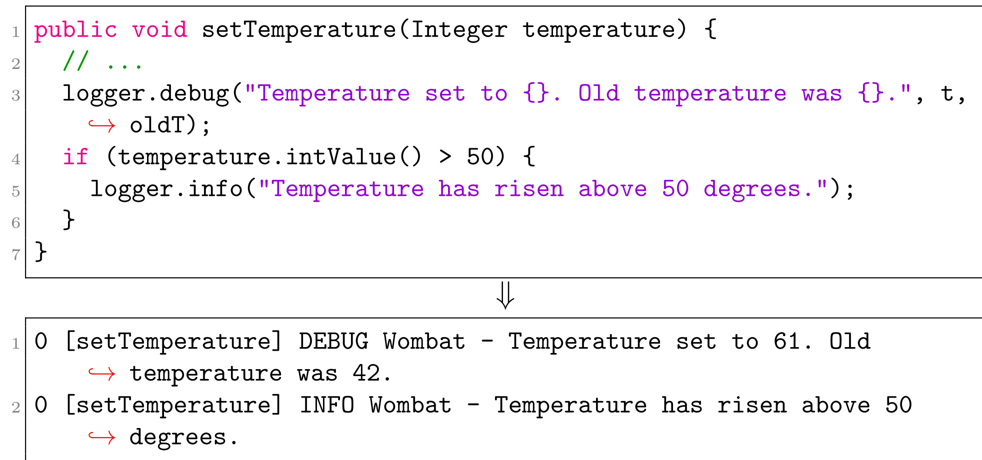
Business Metrics

Business metrics, in addition, reflect the quality of service in a specific time period from the users' aspect. Business metrics, such as response latency, error rate, throughput, mean time to recovery, and availability rate, are also crucial system indicators. Different online services value different business metrics. For example, availability and error rate are common performance attributes of transactional services, while video streaming services are usually based on throughput.

Owing to the distributed and virtualized nature of microservices, the number of monitoring metrics explodes. Also, the mutual influence between monitoring metrics becomes exquisite, making the metric analysis more complex than ever [49, 51].

#### Logging

Software logs have been widely employed in a variety of reliability assurance tasks. Logs also play an indispensable role in data-driven decision-making in industry [123]. In general, logs are semi-structured text printed by logging statements (*e.g.*, `printf()`, `logger.info()`) in the source code. They often record software runtime information with text. For example, in Figure 2.3, the two log messages are printed by the two logging



```
1 public void setTemperature(Integer temperature) {
2   // ...
3   logger.debug("Temperature set to {}. Old temperature was {}.", t,
      ↪ oldT);
4   if (temperature.intValue() > 50) {
5     logger.info("Temperature has risen above 50 degrees.");
6   }
7 }
```

⇓

```
1 0 [setTemperature] DEBUG Wombat - Temperature set to 61. Old
    ↪ temperature was 42.
2 0 [setTemperature] INFO Wombat - Temperature has risen above 50
    ↪ degrees.
```

Figure 2.3: An example of logging statements by SLF4J and the generated logs.

statements in the source code. The first few words (*e.g.*, "Wombat") of the log messages are decided by the corresponding logging framework (*e.g.*, SLF4J) and they are in structured form. On the contrary, the remaining words (*e.g.*, "50 degrees") are unstructured because they are written by developers to describe specific system runtime events.

### Alerting

Alerting is a practical way to call for immediate human intervention upon system anomalies. I introduce the necessities, attributes, generation, and clearance of alerts as the background knowledge. I also provide some typical examples of alerts from production cloud systems.

#### Necessities of Alerts

Service reliability is one of the most important factors for both online service providers and their clients. To clients, unplanned service failure may cause serious damages to their applications. To service providers, offering reliable services can attract more

clients and will bring them higher market share and more profit. Service providers and clients will reach a Service Level Agreement (SLA) on the reliability of the target services. Service available time, as a direct indicator, is often included in SLAs. Failures that prevent online services from properly functioning are inevitable [31]. In order to satisfy SLAs, online service providers need to deal with service and microservice anomalies before they escalate their effect into severe failures and incidents. Alerting is a practical way to achieve this goal. Figure 2.4 demonstrates the significance of alerts. By continuously monitoring online services via traces, logs, metrics, the monitoring system will send alerts[3] to OCEs upon detecting anomalous service states. With the information provided in the alerts, OCEs can judge with their domain knowledge, fix the problems, and clear the alert. As a result, unplanned failures and incidents can be avoided or quickly mitigated.

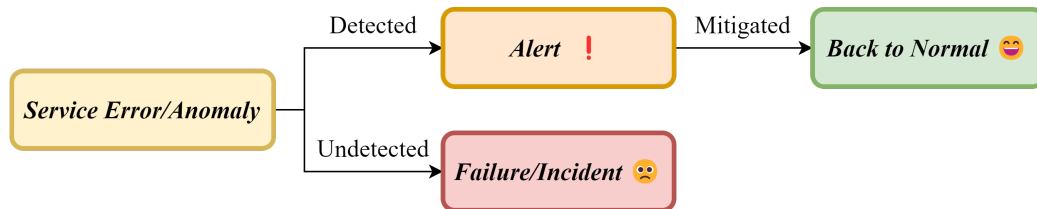



Figure 2.4: The significance of alerts for cloud reliability.

### Attributes of Alerts

Alerts have many attributes that are helpful for OCEs' diagnosis, including title of alerts, severity level, time, service name, duration, location information. The *Title of an Alert* concisely describes the alert. Typically, the title should contain information like "the affected service or microservice" and "the manifes-

[3] Here I only focuses on the alerts that indicate potential bugs and failures, i.e., the system reliability alerts.

Table 2.3: Sample reliability alerts in a cloud system. The names of microservices are omitted due to confidentiality.

| No. | Severity | Time | Service | Alert Title | Duration | Location |
|---|---|---|---|---|---|---|
| 1 | Minor | 2021/05/18 21:04 | Elastic Computing | Connection to Authentication service timed out | 52 sec | Region=X;DC=2;... |
| 2 | Major | 2021/05/18 06:36 | Block Storage | Failed to allocate new blocks, disk full | 10 min | Region=X;DC=1;... |
| 3 | Critical | 2021/05/18 06:38 | Database | Failed to commit changes ... | 2 min | Region=X;DC=1;... |
| 4 | Critical | 2021/05/18 06:38 | Database | Failed to commit changes ... | 5 min | Region=X;DC=1;... |
| 5 | Critical | 2021/05/18 06:39 | Database | Failed to commit changes ... | 5 min | Region=X;DC=1;... |
| 6 | Warning | 2021/05/18 09:01 | API Gateway | Requests to APIGW exceed threshold | 2 min | Region=Y;DC=1;... |

tation of the failure". The OCEs will look up the alert title to find the corresponding SOP and perform predefined actions to mitigate the alert. The *Severity Level* indicates how severe the alert is. The corresponding *Alert Strategy* defines the severity level and alert title according to the nature of the affected service or microservice. The *Time* means the time of occurrence of the alert, and *Duration* is the duration between the occurrence and the clearance of the alert. The *Location Information* contains the necessary information to locate the anomalous service or microservice. Table 2.3 shows the samples of alerts from the monitoring system of Huawei Cloud.

#### Generation of Alerts

An alert represents a specific abnormal state of the cloud system. The first and foremost step of alert generation is anomaly detection. Anomaly detection in logs [63,81,167], traces [59,149,174], and monitoring metrics [92, 105, 164] of the cloud system have been widely studied.

The cloud monitoring system will continuously detect anomalies and generate system reliability alerts according to the alert strategies associated with specific services or microservices. The strategies for system reliability alerts can be divided into three categories, i.e., probes, logs, and metrics.

- *Probes:* The cloud monitoring system will send probing requests to the target services and receive the heartbeat from the target services. Typically, OCEs set fixed thresholds of no-response time for different services as the strategy of probes. If a target service does not respond to the probing requests for a long time, an alert will be generated.

- *Logs:* The cloud monitoring system will process logs of the

target services. OCEs can set flexible rules for different services. Typical rules of logs are keyword matching, e.g., "IF the logs contain 5 `ERROR`s in the past 2 minutes, THEN generate an alert." Traces can also be viewed as special logs and will be processed similarly.

- *Metrics:* Performance metrics are time series that show the states of a running service, e.g., latency, no. of requests, network throughput, CPU utilization, disk usage, memory utilization, etc. The alert strategy for metrics varies from static threshold to algorithmic anomaly detection. Typical rules of metrics are like "IF the disk usage exceeds 90%, THEN generate an alert" and "If abnormal network throughput is detected, then generate an alert".

Clearance of Alerts

Alerts can be cleared manually or automatically. On the one hand, after the human intervention, if the OCE confirms the mitigation of the anomaly, the OCE can manually mark the alert as "cleared". On the other hand, the cloud monitoring system can automatically clear some alerts. For system reliability alerts of *probes* and *metrics*, the cloud monitoring system will continue to monitor the status of the associated service. If the service returns to a normal state, the cloud monitoring system will mark the corresponding alert as "automatically cleared".

## 2.2 Research Problems

### 2.2.1 Intensity of Microservice Dependency

A failed service will only affect services that will invoke it. In other words, service invocations cause dependencies between ser-

vices. Many recent approaches [99, 151] propose to use the dependencies of services to approximate their failure impact. All the services and dependencies in an online service system collectively construct a directed graph of services, which is also called a dependency graph. Identifying whether one service depends on another in online service systems can be well solved by industrial tracing frameworks like Dapper, Jaeger, and Zipkin. By using these frameworks, all the invocations between the caller and callee services can be recorded as traces that are composed of spans. The attributes about each invocation, like duration, status, invoked service name, timestamp, etc., are recorded in each span. Based on the spans, current dependency detection methods treat the dependency as a binary value indicating whether one service invokes another or not.

However, modeling the relations of services solely with binary dependencies is not precise enough. Our empirical study on the outages of Amazon Web Service and Huawei Cloud points out that the existing binary definition of dependency will cause inefficiency in fault diagnosis and failure recovery. This is because the callee microservice impact the caller services in different ways. Hence, the procedure of failure recovery can be sped up by skipping those unimportant services. Manual examination of different dependencies without any priority is inefficient, especially in cloud systems where the number of dependencies could be large. Based on this observation, I argue that it will be helpful if the dependency can be measured as a continuous value that indicates the intensity of this dependency. Specifically, by checking services that are dependent on the failed service with large intensity values, OCEs can find the root cause of a system failure with a higher probability. By recovering the services

that are strongly dependent on the failed one, the whole system could be restored faster.

### 2.2.2 Microservice Resilience Testing

The resilience of a microservice system refers to the ability to maintain the performance of services at an acceptable level and recover the service back to normal when a failure in one or more parts of the system causes the service degradation [127, 152]. Building resilient online services becomes necessary as faults and failures are unavoidable [68, 91]. Resilience to unexpected failures is also essential for reducing downtime, maintaining the quality of service, and meeting the service-level agreement.

Resilience testing [106] is one of the primary ways to measure the resilience of software. All new or updated microservices need to pass many resilience tests to ensure the resilience of online services. Unlike other test techniques that ensure the functional correctness of software [8], resilience testing focuses on making applications perform core functions and avoid data loss under stress or in chaotic environments. By purposefully introducing failures into the system, the test engineers can monitor how the microservice system performs and improve the architectural design according to the discovered flaws [65]. If the microservice can still provide acceptable service and is not affected by the failures, it passes the resilience tests. Industrial practitioners also employ chaos engineering [15, 22] to test their software's resilience in the production environment with live traffic. On balance, the steps involved in resilience testing include [83] *determining metrics*, *generating load*, *introducing failures*, *collecting metrics*, and *getting test results*.

For example, the resilience test of a program when it encoun-

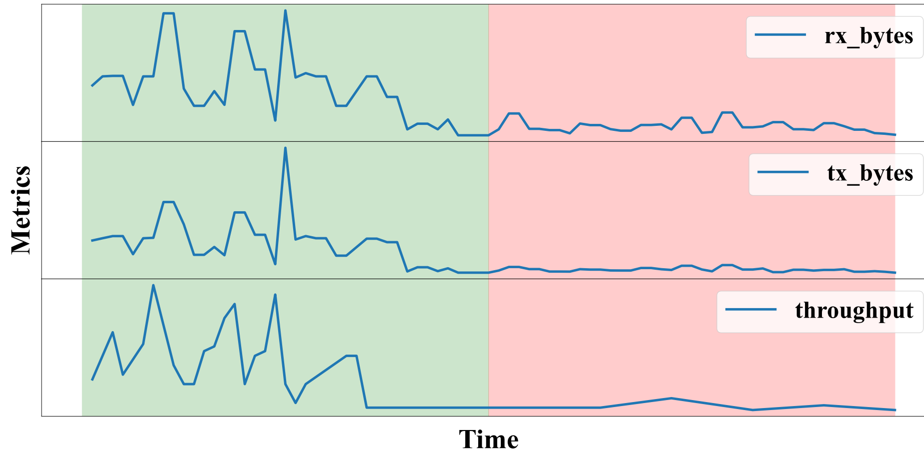



Figure 2.5: The monitoring metrics during the normal period (the green area) and the failure injection period (the red area).

ters a high network packet loss should be accomplished by the following steps. First, test engineers need to introduce network packet loss failure with proper tools. The failure should be injected into the network interface. Then the second step will be collecting the corresponding monitoring metrics based on their domain knowledge. Network tx and rx rate, request throughput will be selected in this case. Suppose the monitoring metrics are shown in Figure 2.5. The green area is the normal period, and the red area is the failure injection period. After comparing the duration and magnitude of the monitoring metrics, engineers can conclude that the program is not resilient to the network packet loss failure because the throughput dropped a lot. In conclusion, the program failed this resilience test.
Automation of the resilience testing procedure is possible, but burdensome standardization of test parameters is still required. In practice, to standardize the resilience testing procedure, test engineers manually determine the set of rules for each failure type. Each set of rules contains five parts, i.e., *failure type*, *load*,

*target metrics*, *degradation measurement*, and *pass criteria*. The five parts are explained below.

- *Failure type* is the failure to inject. Tested service is expected to be resilient to the failure.
- *Load* should be based on the maximum load that the service can handle without encountering performance issues.
- *Target metrics* are the metrics to examine. They should manifest the degradation caused by the injected failure clearly. This may include I/O rates, throughput, mean time to recovery, latency, as well as the relationship between the aforementioned metrics.
- *Degradation measurement* measures the degree of degradation. It is usually measured by the duration and magnitude change of the target metrics.
- *Pass criteria* is the criteria for deciding whether the resilience test is passed. The monitoring metrics under normal and faulty periods should be analyzed, and a `PASS/FAIL` conclusion will be drawn. The criteria should be decided concerning the anticipated service quality.

For each failure type, test engineers follow the aforementioned procedure for resilience testing.

### 2.2.3 Anti-patterns of Alerts

An alert is a notification about a specific abnormal state of the microservice system sent to on-call engineers. Alerts are crucial for requesting prompt human intervention upon system anomalies. The configuration of alert strategies is empirical, which

heavily depends on human expertise. Since different cloud services exhibit different attributes and serve different purposes, their alert strategies vary significantly. In particular, the empiricalness of alert strategies results from two aspects of cloud services. On the one hand, a cloud service's abnormal state may differ because each cloud service implements its own business logic. There is no one-fits-all rule for anomaly detection on cloud services, i.e., *when to generate an alert*. On the other hand, the attributes of an alert that helps the manual inspection and mitigation of the abnormal state, e.g., the location information and the free-text title that describes the alert, are also service-specific and lack comprehensive guidelines. In other words, "*what attributes and descriptions an alert should have*" also depends on human expertise. In summary, the configuration of alert strategies, as a precursor step for human intervention in cloud anomalies, is an empirical procedure. The quality of alerts significantly affects the cloud reliability and the cloud provider's business revenue. In practice, I observe on-call engineers being hindered from quickly locating and fixing faulty cloud services because of the vast existence of misleading, non-informative, non-actionable alerts. I call the ineffectiveness of alerts "anti-patterns of alerts".

To better understand the anti-patterns of alerts and provide actionable measures to mitigate anti-patterns, in this thesis, I conduct the first empirical study on the practices of mitigating anti-patterns of alerts in an industrial cloud system. I will study the alert strategies and the alert processing procedure at Huawei Cloud, a leading cloud provider. I also survey current reactions to mitigate the anti-patterns of alerts, and the general preventative guidelines for the configuration of alert strategy. Our study

combines the quantitative analysis of millions of alerts in two years and a survey with eighteen experienced engineers.

---

□ **End of chapter.**

# Chapter 3

# Predicting the Aggregated Intensity of Dependency

## 3.1 Introduction

Service reliability is one of the key challenges that cloud providers have to deal with. The common practice nowadays is developing and deploying small, independent, and loosely coupled cloud microservices that collectively serve users' requests. The microservices that serve the same purpose are called cloud services[1]. The microservices communicate with each other through well-defined APIs. Such an architecture is called microservice architecture [108]. The microservice architecture has been widely adopted in cloud systems because of its reliability and flexibility. Under this architecture, microservice management frameworks like Kubernetes will be responsible for managing the life cycles of microservices. Developers can focus on the application logic instead of the bothering tasks of resource management and failure recovery.

Although microservice management frameworks provide auto-

[1]For simplicity, in this chapter, "cloud service" and "cloud microservice" are interchangeable when they are used alone.

matic mechanisms for failure recovery, unplanned service failures may still cause severe cascading effects. For example, failures of critical services that provide basic request routing functions will impact the invocation of cloud services, slow down request processing, and deteriorate customer satisfaction. Therefore, evaluating the impact of service failures rapidly and accurately is critical to the operation and maintenance of cloud systems. Knowing the scope of the impact, reliability engineers can put more emphasis on services that have greater impacts on others. A failed service will only affect services that will invoke it. In other words, service invocations cause dependencies between services. Many recent approaches [99, 151] propose to use the dependencies of services to approximate their failure impact. All the services and dependencies in a cloud system collectively construct a directed graph of services, which is also called a dependency graph. Identifying whether one service depends on another in cloud systems can be well solved by industrial tracing frameworks like Dapper and Jaeger. By using these frameworks, all the invocations between the caller and callee services can be recorded as traces that are composed of spans. The attributes about each invocation, like duration, status, invoked service name, timestamp, etc., are recorded in each span. Based on the spans, current dependency detection methods treat the dependency as a binary value indicating whether one service invokes another or not.

However, modeling the relations of services solely with binary dependencies is not precise enough. To show the insufficiency of existing methods, I first conduct an empirical study on the outages of Amazon Web Service and Huawei Cloud. I point out that it is inefficient to conduct failure diagnosis and recovery based

on binary dependencies. This is because the different dependencies of a cloud service impact the cloud service in different ways. Manual examination of different dependencies without any priority is inefficient, especially in cloud systems where the number of dependencies could be large. Based on this observation, I argue that it will be helpful if the dependency can be measured as a continuous value that indicates the intensity of this dependency. Specifically, by checking services that are dependent on the failed service with large intensity values, on-call engineers (OCEs) can find the root cause of a system failure with a higher probability. By recovering the services that are strongly dependent on the failed one, the whole system could be restored faster.

To improve the reliability of cloud systems, in this thesis, I propose AID, an end-to-end approach to predict the intensity of dependencies between cloud microservices for cascading failure prediction. I first generate a set of candidate dependency pairs from the spans. Then I distribute each span into different fixed-length bins according to their timestamp and service name. I calculate the statistics of all spans in each bin as the Key Performance Indicators (KPIs) for the bin. The KPIs of one service form a multivariate time series that will be treated as the representation of the service's status. For each candidate dependency pair, I calculate the similarities between the statuses of the two services in the pair. Finally, I aggregate the similarities to produce a unified value as the intensity of the pair.

To show the effectiveness of AID, I evaluate AID on two datasets. One is a simulated dataset, and the other is an industrial dataset. For the simulated dataset, I deploy train-ticket, an open-source microservice benchmark system, simulate users' requests, and

collect the traces. For the industrial dataset, I collect the traces from a production cloud system. Then I evaluate AID on the datasets and compare its performance with several baselines. The experimental results show that our proposed method can accurately measure the intensity of dependencies and outperform the baselines. Furthermore, I showcase the successful usage of our method in a large-scale production cloud system. In addition, I release both datasets to facilitate future studies.

The main contributions of this work are highlighted as follows:

- I conduct a comprehensive industrial survey and the first empirical study to identify the inefficiency of using binary-valued dependency for failure diagnosis and failure recovery.

- I propose AID, the first method to quantify the intensity of dependencies between different services.

- The evaluation results show the effectiveness and efficiency of the proposed method.

- I release a simulated dataset and an industrial dataset from a production cloud system to facilitate future studies.

- Additionally, AID have been successfully applied in a leading public cloud provider, and helped greatly reduce manual maintenance effort.

## 3.2 Motivation

The research described in this thesis is motivated by the maintenance of a real-world cloud system in production. In this section, I first survey thirteen publicly known service outages that severely affected Amazon Web Services (AWS) from 2011 to

Table 3.1: Summary of AWS outages related to service dependency.

| Date | Consequences | |
|---|---|---|
| | Cascading Failure | Slow Recovery |
| Apr 21, 2011 | ✓ | |
| June 29, 2012 | | ✓ |
| Oct 22, 2012 | ✓ | |
| Aug 7, 2014 | | ✓ |
| Nov, 25 2020 | ✓ | ✓ |

2020. Among the thirteen outages, I identify five that are related to service dependency and summarized the consequences of inappropriate management of service dependency. Second, I empirically study the diagnosis records of five real outages in the cloud system of Huawei Cloud that are related to inappropriate management of service dependency. Our study indicates that the information in the traces has not been used efficiently and current practice heavily relies on the engineers' familiarity with the dependencies in the system. Lastly, I propose to measure the intensity of dependency in terms of status propagation between dependent cloud services. I demonstrate the usefulness of the intensity by motivating examples in real cloud systems.

### 3.2.1 A Survey of the Outages in AWS

Service outages are inevitable in the cloud [31]. In this section, I empirically analyzed over 1000 incidents of Huawei Cloud in 2019 and thirteen publicly known major outages[2] of AWS from 2011 to 2020. Among the incidents of Huawei Cloud, I found that improper service dependency is the most frequent reason for failures in Huawei Cloud. Among the outage summaries of

[2] https://aws.amazon.com/premiumsupport/technology/pes/

AWS, I also identified that five of the outages (38%) are related[3] to service dependency. As shown in Table 3.1, among the five outages that are related to service dependency, three of them are due to cascading failures triggered by erroneous upgrades of services. During the failure recovery, the inappropriate dependencies lead to slow failure recovery in three outages.

AWS is the worldwide leading cloud provider. It operates in many regions, each consisting of multiple Availability Zones (AZs). Each AZ uses separate physical facilities and independently provides various cloud services [38], including Steam Data Processing (Kinesis), API Usage Analysis (Cognito), Customer Dashboard (Cloudwatch), Elastic Compute Cloud (EC2), Relational Database Service (RDS), Elastic Load Balancing (ELB), and Low-level Block Storage (EBS), etc. For brevity's sake, I simplify the dependencies as 1) EC2, RDS, and ELB all depend on EBS, and 2) Cognito and Cloudwatch depend on Kinesis[4].

The outages on April 21, 2011, and October 22, 2012, are both caused by erroneous upgrades of EBS. When EBS failed, the services that depend on EBS, i.e., EC2,ELB, and RDS, are all affected. The cascading failures resulted in service disruptions of over 48 hours in the US-East-1 Region of AWS.

The outages on June 29, 2012, and August 7, 2014, are both triggered by the blackouts. After the blackout, the RDS and ELB services restarted quickly as expected, but they are still unable to fully recover because they both depend on EBS service which, at that time, can not recover simultaneously. The slow failure recovery incurred by service dependencies affected the

[3]The outages are usually caused by various reasons that mutually affect each other. Service dependency is one of the reasons, so I use the word "related".

[4]The actual dependency relations between these services are complicated. I omit the details here.

service availability for days in the US-East-1 Region and the EU West-1 Region of AWS. As a follow-up optimization, ELB service reduced the dependency on EBS after the outage in 2014. On November, 25 2020, the erroneous upgrade of Kinesis lead to its failure, cascadingly causing the failure of Cognito and Cloudwatch. More severely, during the recovery, AWS could not notify the customers via the normal way because the normal customer notification service also relied on Cognito. Due to the inner mechanism of Kinesis, the recovery of Kinesis took more than ten hours. Thus the recoveries of Cognito and Cloudwatch were also slowed down. As a follow-up optimization, Cognito and Cloudwatch services reduced the dependency on Kinesis after the severe outage.

### 3.2.2 Drawbacks of Current Failure Diagnosis Methods

To gain more knowledge about the procedure of failure diagnosis in industrial circumstances, I first interviewed engineers in Huawei Cloud[5]. Then I summarize the procedure of failure diagnosis, and point out the drawbacks of current practice in Huawei Cloud.

In Huawei Cloud, the failure diagnosis can be triggered by two systems, i.e., the customer support system and the monitoring system. When a customer experiences a service disruption, the customer can submit a support ticket in the customer support system. The on-call engineers will distribute the support ticket to the corresponding engineers responsible for the service. The monitoring system, on the other hand, monitors the Key Per-

[5]AWS does not disclose the detailed procedures of failure diagnosis related to the five outages, so I cannot analyze the aforementioned outages in depth.

formance Indicators (KPIs) and the logs of each service in the cloud system. If the KPIs or the number of erroneous logs of one service increased abnormally or reached predefined thresholds, the monitoring system will send an alert to the corresponding engineers. Upon receiving the support ticket or alert, engineers start diagnosing the failures.

I summarize the common practice of failure diagnosis in Huawei Cloud as follows. Suppose the anomalous service is `A`, OCEs will first check whether the failure is caused by the faults of service `A` (e.g., an erroneous upgrade). If so, the development team of service `A` will handle the failure. If service `A` is in good condition, OCEs will analyze the status of all services that `A` depends on. The status includes the number of calls, the error rate, etc. If they found the failure of a service `B` is likely to cause the failure of service `A`, then engineers will continue to investigate service `B`. Recall that all the services construct a directed graph where each node represents a service. The failure diagnosis procedure can be viewed as a recursive search on the service dependency graph.

The practice works well in small cloud systems that contain tens of cloud services. However, the dependencies in large-scale cloud systems are much more complicated [51], making manual failure diagnosis inefficient and difficult for engineers. Engineers may have trouble identifying the cause of the failure. In this case, the development teams of all cloud services have to check whether the failure is caused by their corresponding services. Sometimes engineers may infer the possible causes of a failure, but it heavily relies on the engineer's familiarity with the dependencies in the system. In summary, the complex dependency relations in large-scale cloud systems make failure diagnosis difficult, and current

practice is inefficient and dependent on the human experience.

### 3.2.3 Intensity of Service Dependency

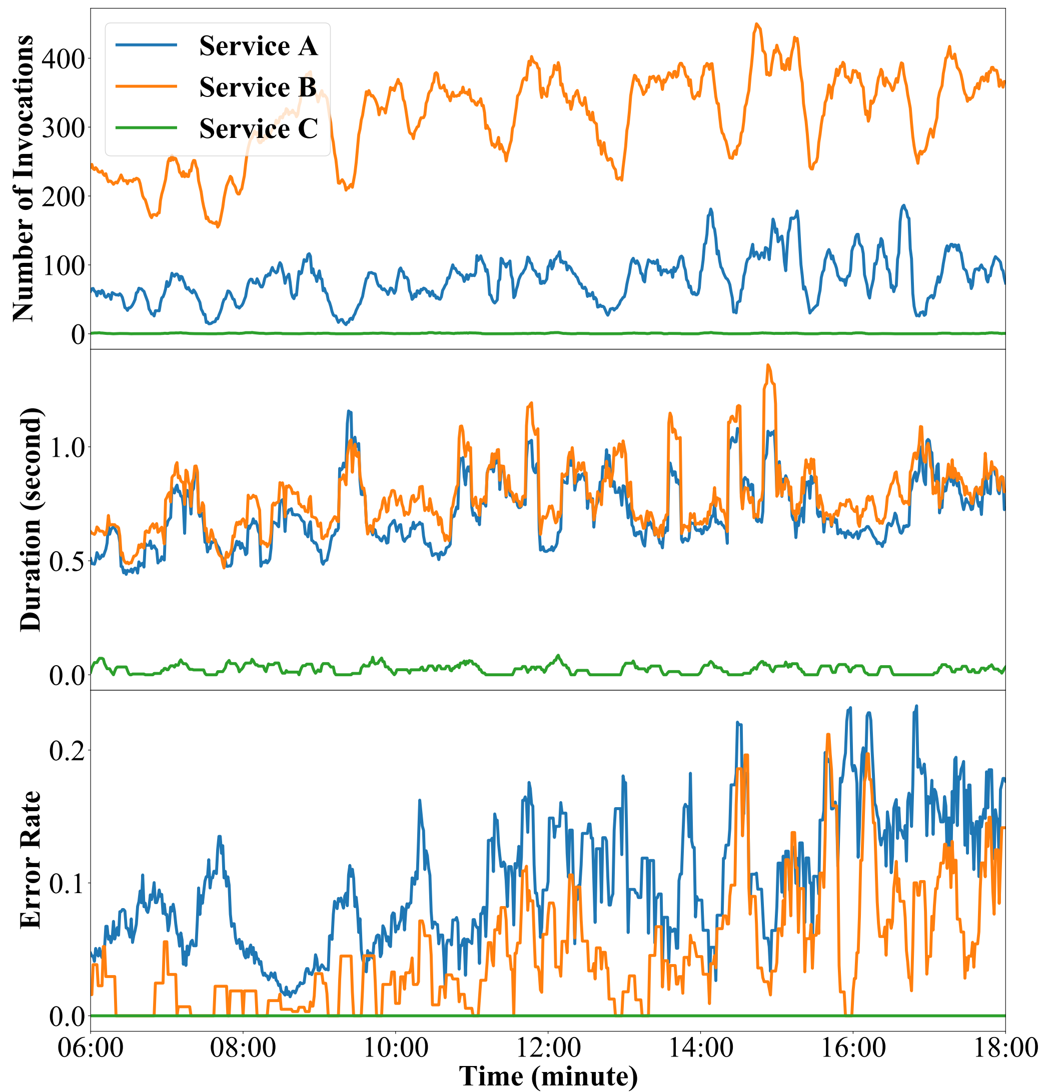



Figure 3.1: The statuses of service `A`, `B` and `C`. `A` invokes `B` and `C` but `B` has a greater effect on `A`.

A cloud system is composed of many services. The dependency between two services is caused by one service invoking the other via predefined APIs. Existing tools [97, 99, 157] treat the de-

pendency as a binary relation, i.e., if the caller service invokes the callee service, then the caller is dependent on the callee. I suggest that this binary dependency metric is not fine-grained enough for cloud maintenance. Figure 3.1 shows the statuses of three services[6] `A`, `B`, and `C` in Huawei Cloud. Service `A` invokes both service `B` and service `C`. Service `B` encountered failures. The x-axis represents time in minute. The y-axes represent the number of invocations per minute, the average duration of invocations per minute, and the error rate per minute of `A`, `B`, and `C`. Although service `A` invokes service `B` and service `C`, it is obvious that the statuses of `B` and `C` influence the status of `A` in different degrees. The reason is that the functionalities provided by service `A` and `B` are creating virtual machines, and allocating block storage, respectively. Creating a virtual machine requires allocating one or more block storage. Thus, the failure of service `B` inevitably affects service `A`. On the contrary, due to the fault tolerance mechanism of service `A`, the failure of service `C` will not affect service `A` a lot. Thus, it is more accurate to say that the intensity of dependency between service `A` and service `B` is higher than the intensity of dependency between service `A` and service `C`. As can be seen in Figure 3.1, the similarity of the statuses reflect the difference in the intensities.

Ideally, if the development team of every cloud microservice accurately provides the intensity of dependencies for every dependent services, the failure diagnosis could be accelerated. OCEs can prioritize the services that exhibit higher intensity of dependency instead of inspecting all the dependent services (Section § 3.2.1) if they have accurate intensity information. However, due to the complexity and the fast-evolving nature of cloud

[6] For confidentiality reasons, I cannot reveal the names of related services.

systems [3], manually maintaining the dependency relations with intensity is very difficult. As a result, OCEs often struggle in diagnosing failures due to the lack of intensities. In order to relieve the pressure on OCEs, I propose to predict the intensity of dependency from the statuses of services.

## 3.3 Problem Definition

Motivated by Section § 3.2, I define the intensity of dependency between two microservices as how much the status of the callee microservice influences the status of the caller microservice. Formally, given a pair of microservices $(P, C)$ where $P$ means the caller microservice and $C$ means the callee microservice, the proposed approach should produce an intensity value $I \in (0, 1)$ that represents to what extent the status of $C$ will affect the status of $P$.

## 3.4 Methodology

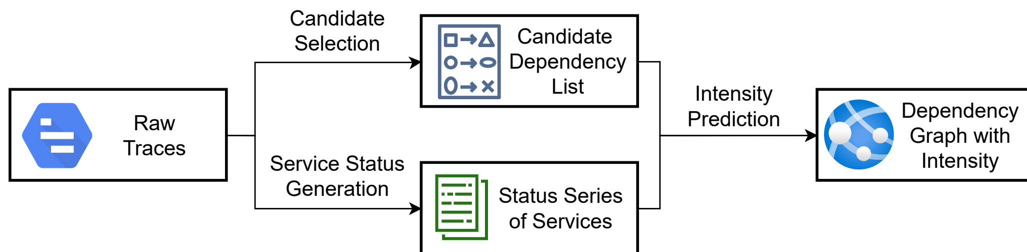



Figure 3.2: The overall workflow of AID.

In this section, I present AID, a framework for predicting the Aggregated Intensity of service Dependency in large-scale cloud systems. I first present the overall workflow of AID. Then I

elaborate on each step in detail, i.e., candidate selection, service status generation, and intensity prediction.

### 3.4.1 Overview

The intensity of dependency is inherently determined by the program logic of microservices. Manual evaluation of the intensity of dependency is hard due to the dynamic and decoupled nature of microservice systems. Thus, in this thesis, I propose AID to predict the intensity of dependency based on all the spans between microservices $(P, C)$ over a period $T$. In this way, non-intrusive prediction of intensity of dependency can be achieved, which eases the system integration and increases the applicability of AID.

The overall workflow of AID is illustrated in Figure 3.2. AID consists of three steps: candidate selection, status generation, and intensity prediction. Given the raw traces, AID first generates a set of candidate service pairs $(P, C)$ where service `P` directly invokes service `C` (Section § 3.4.2). The intuition is that direct service invocation incurs direct dependency to some degree. Indirect dependencies through the transitivity of service invocation will be discussed in Section § 3.7.1. For status generation, AID generate the status of all services (Section § 3.4.3). The status of one service is composed of three aspects of dependency, i.e., number of invocations, duration of invocations, error of invocations. Each aspect of the service's status contains one or more Key Performance Indicators (KPIs), depending on the actual implementation of the distributed tracing system. A KPI is an aggregated value of a service status of all the spans of a service in a fixed time interval, e.g., 1 minute. AID use the statistical indicators of each aggregation as the values of

the KPIs. Motivated by the experience of engineers introduced in Section § 3.2.2, I propose to predict the intensity of service dependencies from the similarity of the statuses of dependent services. The intuition behind using the similarity of time series is to evaluate the propagation of service statuses. The intensity prediction step (Section § 3.4.4) predicts the intensity of dependency by measuring the similarity between two service's statuses. The similarity between two service's statuses is a normalized and weighted average of the similarity of all the KPIs of the two services. AID calculate the similarity between two KPIs by a dynamic status warping algorithm. Finally, AID produces the dependency graph with intensity.

### 3.4.2 Candidate Selection

In general, direct service invocations can be divided into two categories, i.e., synchronous invocations and asynchronous invocations. Modern tracing mechanisms can keep track of both synchronous and asynchronous invocations [120]. Given all the raw traces of the cloud system, in this step, AID generate a candidate dependency set $Cand$. The candidate dependency set $Cand$ contains service invocation pairs $(P_1, C_1), (P_2, C_2), \cdots, (P_n, C_n)$. Each pair $(P_i, C_i)$ in the candidate dependency set denotes that the service named $P_i$ invokes the service named $C_i$ at least once. Therefore, service $P_i$ depends on service $C_i$. This step is to shrink the search space of possible dependent pairs because the service invocations indicate direct dependencies.

To generate the candidate dependency set, AID need to know the name of the caller service and the callee service. The name of callee service is clearly recorded in the span, but the name of the caller service is not. Hence, AID first augments each span $s$ by

adding another attribute $s^{pname}$ which denotes the service name of the parent span. Specifically, the augmentation of attribute $s^{pname}$ is achieved by 1) looking for another span $s'$ whose *id* is the same as $s^{pid}$, and 2) set the *name* of $s'$ as $s^{pname}$. Then AID iterates over all the spans and add $(s^{pname}, s^{name})$ to the candidate dependency set by the union operation.
For example, assuming the name of services are the same as the index of spans, the six spans in Figure 2.2 will result in a candidate set of { $(Service_0,Service_1)$, $(Service_1, Service_4)$, $(Service_0, Service_2)$, $(Service_0,Service_3)$, $(Service_3,Service_5)$ } .

### 3.4.3 Service Status Generation

In this step, AID generates the status of all cloud services from the traces. AID starts by defining the status of a cloud service (i.e., service status) and then describe the procedure of service status generation.

#### Definition of Service Status

A service invocation is composed of three logical components, i.e., the caller service, the callee service, and the network communication. In particular, the caller service initiates an invocation to the callee service via the network. The callee service then processes the invocation, during which it may invoke other services. After the processing is finished, the callee service will send the result, e.g., status, to the caller service via the network. Hence, AID could derive three aspects of service invocations: *initiation of invocation*, *processing*, *result*. As service invocations occur repeatedly, the three aspects of service invocations

can derive three aspects of service dependency:

- *Number of Invocations*: The number of invocations from the caller to the callee.
- *Duration of Invocations*: The duration of invocations.
- *Error of Invocations*: The number of successful invocations from the caller to the callee.

#### Representation of Service Status

In a cloud system, the spans record information about every invocation. Intuitively, the status of a cloud service can be easily obtained from the spans of that service. Inspired by the common practice in cloud monitoring [1], AID distributes the spans of one service into many bins according to the spans' timestamps. Each bin accepts spans whose timestamp is in a short, fixed-length period. I denote the length of the short period as $\tau$. For example, the span shown in Table 2.2 will be put in the bin of `ts-preserve-service` at time 04:58, April 17 2020. AID can then represent the status of a cloud service in a short period by the statistical indicators of all the spans in the corresponding bin.

Formally, given all the spans in the cloud system over a long period $T$, AID first initiates $\mathbf{S} \times \mathbf{N}$ empty bins of the predefined size $\tau$. $\mathbf{S}$ is the number of microservices. $\mathbf{N}$, determined by $\frac{T}{\tau}$, is the number of bins. Then AID distributes all spans into different bins according to their timestamp $s^{ts}$ and service name $s^{name}$. After that, AID calculates the following three types of indicators as the KPIs for each bin.

- $invo_t^M$: Total number of invocations (spans) in the bin;

- $err_t^M$: Error rate of the bin, i.e., the number of errors divided by the number of invocations;
- $dur_t^M$: Averaged duration of all spans in the bin;

where $t$ is the time of the bin and $M$ is the microservice name of the bin. If a service is not invoked in a particular bin (i.e., the corresponding bin is empty), all the KPIs will be zero. In the end, AID gets the KPIs of every service $M$ at every period $t$. Ordering the bins by $t$, AID gets three time series of KPIs for each cloud service, denoted as $invo^M$, $err^M$, and $dur^M$. I name the time series of server KPIs as status series.

### 3.4.4 Intensity Prediction

In this thesis, I define the intensity of dependency between two services as *how much the status of the callee service influences the status of the caller service*. The step of intensity prediction quantitatively predicts the intensity of dependency by measuring the similarity between two services' status series. Specifically, AID calculates the similarity of two different status series with dynamic status warping and aggregate all the similarities to get the overall similarity.

#### Dynamic Status Warping

Inspired by the dynamic time warping algorithm (DTW) [75], I propose the dynamic status warping (DSW) algorithm (Algorithm 1) to calculate the distance between two status series. DSW automatically warps the time in chronological order to make the two status series as similar as possible and get the similarity by summing the cost of warping. It utilizes dynamic

**Algorithm 1:** Dynamic Status Warping

**Input:** The status series of caller service and callee service $status^P, status^C$; duration series of callee $dur^C$, estimated round trip time $\delta_{rtt}$, max time drift $\delta_d$

**Output:** The similarity between two status series

1 Set the warping window $w = \max(dur^C) + \delta_{rtt}$
2 $K = length(status^C)$
3 $N = length(status^P)$
4 Initialize the cost matrix $\mathbf{C} \in \mathbb{R}^{K \times N}$, set the initial values as $+\infty$
5 $\mathbf{C}_{1,1} = (status_1^P - status_1^C)^2$
6 **for** $i = 2 \ldots \min(\delta_d, K)$ **do** // Initialize the first column
7     $\mathbf{C}_{i,1} = \mathbf{C}_{i-1,1} + (status_1^P - status_i^C)^2$
8 **end**
9 **for** $j = 2 \ldots \min(w + \delta_d, N)$ **do** // Initialize the first row
10     $\mathbf{C}_{1,j} = \mathbf{C}_{1,j-1} + (status_j^P - status_1^C)^2$
11 **end**
12 **for** $i = 2 \ldots K$ **do**
13     **for** $j = \max(2, i - \delta_d) \ldots \min(N, i + w + \delta_d)$ **do**
14         $\mathbf{C}_{i,j} = \min(\mathbf{C}_{i-1,j-1}, \mathbf{C}_{i-1,j}, \mathbf{C}_{i,j-1}) + (status_j^P - status_i^C)^2$
15     **end**
16 **end**
17 **return** $\mathbf{C}_{K,N}$

programming to calculate an optimal matching between two status series. Given two services $P$, $C$, and their status series $invo^P$, $invo^C$, $err^P$, $err^C$, $dur^P$, and $dur^C$, the warping from the callee $C$ to the caller $P$ is specially designed for the cloud environment. The design considerations include:

*Directed warping:* Due to the latency of the network and the time of processing, it takes some time for the status of the callee service to affect the status of the caller service. Therefore, different from dynamic time warping, the time warping of DSW is directed, meaning that the matching from the callee to the caller can only happen in chronological order.

*Adaptive propagation window:* In cloud systems, after the round trip time ($\delta_{rtt}$) plus the duration of request processing, the caller can receive the result of an invocation. Thus, the size of the directed warping window $w$ is automatically set as the maximum duration of the callee's spans plus $\delta_{rtt}$.

*Time drift:* The machine time may drift due to issues with time synchronization in cloud systems, so I add an undirected time drift $\delta_d$ to the warping window.

In summary, $status_i^C$ can only be matched with one of the timestamp in $[status_{i-\delta_d}^P, status_{i+w+\delta_d}^P]$. The DSW returns the warping cost $\mathbf{C}_{M,N}$ as the measure of similarity.

#### Similarity Aggregation

For all $(P_i, C_i) \in Cand$, AID calculates similarities between their status series, denoted as $d_{invo}^{(P_i,C_i)}$, $d_{err}^{(P_i,C_i)}$, and $d_{dur}^{(P_i,C_i)}$. AID normalizes the similarity across the whole candidate set with a min-max normalization with Equation 3.1, where $status \in \{invo, err, dur\}$.

$$d_{status}^{(P_i,C_i)} = \frac{d_{status}^{(P_i,C_i)} - \min(d_{status}^{(P,C)})}{\max(d_{status}^{(P,C)}) - \min(d_{status}^{(P,C)})} \tag{3.1}$$

The intensity of dependency between $P_i$ and $C_i$ is the average similarity of all three similarities between their status series.

$$I^{(P_i,C_i)} = \frac{1}{3} \sum_{status \in S} d_{status}^{(P_i,C_i)},\ S = \{invo, err, dur\} \tag{3.2}$$

Finally, AID can build the dependency graph with intensity from the candidate set and the corresponding intensity values.

## 3.5 Evaluation

In this section, I evaluate AID on both a simulated dataset and an industrial dataset. Particularly, I aim to answer the following research questions (RQs):

- **RQ1.** How effective is AID in predicting the intensity of dependency?
- **RQ2.** What is the impact of different parameter settings?
- **RQ3.** What is the impact of different similarity measures?
- **RQ4.** How efficient is AID?

### 3.5.1 Experimental Setup

#### Dataset

To show the practical effectiveness of AID, I further conduct experiments on the simulated dataset and an industrial dataset from the cloud system of Huawei Cloud. Since there are no existing datasets of trace logs, I deploy a benchmark microservice

system to simulate a real cloud system. I simulate user requests and collect the generated trace logs to construct the simulated dataset. I release both datasets with the paper to facilitate future studies in this field[7].

Table 3.2: Dataset statistics.

| Dataset | TT | Industry[8] |
|---|---|---|
| # Microservices | 25 | 192 |
| # Spans | 17,471,024 | About 1.0e10 |
| # Strong | 18 | 67 |
| # Weak | 1 | 8 |

**Simulated dataset**: For the simulated dataset, I deploy train-ticket [173], an open-source microservice benchmark, for data collection. Train-ticket is a web-based ticketing system with 25 microservices, through which users can search for tickets, reserve tickets, and pay for the reserved tickets. An open-source tracing framework, Jaeger, is used to trace all the API calls. To generate traces, I develop a request simulator that simulates normal users' access to the ticketing system. The simulator will log in to the system, search for tickets, reserve a ticket according to the results of the search, and pay for the ticket. Then I collect the traces from Jaeger and transform the traces into 17,471,024 spans. The dataset is termed as "TT" in Table 3.2.
**Industrial dataset**: Apart from the simulated dataset, I also collected traces from a region of Huawei Cloud to evaluate AID. To support tens of millions of users worldwide, the cloud system of Huawei Cloud contains numerous cloud services and microservices. The service invocations are monitored and recorded by

[7] `https://github.com/OpsPAI/aid`
[8] Only 75 dependencies that the engineers are familiar with are labeled.

an independently developed distributed tracing system. The complex dependency relations in the cloud system increase the burden of OCEs. The OCEs can diagnose problematic microservices timely if the intensity of dependencies can be automatically detected in real-time. To evaluate the practical effectiveness of our method, I collected a 7-day-long trace dataset with 192 microservices in April 2021. The dataset is termed as "Industry" in Table 3.2.

**Manual labeling**: Since our method is unsupervised, labels are only for evaluation. Neither of the datasets has labels about the intensity of dependency, so manual labeling is needed. I set two candidate labels for the intensity of dependency, i.e., "strong" and "weak". Given a candidate dependency pair $(P, C)$, if the failure of service $C$ will cause the failure of service $P$, the intensity between $(P, C)$ should be labeled "strong"; otherwise it should be labeled "weak". For the simulated dataset, two Ph.D. students inspect the source code of all microservices and label every service dependency independently. For the industrial dataset, several senior engineers are invited to manually label the intensity of dependency. In both processes, disagreement on labels will be discussed until consensus is reached. Finally, I convert the "strong" labels to 1 and the "weak" labels to 0 so that they can be effectively compared with the computed intensities. The statistics of the datasets are listed in Table 3.2. "# Microservices" denotes the number of microservices in the dataset. "# Spans" denotes the number of spans in the dataset. "# Strong" and "# Weak" denote the number of dependencies that are labeled with "strong" or "weak" respectively.

**Baselines**

Since there is no existing work that measures the intensity of service dependency, I use Pearson correlation coefficient, Spearman correlation coefficient, and Kendall Rank correlation coefficient as the baseline. Particularly, I calculate correlation on the status series of a candidate dependency pair $(P, C)$, denoted as $corrp_{status}^{(P,C)}$ and $corrs_{status}^{(P,C)}$. For the baselines, I directly use the implementation from the Python package `scipy`. I map the correlation to $[0, 1]$ with the function $f(x) = (x + 1)/2$. The intensities of dependencies are then produced in the same way as Equation 3.2.

**Evaluation Metrics**

I employ Cross Entropy (CE), Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE), as calculated in Equation 3.3, to evaluate the effectiveness of AID in predicting the intensity of dependency. Specifically, cross entropy calculates the difference between the probability distributions of the label and the prediction. Mean absolute error and root mean squared error measures the absolute and squared error. Lower CE, MAE, and RMSE values indicate a better prediction.

$$
\begin{aligned}
CE &= \frac{1}{N}\sum_{i=1}^{N} -[y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i)] \\
MAE &= \frac{\sum_{i=1}^{N} |y_i - p_i|}{n} \\
RMSE &= \sqrt{\frac{\sum_{i=1}^{N} (y_i - p_i)^2}{N}}
\end{aligned} \tag{3.3}
$$

**Experimental Environments**

I run the experiments on the simulated dataset on a Linux server with Intel Xeon E5-2670 CPU @ 2.40GHZ and 128 GB RAM. The experiments on the industrial dataset run on a Laptop with Intel Core i7 CPU @ 2.60 GHz and 16 GB RAM.

### 3.5.2 RQ1: Effectiveness

Table 3.3: Performance Comparison of Different Methods on Two Datasets

| Dataset | Method | Metric | | |
|---|---|---|---|---|
| | | CE | MAE | RMSE |
| TT | Pearson | 0.6872 | **0.3305** | 0.4388 |
| | Spearman | 0.7512 | 0.3735 | 0.4697 |
| | Kendall | 0.6464 | 0.3749 | 0.4577 |
| | AID | **0.4562** | 0.3435 | **0.3859** |
| Industry | Pearson | 0.6076 | 0.4524 | 0.4563 |
| | Spearman | 0.6030 | 0.4501 | 0.4537 |
| | Kendall | 0.6258 | 0.4636 | 0.4656 |
| | AID | **0.3270** | **0.1751** | **0.3044** |

To study the effectiveness of AID, I compare its performance with the baseline models on both the simulated dataset and the industrial dataset collected from Huawei Cloud. For the parameters of AID, I set the bin size $\tau = 1$ *minute*, the estimated round trip time $\delta_{rtt} = 0$. Specially, I set the max time drift $\delta_d = 1$ *minute* for the industrial dataset and set $\delta_d = 0$ for the simulated dataset. I do this because the simulated dataset is deployed in a single server, so the time drift will not be a problem. In addition, I use moving average to smoothen the status series for the baselines and our method. The outputs are scalar values ranging from 0 to 1. A larger value indicates higher intensity.

The overall performance is shown in Table 3.3, where I mark the smallest loss for each loss metric and dataset.
AID achieves the best performance on the industrial dataset and reduces the loss by 45.8%, 61.1%, and 33.2% in terms of cross entropy, mean absolute error, and root mean squared error. On the simulated dataset, AID achieves the best performance in terms of cross entropy and root mean squared error. Pearson correlation coefficient marginally outperforms AID on the simulated dataset. The improvement of AID on the simulated dataset is smaller than that on the industrial dataset. This is because the benchmark for simulation did incorporate very few fault tolerance mechanisms, making most of the dependencies strong. Moreover, since the service invocations of the TT benchmark are very fast, the statuses of TT's services are relatively similar, making simple baselines and our approach perform similarly.

### 3.5.3 RQ2: Impact of Different Parameter Settings

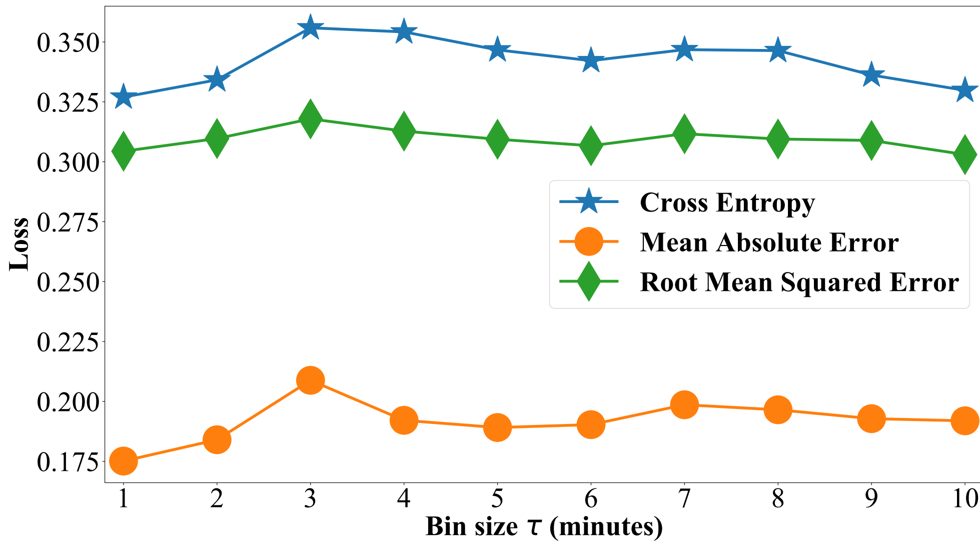



Figure 3.3: Prediction loss under different bin size $\tau$.

Since the estimated round trip time $\delta_{rtt}$ and the max time drift $\delta_d$ are minuscule, I only study the impact of the bin size $\tau$. As the range of time of the simulated dataset is small, I only study the impact of the bin size $\tau$ in the industrial dataset. In particular, I conduct experiments on with the bin size $\tau \in [1, 10](minutes)$, and keep $\delta_{rtt} = 0$ and $\delta_d = 1$ *minute*. I did not set larger bin sizes because larger bin sizes result in more coarse-grained sampling of the service status, which will add difficulty to the similarity calculation in the subsequent DSW algorithm.
Figure 3.3 shows the prediction loss under different bin size $\tau$. The x-axis denotes the bin size and the y-axis shows the three loss metrics. The results indicate that the impact of different bin sizes in a reasonable range is small, but $\tau = 1$ *minute* gives the best performance on the industrial dataset.

### 3.5.4 RQ3: Impact of Different Similarity Measures

Table 3.4: The impact of different similarity measures

| Dataset /Bin size | Method | Metric | | |
|---|---|---|---|---|
| | | CE | MAE | RMSE |
| TT /1min | $\text{AID}_{DSW}$ | 0.4562 | 0.3435 | 0.3859 |
| | $\text{AID}_{DTW}$ | 0.4494 | 0.3467 | 0.3832 |
| Industry /1min | $\text{AID}_{DSW}$ | 0.3270 | 0.1751 | 0.3044 |
| | $\text{AID}_{DTW}$ | 0.3584 | 0.1996 | 0.3169 |

I further study the impact of different similarity measures on both datasets. $\text{AID}_{DSW}$ denotes AID that uses the proposed DSW to measure the similarity between status series. $\text{AID}_{DTW}$ denotes AID that uses the DTW [75] to measure the similarity. I keep the bin size $\tau = 1$ *minute* and the estimated round trip

time $\delta_{rtt} = 0$ as usual. Similar to previous experiments, I set the max time drift $\delta_d = 1$ *minute* for the industrial dataset and set $\delta_d = 0$ for the simulated dataset.

Table 3.4 shows the performance of AID$_{DSW}$ and AID$_{DTW}$ on both datasets. On the industrial dataset, the proposed DSW algorithm improves the performance, but on the simulated dataset, the performance is almost the same. This is probably because the duration of spans in the simulated dataset is too small so that the effect of directed warping is weak. The results imply that the proposed DSW algorithm works better in real-world cloud environments.

### 3.5.5 RQ4: Efficiency

The most time-consuming operations are the candidate selection and service status generation steps because I have to iterate over all the spans in the cloud system. Theoretically, the time complexities of the candidate selection and service status generation steps are $O(S)$, where $S$ is the number of spans to process in the cloud system. In practice, the industrial dataset contains about $1.0 \times 10^{10}$ spans, so I process it with a distributed computing service in Huawei Cloud. Since the preprocessing is dynamically scheduled and mixed with other teams' tasks, I could not count the actual time spent on preprocessing. In practice, the preprocessing took less than one hour to process one day's traces. However, in extreme cases, if the number of traces are too large that exceeds the capacity of the distributed computing service, the preprocessing step can be a bottleneck for prompt intensity prediction. For the intensity prediction step, the time complexity is $O(kN^2)$, where $N = \frac{T}{\tau}$ is the number of bins and $k$ is proportional to the warping window $w$. In practice, the inten-

sity prediction step takes 155 seconds on average to process two status series both with 1440 bins on a laptop. Since the similarity calculation of different $(P, C)$ pairs are independent, I could easily parallelize the intensity prediction step to further improve the time efficiency.

## 3.6 Use Cases

In Huawei Cloud, AID has been successfully incorporated into the dependency management system that serves hundreds to thousands of cloud services. Figure 3.4 illustrates the conceptual workflow. AID processes trace logs and continuously updates the aggregated intensity in the dependency management system. The reliability engineers will categorize the intensity into different levels by referring to both the output of AID and their domain expertise. Then the dependency management system will provide reference to the engineers in optimizing dependencies and mitigating cascading failures.

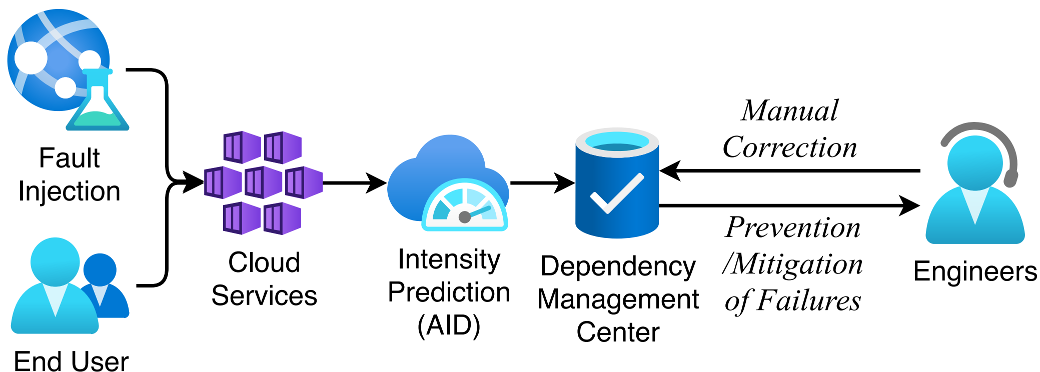



Figure 3.4: The use case of AID.

### 3.6.1 Optimization of Dependencies

In a cloud system, service failures are inevitable, but I can prevent the failures from affecting other services by optimizing improper dependencies. AID assists in the discovery of unnecessary strong dependency on critical cloud services. If a critical cloud service depends on another service with high intensity, the dependency management system will remind the engineers to check whether the dependency is necessary. If the dependency is unnecessary, the development team has to reduce the intensity of the dependency to improve the robustness of the critical cloud service. Since AID's deployment, more than ten unnecessary dependencies of critical cloud services have been discovered by AID and optimized by the development team.

### 3.6.2 Mitigation of Cascading Failures

AID also assists in the mitigation of cascading failures. During a cascading failure, AID can provide the latest intensity of dependency to OCEs, so that they can diagnose service failures efficiently. In addition, when a cascading failure occurs, OCEs can limit the traffic to critical cloud services and recover the dependencies marked as "strong" first. By doing so, the service disruption can get under control. Once a critical failure occurs, the manually confirmed "strong" dependencies will be treated with high priority. I conduct field interviews with OCEs to collect feedback. Based on the feedback, I have seen our method shedding light on reducing the impact of critical failures.

## 3.7 Discussion

### 3.7.1 Practical Usage and Perceived Limitations

#### Indirect Dependencies

In this work, I mainly considered direct dependencies, which is caused by direct service invocations. The proposed approach does not explicitly consider indirect dependencies through transitivity of service invocation because the intensity of indirect dependencies can be easily inferred from direct dependencies. In practice, the intensity of indirect dependencies can be inferred by a "cascading conduction mechanism" that if `A` intensively depends on `B` and `B` intensively depends on `C` then `A` intensively depends on `C`. The proposed approach also works well on dependencies caused by circuit breakers as long as the circuit breakers work transparently.

#### Extension of Service Status

In this chapter, I only derive three aspects of service invocations, i.e., *number of invocations*, *duration of invocations*, *error of invocations*. I utilized them because they are part of the state-of-the-art tracing system. Other aspects like the content of invocation responses can also be important to determine status. In practice, cloud providers can incorporate additional information to extend the representation of service status in their own implementation of AID.

#### Limitations on Asynchronous Invocations

Although modern tracing mechanisms can keep track of asynchronous invocations, AID may suffer from inaccuracies when

dealing with asynchronous invocations. This is because the max time drift $\delta_d$ in Algorithm 1 is hard to estimate for asynchronous service invocations. Furthermore, if the traces of synchronous and asynchronous invocations are mixed, AID may not work well since the time drift of synchronous and asynchronous invocations usually differs a lot. I leave this problem as future work.

### 3.7.2 Threat to Validity

In this work, I identified the following major threats to validity.

#### Labeling accuracy

In this chapter, I propose to measure the intensity of service dependency with AID. To evaluate the practical usage of AID, I conduct experiments on a simulated dataset and an industrial dataset. As it is a new relation between cloud services, manual labeling is needed for the evaluation. The evaluation on the industrial dataset requires engineers to manually inspect the dependencies and label the intensity of dependencies. Limited by the experience of engineers and the lack of resilience rules on the open-source benchmarks, the labels may not be 100% accurate. The fast evolution of cloud services may also change their fault tolerance mechanism, resulting in inaccurate labels. However, the engineers I invited have rich domain knowledge and are in charge of the architecture design of the cloud system of Huawei Cloud. They also discuss with each other when there are disagreements. Moreover, the labeled dependencies are the core cloud services in Huawei Cloud, so the intensity of dependencies are stable during the data collection period. I could set fixed

rules (e.g., the golden criteria for "failure" of a microservice) to make the labeling process more reproducible. The Cohen's Kappa index could also be used to measure the inter-annotator agreement (IAA) between the labelers. But most importantly, our method is unsupervised, so inaccurate labels will not affect the prediction results of the proposed method.

#### Insufficiency of the simulation

For the evaluation purpose, I deploy an open-source microservice benchmark to simulate a real cloud system. The benchmark only contains 25 microservices, which is far below the number of cloud microservices in a real cloud. Additionally, the implementation of the open-source benchmark did not fully consider the fault tolerance, resulting in only one weak dependency in the simulation. Hence, the simulated dataset may not exhibit some common attributes of a real cloud system. For example, the proportion of "strong" dependency in the simulated dataset is twice the proportion of "strong" dependency in the industrial dataset. However, the insufficiency of the simulation will not hinder the practical usefulness of AID in the real cloud system. On the contrary, as I show in Section § 3.5, the proposed method works better on the industrial dataset. The experimental results on the simulated dataset only confirm the insufficiency of the simulation.

## 3.8 Related Work

#### Cloud Monitoring

Monitoring cloud services properly with low overhead is the key to provide reliable services. Distributed Tracing, as a means of

monitoring distributed cloud services, has been widely studied in the literature. All the distributed tracing approaches can be classified as intrusive tracing and non-intrusive tracing. Intrusive tracing requires modification to application code either in run time or at compile time. Google proposes Dapper [136] to help engineers understand system behavior and reasoning about performance issues. It reduces the tracing overhead by sampling and restricting the instrumentation number. Pivot Tracing [100] provide the causal relationship of system events by combining dynamic instrumentation and happen-before join. X-Trace [46] monitors and reconstructs the whole request path from a client by modifying all the network protocols and embedding the tracing data to the package header.

Non-intrusive tracing approaches do not require code modification and usually have a lower overhead. Normally, these approaches leverage information like the system runtime logs and the source code to reconstruct the real event traces. Zhao et al. [171] propose lprof to reconstruct the execution flow of distributed systems using the runtime log of these systems. lprof conducts static analysis on the call-graph of request processing code of the system to attribute a log output to a client request. Chow et al. [33] also leverage system runtime logs to conduct performance monitoring and analysis. They propose *ÜberTrace* to reconstruct traces from the existing logs, then use *The Mystery Machine* to construct a causal model and conduct analyses. Zhao et al. [170] propose Stitch, a non-intrusive performance monitoring tool, to obtain and present information that is helpful to locate performance bugs. Stitch uses pattern matching on logs to reconstruct the hierarchical relationship of events in a system. Pensieve [163] automatically reconstructs a chain of

causally dependent events that leads to a system failure exploiting the log files and system bytecode.

#### Dependency Mining

Automatically discovering service dependencies is critical to cloud system administration and maintenance. There are two major types of dependency mining approaches, i.e., passive dependency mining and active ones. Passive dependency mining generates service dependency based purely on the runtime logs or KPIs. Shah et al. [131] propose to use Recurrent Neural Networks (RNNs) to analyze and extract dependencies in KPIs and use the discovered dependencies to identify early indicators for a given performance metric, analyze lagged and temporal dependencies, and to improve forecast accuracy. EIDefrawy et al. [43] use Transfer Entropy to passively mine the dependencies. Luo et al. [97] apply log parsing and Bayesian decision theory to estimate the direction of dependencies among services. They employ time delay consistency to reduce false dependencies. Zand et al. [156] construct a service correlation graph based on network measures and extract dependencies using hypothesis-testing. They further compute an importance metric for network's components to facilitate administration. CloudScout [151] employs Pearson Product-moment Correction Coefficient over machine-level KPIs such as TCP/UDP connection numbers and CPU utilization to calculate the similarity between different services. The similarity measure is used to cluster different services together and to conduct VM consolidation based on the service clusters. Unlike all these approaches that mostly use physical machine metrics to infer service dependencies, our method is designed for the emerging microservice

architecture and utilizes the trace logs that directly record service invocations.

Active dependency mining requires modification to services. Ma et al. propose GMAT [99], which generates service dependencies in the microservice architecture leveraging the reflection feature of Java and visualizes the dependencies to engineers. Rippler [157] extracts the dependencies by randomly injecting temporal perturbation patterns in request arrival timings for different services and investigates the propagation of the patterns. Wang et al. [141] constructs a service knowledge graph using real-time measures, operational metadata, and business features. They propose new metrics to measure the popularity of services based on their dependencies. Novotny et al. [117] focus on mining dependencies on the highly dynamic mobile networks. They use local monitors to collect local views of dependencies and generate a global view of dependency on demand.

#### Time Series Similarity

One important task in time series data mining is to measure the similarity between two time series. Similar to human intuition, the similarity measure is usually based on the similarity between the shapes of two time series [45]. In particular, the similarity measure of time series should be consistent with human perception and intuition and abide the following properties [44, 124]:

- It should resemble human intuition and identify perceptually similar datasets, even if they are not identical mathematically.
- It should not be restricted to particular time series or constraints.

- It should be robust to noise, distortions and set of transformations like amplitude, scaling, temporal warping.
- It should be able to capture global and local similarities.

Dynamic time warping (DTW) [126] is a widely-used similarity measure when two time series have the same overall component shapes but are not aligned on the timeline. It attempts to align two time series along a timeline by distorting the timeline for one time series so that its converted form is better aligned with the second time series. DTW was initially used in speech recognition applications [126] and extended and optimized by many works [20, 113, 116].

---

□ **End of chapter.**

# Chapter 4

# Self-adaptive Microservice Resilience Testing

## 4.1 Introduction

Modern online services are moving towards the microservice architecture [108], where a monolithic application is split into fine-grained, independently-managed microservices. The microservice architecture exhibits three prominent attributes. First, a microservice system is highly distributed [9] and contains a large number of microservices. For example, the system of Netflix contains hundreds to thousands of microservices [65]. Second, new features and updates are delivered continuously and frequently [55], making microservices dynamic. Last, the failures resulting in service degradation are usually cascaded due to the multi-layer deployment and inter-service dependencies architecture [91, 149, 159].

Resilience [127], i.e., the ability to maintain performance at an acceptable level and recover the service back to normal under service failures, is essentially a desired ability of microservices. Figure 2.5 illustrates the request throughput of a service during the normal (green) and the failure (red) period. Intuitively,

the resilience of the service is low because the failure causes service degradation, reflected by the throughput decrement. Since faults and failures are unavoidable [68, 91], test engineers conduct resilience tests on microservices to ensure service reliability. All new or updated microservices need to pass a lot of resilience tests before their deployments. Specifically, test engineers purposefully inject failures into the system to discover flaws [65,106]. Improvements on architectural design are then adopted according to the test results.

The current practice [83] for resilience testing involves manually setting resilience rules, including the concerned failure types, the metrics to monitor, the measure of degradation, and the criteria for passing or failing the tests. However, as the scale and complexity of microservice systems keep growing, existing approaches suffer from scalability and adaptivity issues (Section § 4.2.1). On the one hand, manual rule identification is labor-intensive and cannot scale to the large-scale evolving services. The manual identification of the rule sets relies heavily on domain expertise to define the rule sets that can represent the degradation caused by failures. It usually takes days or weeks of discussion before the engineers reaching a consensus on the rule sets. Furthermore, the fast-evolving nature [9] of microservices requires frequent updates of rule sets, increasing the burden on test engineers. On the other hand, pass/fail criteria defined by fixed rules cannot adapt to online services' various refined resilience mechanisms. This is because microservices fail in many ways, and the manifestations of metrics are also manifold. Resilience rules usually focus on a few metrics like mean time to recover (MTTR) and set fixed thresholds for evaluation. Yet, fixed rules and thresholds cannot discriminate the subtle differ-

ence in an online service's resilience, resulting in the adaptivity issue.

This thesis empirically studies the manifestations of common failures in two different deployments of an open-source microservice system (Section § 4.2.2). One deployment employs common resilience mechanisms while the other does not. Depending on the employed resilience mechanism applied to the system, the same failure causes different degradation in system performance metrics and business metrics. System performance metrics (e.g., memory usage, network throughput) directly reflect the runtime status of the microservice system. In contrast, business metrics (e.g., response latency, mean time to recovery) reflect the quality of service from the users' aspect. By comparing the difference in failures' manifestations between the two deployments, I discover that although services' degradation is manifold, a resilient service's degradation manifests more in system performance metrics than business metrics. Therefore, our insight is that self-adaptive resilience testing can be achieved by comparing the degradation contributed by the business metrics and performance metrics. *If the degradation cannot propagate from system performance metrics to business metrics, the resilience is higher. Otherwise, the resilience is lower.*

Motivated by the findings, I present **AVERT**, the first self-AdaptiVE Resilience Testing framework for microservice systems. AVERT consists of three phases, i.e., *failure execution*, *degradation-based metric lattice search*, and *resilience indexing.* The *failure execution* comprises the *failure injection phase* and the *failure clearance phase.* Given a specified failure, an online service to test, and a predefined *load generator*, AVERT collects the service's monitoring metrics in the normal and fault-

injection period. For the *degradation-based metric lattice search*, I propose a degradation-based algorithm that ranks all the monitoring metrics according to their contributions to the overall service degradation. I construct a metric lattice from the power set (i.e., the set of all the subsets) of the monitoring metric set.The ranking is based on a degradation-based path search in the metric lattice. Lastly, for *resilience indexing*, I index the resilience in $(0, 1)$ by how much the degradation manifestation in system performance metrics propagates to the degradation manifestation in business metrics.

To show the effectiveness of AVERT, I evaluate AVERT on two open-source benchmark microservice systems: Train-Ticket [173] and Social-Network [50]. I inject failures into both systems and compare the performance of resilience testing by AVERT and several baselines. I compare the resilience index with the binary `PASS/FAIL` test results with cross entropy, root mean squared error, and mean absolute error for evaluation. The experimental results demonstrate that our proposed method accurately outputs the resilience test results and outperforms the baselines. Specifically, in terms of cross entropy, AVERT achieves the best performance of 0.1775 on the Train-Ticket benchmark and 0.1159 on Social-Network benchmark. I make the code and dataset publicly available (https://github.com/yttty/avert).

The contributions of this work are highlighted as follows:

- I are the first to identify the scalability and adaptivity issues of current industrial practice for resilience testing. Then I conduct the first empirical study on the failures' manifestations on resilient and unresilient microservice systems. The empirical study demonstrates the feasibility for self-adaptive

resilience testing.

- I propose AVERT, the first self-adaptive resilience testing framework that can automatically index the resilience of a microservice system to different failures. AVERT measures the degradation propagation from system performance metrics to business metrics. The higher the propagation, the lower the resilience.

- The evaluation on two open-source benchmark microservice systems indicates that AVERT can effectively and efficiently produce accurate test results.

## 4.2 Motivation

The research described in this thesis is motivated by the real-world requirements arising from testing the resilience of microservices in Huawei Cloud. In current industrial practice, test engineers conduct resilience tests by manual configuration and fixed rules. First, I argue that current practice is unscalable and inadaptive in testing the microservice system (Section § 4.2.1). Under the fast-evolving microservice [42], an automated approach is needed to accelerate the resilience test procedure. Then, to explore the opportunity to automate the resilience evaluation process, I study the failures' manifestations that affect the resilience of microservices (Section § 4.2.2).
Specifically, I conducted an empirical study on the following research questions (RQs):

- **RQ1**: How scalable and adaptive is the current industrial practice in conducting resilience testing of a microservice system?

- **RQ2**: What are the differences in monitoring metrics of resilient and unresilient services?

### 4.2.1 RQ1: Issues of Current Practice

In the current practice, to evaluate the resilience of an online service, test engineers will manually set a bunch of fixed rules for each service and each failure type. Setting the rules heavily depends on human expertise in the metric selection and the result analysis. It works well on traditional monolithic software, which usually fails as a whole. Nevertheless, as demonstrated below, current practice suffers from scalability and adaptivity issues when evaluating the resilience of an online service composed of multiple fast-evolving microservices.

**Scalability Issue**: The failures of microservice systems are complex and variant. Though the analysis of monitoring metrics can be automated once the resilience rule sets have been manually defined, the rule definition is still labor-intensive. Current practice cannot scale with the microservice systems.

Due to the complex dependency [149, 159] between microservices, the number of failure rule sets increases exponentially with the number of microservice architecture. In addition, the manual identification of the failure rule sets relies heavily on domain expertise to define the failure rule sets that can represent the degradation caused by failures. For example, Huawei Cloud provides many cloud services. Each cloud service comprises 26 microservices on average, while the largest cloud service has over 190 microservices. Each microservice generates over 40 monitoring metrics. Then one cloud service can have around 1040 monitoring metrics. Though the analysis of monitoring metrics can be automated once the resilience rule sets have been manually

defined, the rule definition is still labor-intensive, which requires a huge amount of time and domain expertise. In Huawei Cloud, it usually takes three man-months to tease out the resilience rule set of a single cloud service. Moreover, the fast-evolving nature [9] of microservices requires frequent updates of failure rule sets, increasing the burden on test engineers. As a result, the manual identification of failure rule sets does not scale.

**Adaptivity Issue**: Setting fixed failure rule sets for resilience evaluation is inadaptive because the boundary of "resilient" and "not resilient" is not absolute. A binary `PASS/FAIL` test result cannot depict the subtle difference in an online service's resilience.

The reasons are two-fold. First, the impact of a failure is diversiform in a microservice system. Owing to the distributed nature and complicated communication model [78], the microservices of an online service usually fail partially [48]. Second, online services adopting the microservice architecture usually employs multiple ways for fault tolerance, e.g., multiple replications and active traffic control [79]. With these fault tolerance mechanisms, the online service can be in a gray-failure status [68] instead of failing as a whole. Consequently, the current practice of defining fixed failure rule sets for evaluating resilience is inadaptive.

For example, suppose I conduct a resilience test on two versions of an online service. The passing criteria require the mean time to recovery (MTTR) to be 5 minutes, which means the microservice should recover to the normal status in 5 minutes after the failure injection. Suppose I have the throughput of a microservice's two versions A and B under the same failure. The only difference is that version A takes 5 minutes to recover,

but version B only takes 2 minutes. The resilience of version B is better than version A. However, the passing criteria will let both versions pass the resilience test. Thus, the fixed rule sets cannot satisfy the need of the microservice architecture.

**Finding 1**: Setting rule sets for resilience testing relies on manual configuration and domain knowledge, so it suffers from the scalability issue in microservice systems. The impact of failures is diversiform, the current practice cannot depict the subtle difference in an online service's resilience, which results in the adaptivity issue.

### 4.2.2 RQ2: Failures and Their Impact

In answering this research question, I first summarize the common failures and their manifestations in microservice systems. I demonstrate our insight for discriminating the monitoring metrics that pass and fail the resilience test.
Failures posing threats to the resilience of microservices are ubiquitous in the microservice architecture [152, 159], e.g., microservice bugs [173, 174], unstable message passing between microservices [70], faulty cloud infrastructure like unreliable containers, virtual machines or bare metal servers [91, 130], etc. Even normal operations taken in the cloud environment, such as software / hardware upgrades and dynamic changes to configuration files, can lead to serious service interruption [58].
To know what failures are related to microservice resilience, I quantitatively analyzed many[1] incident reports related to service resilience over the last three years in Huawei Cloud. Each

[1] I hide the precise amount as required by our industrial collaborator.

incident report contains the failure description in plain text. I asked two senior Ph.D. students familiar with the cloud computing system to go through all the incident reports and classify each failure mentioned in the incident reports with a failure level (infrastructure or container level) and a failure type (memory, network, machine, etc.). A failure in the infrastructure level may cause cascading failures at the container level, but not vise versa. All the failures that (1) happened once or more and (2) are related to service resilience are collected. Also, I consulted an experienced cloud system architect to make sure I have included all the failures related to service resilience. In the end, I get 26 failures related to service resilience, listed in Table 4.4. I divide the failures into two architectural levels and then categorize them according to the type of the failed resource. Note that I exclude software bugs since software bugs are supposed to be detected via functional tests.

To understand the service degradation caused by the failures, I conducted an empirical study on two different deployments of an open-source microservice benchmark system. One deployment is equipped with the common resilience mechanisms described below and the other deployment is not. By injecting the same failures into the two deployments, I can compare the manifestations with and without the common resilience mechanisms. Specifically, the resilience mechanisms applied are (1) two replications for each microservice, and (2) load balancing for each replication using Ingress[2]. I set up a Kubernetes cluster on one physical server with 128 GB memory and 24 CPU cores. Then I create one deployment of the Train-Ticket benchmark system [173] without the resilience mechanisms. User requests with feasible

[2] https://kubernetes.io/docs/concepts/services-networking/ingress/

parameters are generated as the load of the benchmark system. I inject failures listed in Table 4.4 into the Kubernetes cluster with ChaosBlade [4], and record how the system degrades under such failures. I employ cAdvisor [54] and Prometheus[3] to collect and visualize the monitoring metrics. After that, I repeat the steps on the other deployment equipped with the common resilience mechanisms.

Service degradation manifests the impact of the injected failures. The degraded status of the service can be measured by how much the service performance is lower than the service performance benchmark [152]. As the environment for the empirical study is simple, I simply use the service's average performance without any injected failures as the service performance benchmark in this chapter. Then the service degradation is measured by the performance difference between the normal period and the fault-injection period. In Table 4.4, the penultimate column describes the failure manifestations without applying the resilience mechanisms described below, and the last column depicts the failure manifestations with the resilience mechanisms. Depending on the employed resilience mechanism of the service, the same failure may cause different degradations.

Comparing the last two columns in Table 4.4, I can discover that although services' degradations are manifold, a resilient service's degradation manifests more in performance metrics than in business metrics. I argue that self-adaptive resilience evaluation can be achieved by measuring the degradation propagated from the system performance metrics to the business metrics. For example, a container CPU overload failure will keep one container's CPU usage at 100% for a long time, affecting the end user's

[3] https://prometheus.io/

experience. However, if there are multiple replications, the high CPU usage will impact less on the business metric, e.g., the throughput will drop less. For another example, a microservice with two active replications can quickly recover from the "container instance killed" failure. In contrast, a microservice without such a replication mechanism will take much longer to recover or even be broken. In conclusion, if the degradations of business metrics are similar to the degradations of system performance metrics, the failure's impact is propagated from the system performance metrics to the business metrics. Thus the service is less resilient. The higher the degradation propagation, the lower the resilience of the microservice system.

> **Finding 2**: Microservices have a variety of failure models, and the impact on metrics is also manifold. Service degradation is the primary manifestation of the failures' impact. The less the degradations of system performance metrics propagate to the degradations of business metrics, the higher the resilience.

In summary, the findings of RQ1 and RQ2 promise the necessity and feasibility of designing a self-adaptive framework for the evaluation of a microservice application's resilience to different failures without the manual definition of resilience rule sets.

## 4.3 Methodology

I present AVERT, a self-AdaptiVE Resilience Testing framework for microservice systems. The diversiform impact of failures in a microservice system makes self-adaption to different failures

non-trivial. This section first enumerates the properties deemed necessary for a self-adaptive resilience testing framework, which deeply root in the empirical study in Section § 4.2. I then present an overall architecture of the proposed framework of AVERT. Then I elaborate on each part of AVERT in the following subsections.

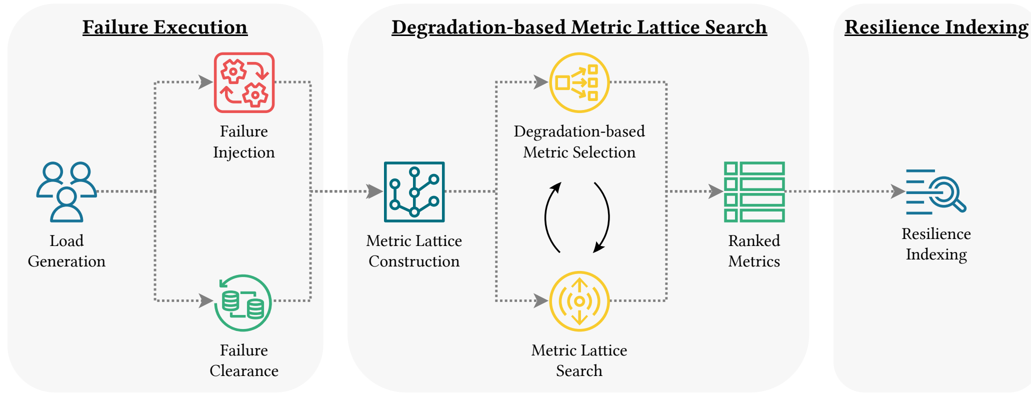



Figure 4.1: Overall framework of AVERT.

### 4.3.1 Design Objectives

Inspired by our empirical study and previous works [106, 127, 152], I consider the following desirable properties to design the self-adaptive resilience testing framework:

- *Measure performance loss*: Performance loss quantifies the degree of service degradation in monitoring metrics. It measures the cumulative degraded performance during service degradation. Performance loss can reveal the business loss in service degradation. For example, if a throughput-based service is degraded, performance loss can consider how much less data is transmitted than expected.

- *Measure disruption tolerance*: Disruption tolerance measures

the degree of degradation in system performance metrics compared with that in business metrics. A resilient microservice system shall keep the degradation within the system performance metrics, which is less impactful to users.

- *Self-adaption*: Since the failures' impact on the monitoring metrics is manifold, the method should be able to adapt to different failures and different services. It should have a small number of parameters.

### 4.3.2 Overview

The overall workflow of AVERT is illustrated in Figure 4.1. AVERT consists of three phases, i.e., *failure execution*, *degradation-based metric lattice search*, and *resilience indexing*. The *failure execution* is composed of the failure injection phase and the failure clearance phase. Given a specified failure, an online service to test, and a predefined load generator, AVERT collects the service's monitoring metrics in the normal and failure-injection period. For the *degradation-based metric lattice search*, I propose a degradation-based metric selection algorithm to search the metric lattice, a partially ordered data structure constructed from the power set (i.e., the set of all subests) of the entire monitoring metric set. Figure 4.2 illustrates an example metric lattice. Specifically, AVERT first organizes all subsets into a metric lattice. Then AVERT search the lattice in a degradation-based manner. In this way, the search path naturally produces a ranked list of monitoring metrics along with their contribution to the overall service degradation. Lastly, for *resilience indexing*, AVERT calculates the resilience index by how much the degradation manifestation in system performance metrics propagates

to the degradation manifestation in business metrics.
AVERT measures the performance loss (i.e., the degree of service degradation in monitoring metrics) by comparing the difference in metrics between normal and failure-injection period, and measures the disruption tolerance by ranking the monitoring metrics affected by the injected failures and quantifying the degradation propagated from the performance metrics to the business metrics. The less the degradation of system performance metrics propagate to the degradation of business metrics, the higher the resilience. As AVERT does not need any knowledge about the system architecture or the adopted resilience mechanisms, it can easily adapt to various microservice systems. Moreover, once the types of failures are determined, test engineers do not need to define resilience testing rules for different microservice systems, which means that resilience testing with AVERT can easily scale.

### 4.3.3 Failure Execution

The *failure execution* consists of the *failure injection* and the *failure clearance* phases. First, a test engineer needs to provide a *load generator* to the online service being tested. The load generator should mimic real-world requests from users. Second, the test engineer selects a list of failures to test. The failure can be injected in the infrastructure level or in the container level. Then, AVERT automatically generates a failure injection pipeline. For each failure, AVERT injects the failure, clears the failure, and collects the service's monitoring metrics in the meanwhile. The duration of failure injection and failure clearance are the same for each failure.
During the two phases, AVERT collects two types of metrics,

*i.e.*, business metrics and system performance metrics. Suppose $\mathcal{B}$ is the business metrics set and $\mathcal{P}$ is the system performance metrics set in the system, I denote the set of all the business metrics and system performance metrics as $\mathcal{M} = \mathcal{B} \cup \mathcal{P}$. Suppose $card(\mathcal{M}) = M$, I index all the monitoring metrics from $m_1$ to $m_M$. In other words, $\mathcal{M} = \{m_1, m_2, \cdots, m_M\}$. Thus, for any $i \in [1, M]$, either $m_i \in \mathcal{B}$ or $m_i \in \mathcal{P}$ stands. I denote the monitoring metrics during the failure injection period as $\mathcal{M}^f = \{m_1^f, m_2^f, \cdots, m_M^f\}$. For each $i$, $m_i^f$ is a univariate time series denoting the monitoring metrics during the failure injection period. Likewise, I denote the monitoring metrics during the failure clearance (normal) period as $\mathcal{M}^n = \{m_1^n, m_2^n, \cdots, m_M^n\}$. Also, for each $i$, $m_i^n$ is a univariate time series denoting the monitoring metrics during the failure clearance (normal) period. AVERT ensures that $length(m_i^f) = length(m_i^n) = T$.

### 4.3.4 Degradation-based Metric Lattice Search

**Algorithm 2:** Degradation-based Metric Lattice Search

**Input:** The monitoring metrics $\mathcal{M} = \{m_1, m_2, \cdots, m_M\}$; The monitoring metrics during the failure injection period $\mathcal{M}^f = \{m_1^f, m_2^f, \cdots, m_M^f\}$; The monitoring metrics during the failure clearance period $\mathcal{M}^n = \{m_1^n, m_2^n, \cdots, m_M^n\}$

**Output:** An ranked list of metrics $\hat{\mathcal{M}}$

1 Construct the metric lattice
2 $\mathcal{L} = EmptyList()$
3 $M = \mathcal{M}$
4 **while** $M \neq \emptyset$ **do** // Metric Lattice Search
5 | $cmax, m_{imax} =$ `MetricSelection`$(M)$
6 | $\mathcal{L}.append((cmax, m_{imax}))$
7 | $M = M - \{m_{imax}\}$
8 **end**
9 **return** $\mathcal{L}$

The degradation-based metric lattice search aims at comparing and ranking the contribution of different monitoring metrics to the overall service degradation caused by the failure. Algorithm 2 shows the procedure for degradation-based metric lattice search. I introduce the degradation-based metric lattice search from the following three aspects, i.e., *metric lattice construction*, *degradation-based metric selection*, and *metric lattice search*.

**Metric Lattice Construction**

Formally, a lattice is a partially ordered set in which each pair of elements has a least upper bound and a greatest lower bound. Inspired by the frequent itemset mining algorithm [60, 110], I construct a lattice from the power set (i.e., the set of all subests) of the entire monitoring metric set. Let each subset of the entire monitoring metrics set $\mathcal{M}$ be a node in the metric lattice $\mathcal{L}$. I define the order between any two nodes of the lattice as the subset-superset relation. Formally, suppose I have $a, b \subseteq \mathcal{M}$ and $a \neq b$, then $a \subset b (\subseteq \mathcal{M})$ (in the monitoring metric set) indicates $a \leq b$ ($b \rightarrow b$ in the metric lattice). The metric lattice will be searched starting from the node $\mathcal{M}$ in later steps.
Figure 4.2 illustrates an example metric lattice constructed from $\mathcal{M} = \{m_1, \cdots, m_4\}$. Each directed edge indicates a subset-superset relation, pointing from the metric superset to the metric subset. Note that I set the number of monitoring metrics as a small value, 4, for a clear illustration.

**Degradation-based Metric Selection**

As mentioned in Section § 4.2.2, service degradation is the primary manifestation of the failures' impact. I propose to measure the service degradation via the fluctuation of system per-

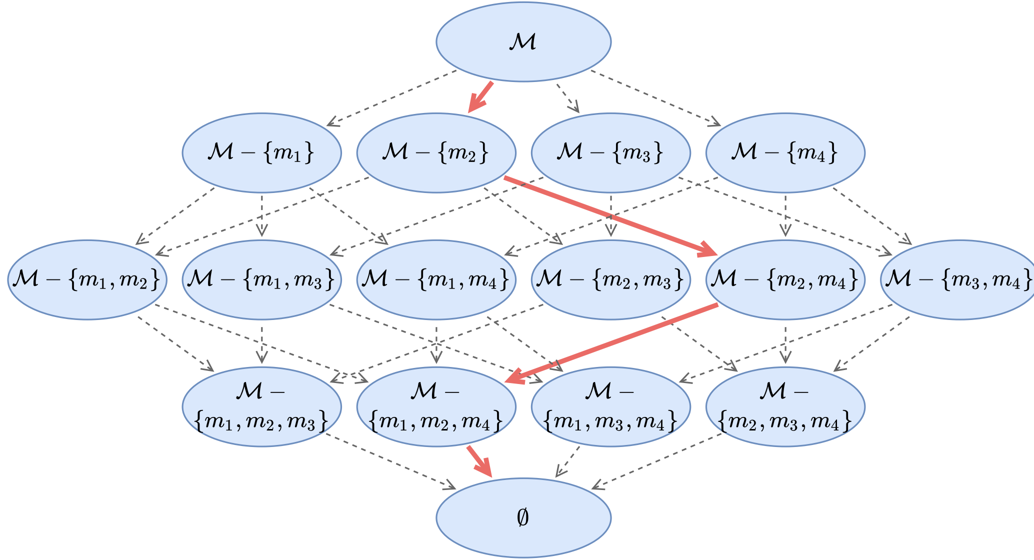



Figure 4.2: An example metric lattice constructed from $\mathcal{M} = \{m_1, \cdots, m_4\}$. I set number of monitoring metrics as a small value, 4, for a clear illustration. The path of all solid red edges forms a ranked list.

formance metrics and business metrics. If the degradation of system performance metrics cannot propagate to the degradation of business metrics, the resilience is higher. Otherwise, the resilience is lower. Therefore, the key is to select the monitoring metric that contributes most to the overall service degradation among all the monitoring metrics.

Algorithm 3 shows how to select the metric that contributes most to the overall service degradation. Expressly, given a subset of the entire monitoring metrics set $\mathcal{M}' \subseteq \mathcal{M}$ and the metrics during the failure injection and clearance period $\mathcal{M}'^f$ and $\mathcal{M}'^n$. AVERT first measures the performance difference $\delta_i$ of each monitoring metric $m_i$ by calculating the absolute difference of monitoring metrics between the failure injection period and the failure clearance period. All metrics' performance difference naturally forms a performance difference matrix $\mathbf{D}$. By reducing $\mathbf{D}$ to 1 dimension with Principal Component Anal-

**Algorithm 3:** Degradation-based Metric Selection

**Input:** The monitoring metric subset $\mathcal{M}'$; The monitoring metrics during the failure injection period $\mathcal{M}'^f$; The monitoring metrics during the failure clearance period $\mathcal{M}'^n$

**Output:** The metric $m_i \in \mathcal{M}'$ where $m_i$ contribute most to the overall service degradation

1 **Function** `MetricSelection`($\mathcal{M}'$, $\mathcal{M}'^f$, $\mathcal{M}'^n$):

2 $\quad T$ = length of the monitoring metrics

3 $\quad \mathbf{D} = []$

4 $\quad$ **for** $m_i \in \mathcal{M}'$ **do**

5 $\quad\quad$ // Compute the performance difference of each individual metric

6 $\quad\quad$ **for** $t = 1 \ldots T$ **do**

7 $\quad\quad\quad \delta_i(t) = |m_i^f(t) - m_i^n(t)|$

8 $\quad\quad$ **end**

9 $\quad\quad \hat{\delta}_i = \delta_i - \bar{\delta}_i$ // Normalize $\delta_i$

10 $\quad\quad \mathbf{D} = [\mathbf{D}; \hat{\delta}_i]$ // Concatenate the normalized performance difference

11 $\quad$ **end**

12 $\quad \delta_{PC1} = \texttt{PCA}(\mathbf{D}, dim = 1)$ // Reduce to one dimension via Principal Component Analysis

13 $\quad$ // Select the metric that contribute most to the performance difference

14 $\quad$ **for** $\hat{\delta}_i \in \mathbf{D}$ **do**

15 $\quad\quad c_i = \texttt{Contribution}(\delta_{PC1}, \hat{\delta}_i)$

16 $\quad$ **end**

17 $\quad cmax = \max(c_i)$

18 $\quad imax = \arg\max_i(c_i)$

19 $\quad$ **return** $cmax$, $m_{imax}$

20 **End**

ysis (PCA) [60, 139], AVERT gets $\delta_{PC1}$, a sequence of length $T$. Let $\delta_{PC1}$ represent the overall service degradation caused by the injected failure. AVERT computes the contribution of each metric to the overall degradation via a contribution measure `Contribution()`. The higher the similarity between $\delta_{PC1}$ and $\delta_i$, the larger the `Contribution()` outputs. `Contribution()` can be a correlation coefficient (e.g., Pearson correlation coefficient) or any other distance measure (e.g., Euclidean distance or dynamic time warping distance) deemed appropriate. I discuss the selection of `Contribution()` in Section § 4.4.3. In the end, the function returns the metric $m_{imax}$ that contributes most to the overall service degradation, along with its contribution $cmax$. $m_{imax}$ will guide the metric lattice search, and $cmax$ will be used to calculate resilience in Section § 4.3.5.

#### Metric Lattice Search

The metric lattice search is straightforward with the degradation-based metric selection. As shown in Algorithm 2, the search starts from the node of the entire metric set $\mathcal{M}$. At each node $\mathcal{M}' \in \mathcal{M}$, AVERT selects the metric $m_{imax}$ that contributes most to the service degradation on the metric set $\mathcal{M}'$. AVERT then eliminates the monitoring metric $m_{imax}$ from $\mathcal{M}'$ and proceed to the next node until all the monitoring metrics are eliminated. The path from $\mathcal{M}$ to $\emptyset$ naturally forms an ordered list of all the monitoring metrics $m$ and their contribution value $c$, denoted as $\mathcal{L}$. For example, in Figure 4.2, the path of all solid red edges forms the ordered list $[m_2, m_4, m_1, m_3]$.

### 4.3.5 Resilience Indexing

Section § 4.2.2 finds that the resilience can be inferred from whether the degradation manifestation in system performance metrics propagates to the degradation manifestation in business metrics. AVERT calculates the degradation in system performance metrics and business metrics with Equation 4.1 and Equation 4.2 respectively.

$$D_{\mathcal{P}} = \sum_{m_i \in \mathcal{P}} \frac{c_i}{\log_2(rank(m_i; \mathcal{L}) + 1)} \tag{4.1}$$

$$D_{\mathcal{B}} = \sum_{m_i \in \mathcal{B}} \frac{c_i}{\log_2(rank(m_i; \mathcal{L}) + 1)} \tag{4.2}$$

In the end, AVERT utilizes the sigmoid function to map the difference between $\mathcal{B}$'s and $\mathcal{P}$'s contribution to a float value $r \in (0, 1)$, as shown in Equation 4.3.

$$r = \frac{1}{1 + e^{D_{\mathcal{B}} - D_{\mathcal{P}}}} \tag{4.3}$$

$r$ measures the "degradation propagation ratio" from the system performance metrics to the business metrics. The larger $r$ is, the higher the resilience is.

## 4.4 Evaluation

This section introduces the experiment settings, including the dataset, parameter settings, evaluation metrics, and baseline methods, and then analyze the experimental results. I aim to investigate the following research questions.

- **RQ3.** How effective is AVERT in evaluating the resiliency

of online services?

- **RQ4.** How do different contribution measures affect the performance of AVERT?
- **RQ5.** How efficient is AVERT?

### 4.4.1 Experiment Settings

#### Dataset

To illustrate the practical effectiveness of aid, I carried out experiments on two simulated datasets. Since there is no existing dataset for resilience testing, I deployed two benchmark microservice systems and conducted resilience tests on then. I collect the monitoring metrics and manually labeled the resilience testing results to build the datasets. I release both datasets in the paper to promote future research in this field.

**Dataset I**: For collecting the first dataset, I deploy Train-Ticket [173], an open-source microservice benchmark system, with "Kubernetes", a popular microservice orchestrator. Train-Ticket is a web-based ticketing system with 15 microservices. Users can search for tickets, reserve tickets, order food and insurance, and pay for reserved tickets through the system. For load generation, I develop a request simulator to simulate the access of ordinary users to the ticketing system. The simulator will log in to the system, search for tickets, order tickets, foods, and insurance according to the search results, and make the payment. I inject 24 failures listed in Table 4.4 into the benchmark microservice system with ChaosBlade. (I omit the three failures in "Infrastructure - Machine" as I do not have access to the physical server.) For each failure, the failure injection period

and failure clearance period both lasts for 10 minutes, during which the request simulator continuously send external requests to the system. cAdvisor [54] is used to collect 13 system performance metrics. The system performance metrics cover all major aspects of the microservice system, including CPU, file system, memory, network. As for the business metrics, I use Jaeger, an open-source tracing framework, to trace all the API calls. Following the existing research [149], I calculate the average response time and the request error rate in seconds as the business metrics. The dataset is referred to as "Train-Ticket" in Table 4.1.

**Dataset II**: To illustrate the universality of our method, I collected another dataset on another widely-used microservice orchestrator "docker-compose" to evaluate AVERT. Compared with "Kubernetes", "docker-compose" runs on a single server for microservice orchestration. The resilience of a microservice system managed by "docker-compose" depends more on the microservice developer. I select Social-Network [50] as the benchmark microservice system. It is a Twitter-like social media platform composed of multiple microservices, including 12 microservices for processing user requests and 13 microservices for data storage. The business metrics of the dataset include the average response time and the request error rate. As "docker-compose" employs few resilience mechanisms at the infrastructure level, I do only inject failures at the container level. Similarly, I inject 10 failures using ChaosBlade. Each failure lasts for 5 minutes since the Social-Network benchmark responses faster than Train-Ticket. This data set is referred to as "Social-Network" in Table 4.1.

**Manual labeling**: As AVERT is unsupervised, labels are only

Table 4.1: Dataset Statistics

| **Dataset** | $\lvert\mathcal{B}\rvert$ | $\lvert\mathcal{P}\rvert$ | *#Microservices* | *#Failures* |
|---|---|---|---|---|
| *Train-Ticket* | 30 | 209 | 15 | 24 |
| *Social-Network* | 50 | 325 | 25 | 10 |

for evaluation. For both datasets, I adopt the criteria in Section § 4.2 for resilience, i.e., whether the degradation in system performance metrics propagates to the degradation in business metrics. I invited two senior Ph.D. students to inspect the collected monitoring data and give `PASS/FAIL` conclusions on each injected failure. I use `PASS/FAIL` to make it easier to reach an agreement. In case of disagreement on labels, I will invite a third student to judge and label the case. I also invited the test engineers from Huawei Cloud to investigate the monitoring data and draw `PASS/FAIL` conclusions according to the existing resilience rule sets in Huawei Cloud. Finally, I convert `PASS` to 1 and `FAIL` to 0 so that they can be effectively compared with the computed resilience values.
Table 4.1 shows the statistics of the two datasets. As the number of monitoring metrics varies with the microservice system architecture, I list the number of system performance metrics (denoted as $|\mathcal{P}|$) and business metrics (denoted as $|\mathcal{B}|$) in Table 4.1. "# Microservices" and "# Failures" means the number of microservices, and the number of injected failures in the dataset, respectively.

#### Baselines

Since there is no existing public work that measures the resilience of a microservice system, I adopt the common predictive methods as baselines, i.e., Support Vector Machine Classi-

fier [37] (denoted as SVC), Random Forest [23] (denoted as RF), and Extra Trees [53] (denoted as ET). Specifically, let the input $X_t$ be all the monitoring data at time $t$, and the output $y_t$ be whether $t$ is in the failure injection period, I train the baseline models with all $X_t$ and $y_t$. I use the rank of feature importance (for ET and RF) and the rank of coefficient (for SVC) as the ordered sequence of the monitoring metrics. For the baselines, I directly use the implementation from the Python package `sklearn`[4]. Then I set $c_i = 1$ then calculate the resilience index the same way as in Section § 4.3.5.

#### Evaluation Metrics

Since the labels are binary values and the prediction results are float values, I employ Mean Absolute Error $MAE = \frac{\sum_{i=1}^{N} |y_i - p_i|}{n}$, Root Mean Squared Error $RMSE = \sqrt{\frac{\sum_{i=1}^{N} (y_i - p_i)^2}{N}}$, and Cross Entropy $CE = \frac{1}{N} \sum_{i=1}^{N} -[y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$ to evaluate the effectiveness of AVERT in predicting the resilience. Specifically, cross entropy calculates the difference between the label and the predicted probability distribution. Mean absolute error and root mean square error measurement absolute error and root mean square error. Lower CE, MAE, and RMSE values indicate better prediction results.

#### Experimental Environments

I deployed the Train-Ticket benchmark in a Kubernetes cluster of two physical servers. Both servers have 128 GB RAM and 24 CPU cores. The Social-Network benchmark was deployed in a t2.2xlarge EC2 instance of AWS with 8 GB RAM and 8 CPU

[4] https://scikit-learn.org/

cores. For both datasets, I run the degradation-based metric lattice search and the resilience indexing on a laptop with 4 Intel CPU cores and 8 GB RAM.

### 4.4.2 RQ3: Effectiveness

To study the effectiveness of AVERT, I compare its performance with the baseline models on both the simulated dataset and the industrial dataset collected from Huawei Cloud. For the contribution measure of AVERT, I employ dynamic time warpping (dtw) [75] algorithm. Specifically, for the parameters of dtw, I set the warpping window to be 5 seconds (for Train-Ticket) and 2 seconds (for Social-Network), and use the square of the absolute difference as the distance measure. I do this because the Social-Network benchmark is deployed in a single server, and it responds faster than the Train-Ticket benchmark. In addition, I use moving average of a window size 3 to smoothen the monitoring metrics for the baselines and our method. The outputs are scalar values in the range $(0, 1)$. A larger value indicates higher resilience. The overall performance is shown in Table 4.2, where I mark the smallest loss for each loss metric and dataset.

Table 4.2: Performance Comparison of AVERT on Two Datasets

| **Method** | *Train-Ticket* | | | *Social-Network* | | |
|---|---|---|---|---|---|---|
| | *CE* | *MAE* | *RMSE* | *CE* | *MAE* | *RMSE* |
| SVC | 0.8864 | 0.4875 | 0.5594 | 0.7483 | 0.4426 | 0.5165 |
| RF | 0.6973 | 0.4259 | 0.5005 | 0.5646 | 0.3787 | 0.4416 |
| ET | 0.8766 | 0.4682 | 0.5470 | 0.6546 | 0.4199 | 0.4893 |
| **AVERT** | **0.1775** | **0.1572** | **0.1842** | **0.1159** | **0.1078** | **0.1203** |

AVERT achieves the best performance on both the datasets and reduces the loss by 79.5%, 77.6%, and 72.7.2% in terms of cross

entropy, mean absolute error, and root mean squared error. The improvement of AVERT on the Train-Ticket dataset is smaller than that on the Social-Network dataset. This is because the Social-Network benchmark incorporates very few fault tolerance mechanisms, making it fail many resilience tests.

### 4.4.3 RQ4: Ablation Study

I study the impact of different contribution measures on the performance of AVERT. In particular, I conduct experiments on with varying contribution measures, i.e., Euclidean Distance (denoted as "AVERT-euc"), Pearson Correlation (denoted as "AVERT-corr"), Complexity Invariant Distance [17,18] (denoted as "AVERT-cid"), and keep other parameters identical. I represent our method as "AVERT-dtw" in the table. Table 4.3 shows the prediction loss under different contribution measures. I marked models with the best performance in terms of CE, MAE, and RMSE. The results indicate that the impact of different contribution measures in a reasonable range is small, but "AVERT-dtw" gives the overall best performance on the Train-Ticket dataset.

Table 4.3: Ablation Study of AVERT on Two Datasets

| **Method** | *Train-Ticket* | | | *Social-Network* | | |
|---|---|---|---|---|---|---|
| | *CE* | *MAE* | *RMSE* | *CE* | *MAE* | *RMSE* |
| AVERT-euc | 0.3379 | 0.2735 | 0.3067 | 0.1874 | 0.1655 | 0.1905 |
| AVERT-corr | 0.2320 | 0.1985 | 0.2296 | 0.2532 | 0.2148 | 0.2449 |
| AVERT-cid | 0.1784 | 0.1589 | **0.1810** | 0.3131 | 0.2542 | 0.2933 |
| **AVERT-dtw** | **0.1775** | **0.1572** | 0.1842 | **0.1159** | **0.1078** | **0.1203** |

### 4.4.4 RQ5: Efficiency

The efficiency of AVERT is composed of three parts, including (1) the time spent on failure execution, (2) the time spent on the degradation-based metric lattice search, and (3) the resilience indexing. As the frequency of the external request (or the load generator) is unique to each service, the required duration of failure injection varies dramatically. Therefore, the time spent on failure execution are omitted when I discuss efficiency. Among the remaining two phases, the most time-consuming phase is the degradation-based metric lattice search. Theoretically, the time complexity of Algorithm 3 depends on the length $T$ of the monitoring metrics, the number of monitoring metrics $|\mathcal{M}|$, and the time complexity of `Cont()`. Since $|\mathcal{M}| << T$ in practice, I treat $|\mathcal{M}|$ as a constant. The computation of performance difference costs $O(T)$, and the dimension reduction with PCA costs $O(T^3)$. As dynamic time warpping can be easily parallelized, I treat the time complexity of `Contribution()` as $O(T)$. Merging together, upper bound of the time complexity is $O(T^3)$. Considering the average time of failure injection is usually several hours, a time complexity of $O(T^3)$ will not be a problem. On average, the latter two phases take 302 seconds to process a failure test case of $T = 1200$ on a laptop.

## 4.5 Discussion

### 4.5.1 Threats to Validity

#### Labeling accuracy

To evaluate AVERT, I conduct experiments on two simulated datasets. The evaluation on both datasets requires Ph.D. stu-

dents to inspect the monitoring metrics and label the resilience test results. Limited by their knowledge, the label may not be 100% accurate. However, the test engineers I invited have rich domain knowledge and are in charge of the resilience assurance of the cloud services of Huawei Cloud. Moreover, since the benchmark systems are simple and the resilience mechanisms and deployment environment of the benchmark systems are clear to all the labelers, the resilience test results are straightforward. I could use existing resilience testing results that are based on fixed rules (e.g., the rules described in Section § 4.2.1) to make the labeling process more reproducible. The Cohen's Kappa index could be used to measure the inter-annotator agreement (IAA) between the labelers. But most importantly, our method is unsupervised, so inaccurate labels will not affect the prediction results of the proposed method.

#### Insufficiency of the simulation

For the evaluation, I deploy open-source benchmark microservice systems to simulate real cloud services. The numbers of microservices are small, and the implementation of the open-source benchmark did not fully consider fault tolerance, resulting in poor resilience in the simulation. Hence, the simulated dataset may not exhibit some common attributes of a real online service. After consulting with the test engineers in Huawei Cloud, I deployed one benchmark on bare metal machines and another benchmark on AWS EC2 instances to increase the representativeness of the datasets. Moreover, I deploy the benchmark microservice systems with two widely-used microservice orchestrators to show the practical usefulness of AVERT in different environments. Last but not least, conducting simulation on pro-

duction or large-scale microservice systems could alleviate the insufficiency of simulation.

## 4.6 Related Work

**Resilience Testing of Online Services**

Modern online services have been moving towards the microservice architecture. To ensure the ability of the system to minimize the impact of potential failures, considerable attention has been paid to resilience testing of microservices, including model-based resilience representation and analysis [103, 152], non-intrusive and automated fault injection [6, 16, 65, 96], scalability resilience testing [2]. [103] used the PRISM probabilistic model checker to analyze the behavior of the Retry and Circuit Breaker resiliency patterns. [152] proposed a Microservice Resilience Measurement Model (MRMM) represent the resilience requirements on MSA Systems. [6] proposed a lineage-driven fault injection approach to infer whether injected faults can prevent correct outcomes by exploring historical data lineage and satisfiability testing. [65] presented a non-intrusive resilience testing framework that injects faults by manipulating the network packets communicated between microservices. [2] simulated delay latency injection to assess the fault scenario's impact on the cloud software service's scalability resilience. [2, 6, 65] all require test engineers to write test descriptions and manually check assertions. Netflix proposed chaos engineering [16] to randomly inject faults in the system. A resent study [96] propose to automatically generate resilience test cases by inferring whether the injected faults can result in severe failures. However, the existing approaches either highly rely on human labors or his-

torical cases, making them less practical in cloud-scale service systems with high dynamism and complex failure models. In contrast, AVERT is an adaptive and efficient approach to evaluate the resilience of online services, which is dispensed with historical testing cases.

#### Combination Searching

Many combination searching techniques [12, 24, 57] have shown its promise in reducing information redundancy and enhancing the performances of data-driven models. The combination searching approaches fall into three categories: score-based [24] and embedding-based [40, 66, 77, 101], and wrapper-based [19, 112, 114, 125]. Specifically, filter approaches attempt to construct a proxy measure to score a feature combination. For example, Kendall's $\tau$ coefficient [24] uses a non-parametric hypothesis test to measure the ordinal association between two features to identify similar features. Embedding-based approaches perform combination selection during the model construction, such as [77], which shrinks many of regression coefficients to zero and remains non-zero regression coefficients for downstream models. Wrapper-based approaches uses different combinations of features to train the same downstream model, whose performances of hold-out testing can be leveraged to qualify the usefulness of the combination. Some heuristic approaches [19, 125] have been developed to narrow the searching space due to the high searching complexity. AVERT employs a wrapper-based method and overcomes the demerits of high computation cost and over-fitting by proper pruning.

---

□ **End of chapter.**

Table 4.4: Typical faults and the corresponding degradation with and without the resilience mechanisms mentioned in § 4.2.2

| **Level** | **Type** | **Failure** | **Degradation w/o resilience mechanisms** | **Degradation w/ resilience mechanisms** |
|---|---|---|---|---|
| *Infrastructure* | *CPU* | CPU overload | High physical CPU usage, slow response speed | Decreased but acceptable response speed |
| | *Memory* | Memory overload | High physical memory usage, slow response speed | Decreased but acceptable response speed |
| | *Storage* | Disk partition full | Unable to read/write, internal error (500) | Response normally |
| | | High disk I/O throughput | High physical I/O throughput | Response normally |
| | | High disk I/O latency | Slow I/O | Response normally |
| | | High disk I/O error | Slow and erroneous I/O | Response normally |
| | | Block storage service stopped | I/O rate drop to zero, internal error (500) | Response normally |
| | *Network* | High HTTP packet loss rate | High retransmission rate | Response normally |
| | | High HTTP request latency | High connection latency, slow response | Return to normal response speed shortly |
| | | TCP disconnection | Connection error, disconnected | Return to normal response speed shortly |
| | | Port in use | Connection initialization error | (same as left) |
| | | NIC down | Connection error, unreachable network | (same as left) |
| | | Running out of network connections | Unable to create new connections | Response normally |
| | *Process* | Critical process killed | Unresponsive process, existing connection down | Response normally after some time |
| | *Machine* | Unplaned reboot | Machine offline | Response normally after some time |
| | | Power outage | Machine offline | Response normally after some time |
| | | System time shift | Process error | Automaitc time correction |
| *Container* | *CPU* | Container CPU overload | High container CPU usage, slow response speed | Decreased but acceptable response speed |
| | *Memory* | Container memory overload | High container memory usage, slow response speed | Decreased but acceptable response speed |
| | *Network* | Container TCP disconnection | Connection error within container | Return to normal response speed shortly |
| | | Unreachable network | Network unreachable error in container | Return to normal response speed shortly |
| | | Container port in use | Connection initialization error | (same as left) |
| | | Container network packet loss | High retransmission rate | Return to normal response speed shortly |
| | | Container virtual NIC down | Connection error | Return to normal response speed shortly |
| | *Storage* | Container disk full | Unable to read/write, internal error (500) | Response normally after some time |
| | *Instance* | Container instance killed | Instance offline, unresponsive microservice endpoint | Response normally after some time |
| | | Container instance suspended | Instance offline, unresponsive microservice endpoint | Response normally after some time |

# Chapter 5

# Empirical Study on Alerting and Logging

## 5.1 Introduction

The boost of cloud adoption puts forward higher requirements on the reliability and availability of cloud services. Typically, cloud services are organized and managed as microservices that interact with each other and serve user requests as a whole. In a large-scale cloud microservice system, unplanned microservice anomalies happen from time to time. Some anomalies are transient, while others persist and require human intervention. If anomalies are not detected and mitigated timely, they may cause severe cloud failures and incidents, affect the availability of cloud services, and deteriorate user satisfaction [31]. Hence, prompt detection, human intervention, and mitigation of service anomalies are critical for the reliability of cloud services. To accomplish that, cloud service providers employ large-scale cloud monitoring systems that monitor the system state and generate alerts that require human intervention. Whenever anomalous states of services emerge, alerts will be generated to notify engineers to prevent service failures.

In a cloud system, an **alert** is a notification sent to On-Call Engineers (OCEs), of the form defined by the *alert strategy*, of a specific abnormal state of the cloud service, i.e., an **anomaly**. A severe enough alert (or a group of related alerts) can escalate to an **incident**, which, by definition, is any unplanned interruption or performance degradation of a service or product, which can lead to service shortages at all service levels [31]. An **alert strategy** defines the policy of alert generation, i.e., *when to generate an alert*, *what attributes and descriptions an alert should have*, and *to whom the alert should be sent*. Once an OCE receives an alert, the OCE will follow the corresponding predefined Standard Operating Procedure (**SOP**) to inspect the state of the cloud service and mitigate the service anomaly based on their domain knowledge. The *alert strategies* and *SOPs* are two key aspects to ensure a prompt and effective response to cloud alerts and incidents. In industrial practice, the two aspects are often considered and managed together because improperly designed alert strategies may lead to non-informative or delayed alerts, affecting the diagnosis and mitigation of the cloud alerts and incidents. I call the unified management of *alert strategies* and *SOPs* **alert governance**. Table 5.1 summarizes the terminologies used in this chapter.

In industrial practice, a cloud provider usually deploys a cloud monitoring system to obtain the telemetry data that reflects the running state of their cloud services [39, 90]. Multiple monitoring techniques are employed to collect various types of telemetry data, including the performance indicators of the monitored service, the low-level resource utilization, the logs printed by the monitored service, etc. For normally functioning services, it is assumed that their states, as well as their telemetry data,

Table 5.1: The Terminology Adopted in This Chapter.

| Term | Explanation |
| --- | --- |
| **Anomaly** | A deviation from the normal state of the cloud system, which will possibly trigger an alert. |
| **Alert** | A notification sent to On-Call Engineers (OCEs), of the form defined by the alert strategy, of a specific anomaly of the cloud system. |
| **Incident** | Any unplanned interruption or performance degradation of a service or product, which can lead to service shortages at all service levels [31]. |
| **Alert Strategy** | The policy of alert generation, including *when to generate an alert*, *what attributes and descriptions an alert should have*, and *to whom the alert should be sent*. |
| **SOP** | A predefined Standard Operating Procedure (SOP) to inspect the state of the cloud system and mitigate the system abnormality upon receiving an alert. The operations can be conducted by OCEs or automatically. |
| **Alert Governance** | The unified management of *alert strategies* and *SOPs*. |

will be stable. For a service that will fail soon, its telemetry data will fluctuate from the normal state [69,142]. Hence, cloud providers typically conduct anomaly detection on the telemetry data to detect the deviation from the normal state. If an anomaly triggers an alert strategy, an alert will be generated, and the cloud monitoring system will notify OCEs according to the configuration of the alert strategy.

The configuration of alert strategies is empirical, which heavily depends on human expertise. Since different cloud services exhibit different attributes and serve different purposes, their alert strategies vary significantly. In particular, the empiricalness of alert strategies results from two aspects of cloud services. On the one hand, a cloud service's abnormal state may differ because

each cloud service implements its own business logic. There is no one-fits-all rule for anomaly detection on cloud services, i.e., *when to generate an alert*. For example, network overload is a crucial anomaly for a virtual network service. However, high connection number becomes a real issue for a database service. On the other hand, the attributes of an alert that helps the manual inspection and mitigation of the abnormal state, e.g., the location information and the free-text title that describes the alert, are also service-specific and lack comprehensive guidelines. In other words, "*what attributes and descriptions an alert should have*" also depends on human expertise. For example, the title "Instance $x$ is abnormal" is non-informative. In summary, the configuration of alert strategies, as a precursor step for human intervention in cloud anomalies, is an empirical procedure.

Manually-configured alert strategies are flexible but can also be ineffective (e.g., misleading, non-informative, and non-actionable) when the engineer is inexperienced or unfamiliar with the monitored cloud service. The ineffectiveness of alerts becomes anti-patterns that hinder the OCEs' diagnosis, especially for inexperienced OCEs. The anti-patterns of alerts, which I will elaborate in Section § 5.2, will frustrate OCEs and deteriorate cloud reliability in the long term.

In this thesis, I conduct the first empirical study on the industrial practice of alert governance in Huawei Cloud[1]. The cloud system considered in this study consists of 11 cloud services and 192 cloud microservices. The procedure of our study includes 1) a quantitative assessment of over 4 million alerts in the time

[1]Huawei Cloud is a global cloud provider and ranked fifth in Gartner's report [21] on the global market share of *Infrastructure as a Service* in 2020.

range of two years to identify the anti-patterns of alerts; 2) interviews with 18 experienced on-call engineers (OCEs) to confirm the identified anti-patterns and summarize the current practice to mitigate the identified anti-patterns. To sum up, I make the following contributions:

- I conduct the first empirical study on characterizing and mitigating anti-patterns of alerts in an industrial cloud system.
- I identify six anti-patterns of alerts in a production cloud system. Specifically, the six anti-patterns can be divided into two categories, namely individual anti-patterns and collective anti-patterns. Individual anti-patterns result from the ineffective patterns in one single alert strategy, including *Unclear Name or Description*, *Misleading Severity*, *Improper and Outdated Alert Strategy*, and *Transient and Toggling Alerts*. Collective anti-patterns are ineffective patterns that a bunch of alerts collectively exhibit, including *repeating* and *cascading alerts*.
- I summarize the current industrial practices for mitigating the anti-patterns of alerts, including postmortem reactions to mitigate the effect of anti-patterns and the preventative guidelines to avoid the anti-patterns. The postmortem reactions include *rule-based alert blocking* and *alert aggregation*, *pattern-based alert correlation analysis*, and *emerging alert detection*. I also describe three aspects of designing preventative guidelines for alert strategies according to our experience in Huawei Cloud.
- Lastly, I share our thoughts on prospective directions to achieve automatic alert governance. I propose to bridge the gap between manual alert strategies and cloud service upgrades

by automatically evaluating the Quality of Alerts (QoA) in terms of *indicativeness*, *impact*, and *handleability*.

## 5.2 The Anti-patterns of Alerts

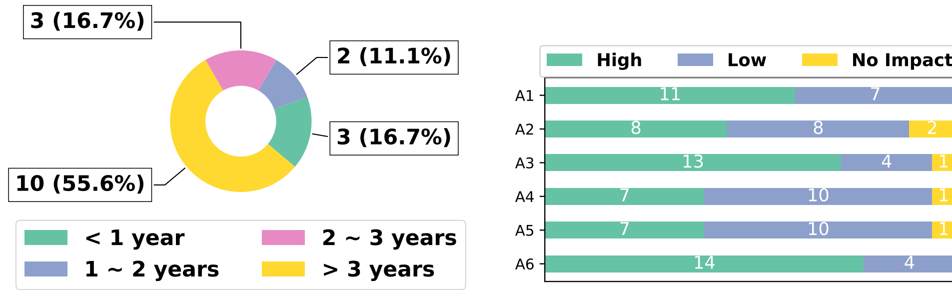
The research described in this chapter is motivated by the pain point of alert governance in a production cloud system. In this section, I present the first empirical study of characterizing the anti-patterns of alerts[2] and how I mitigate the anti-patterns in the production cloud system. Specifically, I study the following research questions (RQs).

- **RQ1**: What anti-patterns exist in alerts? How do these anti-patterns prevent OCEs from promptly and precisely diagnosing the alert?
- **RQ2**: What is the standard procedure to process alerts? Can the standard procedure handle the anti-patterns?
- **RQ3**: What are the current **reactions** to the anti-patterns of alerts? How about their performance?
- **RQ4**: What are the current measures to **avoid** the anti-patterns of alerts? How about their performance?

To answer these research questions, I quantitatively analyzed over 4 million alerts from the production system of Huawei Cloud which serves tens of millions of users and contains hundreds of services. The time range of the alerts spans over two years. I conducted a survey involving 18 experienced OCEs to find out the current practice of mitigating the anti-patterns of alerts. Among them, 10 (55.6%) OCEs have more than 3 years

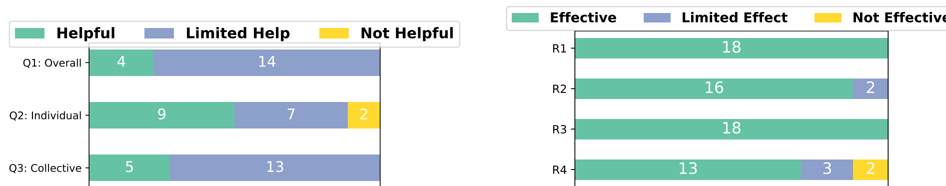
[2] An alert always corresponds to an alert strategy. Therefore, I do not discriminate "anti-pattern of alerts" and "anti-patterns of alert strategies".

of working experience. The number of OCEs with 2 to 3 years' working experience and 1 to 2 years' working experience are 3 (16.7%) and 2 (11.1%). Lastly, 3 (16.7%) OCEs' experience are less than 1 year.




(a) How long have you been working as an OCE?

(b) How about the impact of different anti-patterns to alert diagnosis?




(c) How helpful are the predefined SOPs?

(d) How about the effectiveness of current reactions to anti-patterns?

Figure 5.1: A survey about the current practice of mitigating the anti-patterns of alerts.

### 5.2.1 RQ1: Anti-patterns in Alerts

Anti-patterns of alerts are misconfigured and ineffective patterns in alerts that hinder alert processing in practice. Although alerts provide essential information to OCEs for diagnosing and mitigating failures, anti-patterns of alerts hinder this process. I divide the anti-patterns into two categories, i.e., *individual anti-patterns* and *collective anti-patterns*. *Individual anti-patterns* result from the ineffectiveness of one single alert.

In practice, OCEs usually have limited time to diagnose alerts. If one alert and its SOP are poorly designed, e.g., misleading steps to diagnose or non-informative description, the manual diagnosis will be difficult. *Collective anti-patterns* are ineffectiveness that alerts collectively exhibit. Sometimes, due to inappropriate configuration of alert strategy, complex dependency, and inter-influence effect in the cloud, numerous alerts may simultaneously occur. If alerts flood to OCEs or are collectively hard to handle, it will be too complicated for manual diagnosis, especially for inexperienced OCEs. Characterizing these anti-patterns is the leading step for alert governance.

For this research question, I analyzed more than 4 million alerts over two years to characterize the anti-patterns of alerts. The total number of alert strategies in this empirical study is 2010. To select the candidates of individual anti-patterns, I group the alerts according to the alert strategies, then calculate each strategies' average processing time. The alert strategies that take the top 30% longest time to process are selected as the candidates of individual anti-patterns. To find cases of collective anti-patterns, I first group all the alerts by the hour they occur and the region they belong to. Then I count the number of alerts per hour per region. If the number of alerts per hour per region exceeds 200[3], I select all the alerts in this group as the candidate of collective anti-patterns. I also went through the incident reports over the past two years to seek the ineffectiveness in alerts recorded by OCEs. I get five candidate cases of individual anti-patterns and two candidate cases of collective anti-patterns. After that, I ask two experienced OCEs to mark

[3] I set the threshold as 200 as the estimated maximum number of alerts an OCE team can deal with is 200. Experienced OCEs confirm the threshold.

whether they think the candidate ineffective pattern in alerts is an anti-pattern. If they both agree, I include it as an anti-pattern. If disagreements occur, another experienced OCE is invited to examine the pattern. As a result, I summarized four individual anti-patterns and two collective anti-patterns.
Our survey asked the OCEs to determine the impact of different anti-patterns on alert diagnosis. Figure 5.1(b) shows the answers' distributions. Each bar represents one anti-pattern, which is elaborated below.

#### Individual anti-patterns

Individual anti-patterns are the ineffectiveness of a single alert, including unclear name or description, misleading severity, and improper and outdated generation rule.
[**A1**] *Unclear Name or Description.* Unclear alert name or alert description obstructs the OCEs from gaining intuitive judgment at the first sight, which slows down the diagnosis and even hinders OCEs from knowing the logical connections from the alert to other alerts. Typical unclear alert names describe the system state in a very general way with vague words, e.g., "Elastic Computing Service is abnormal", "Instance $x$ is abnormal", "Component $y$ encounters exceptions", and "Computing cluster has risks". All OCEs agree with the impact of *unclear name or description*, and 61.1% of them think the impact is high.
[**A2**] *Misleading Severity.* Severity helps OCEs to prioritize which alert to diagnose first. Inappropriately high severity level takes up OCE's time for dealing with less essential alerts, while too low severity level may lead to missing important alerts. In our survey, 88.9% of OCEs agree with the impact of *misleading severity.* In practice, I find that the setting of severity heavily

depends on domain knowledge. With the update of the cloud system, especially the enhancement of fault tolerance mechanisms, the severity may also change.

[**A3**] *Improper and Outdated Generation Rule.* Typically, the cloud monitoring system will continuously monitor the performance indicators of both lower-level infrastructures (e.g., CPU usage, disk usage) and higher-level services (e.g., request per second, response latency). If any indicator increases over or drops below the predefined thresholds, an alert will be generated. Although the performance indicators of lower-level infrastructures can provide valuable information when the root cause of the alert is failures of lower-level infrastructures (e.g., high CPU usage), due to the fault-tolerance techniques applied in cloud services, the performance indicators of lower-level infrastructures do not have definite effect on the quality of cloud services from the perspective of customers. According to our survey, 72.2% of OCEs agree that the impact of *improper and outdated generation rule* is high.

[**A4**] *Transient and Toggling Alerts.* As mentioned in Section § 2.1.2, the cloud monitoring system can automatically clear some alerts. When the interval between the generation time and automatic clearance time of an alarm is less than a certain value (known as the intermittent interruption threshold), the alert is called a transient alert. Commonly speaking, a transient alert is an alert that lasts for a short time. When the same alert is generated and cleared multiple times (i.e., oscillation), and the number of oscillations is greater than a certain value (known as the oscillation threshold), it is called a toggling alert. Transient and toggling alerts are usually caused by alert strategies being too sensitive to the fluctuation of the metrics. Transient and tog-

gling alerts cause fatigue of OCEs and also distract the OCEs from being dealing with other important alerts. Although there are disagreements on the level of impact, most OCEs (94.4%) think the impact exists.

#### Collective anti-patterns

Collective anti-patterns result from the ineffective patterns of a bunch of alerts that occur in a short time scope. Zhao et al. [165] defined numerous alerts (e.g., hundreds of alerts) from different cloud services in a short time (e.g., one minute) as "alert storm", and conducted several case studies of alert storms. In alert storms, even if all the individual alerts are effective, the large number of alerts may still set obstacles for OCEs and greatly affect the system reliability in the following three ways. Firstly, during an alert storm, many alerts are generated. If OCEs check each alert manually, the troubleshooting will take unacceptably long time. Secondly, since alert storms occur frequently [165], the OCEs will continually receive alerts by email, SMS, or even phone call. According to our study, alert storms occur weekly or even daily, and 17 out of 18 interviewed OCEs say that the alert storms greatly fatigue them. Lastly, the overwhelming number of alerts adds pressure to the monitoring system, so the latency of generating new alerts may increase.
Inspired by [165], I summarize the following collective anti-patterns from confirmed cases of alert storms in Huawei Cloud. In this study, if the number of alerts from a region exceeds 100 in an hour, I count it as an alert storm. Consecutive hours of alert storm will be merged into one. Among the two collective anti-patterns, "cascading alerts" has already been observed by [165], but "repeating alerts" has not. In particular, I demon-

strate the collective anti-patterns of alerts with a representative alert storm that happened from 7:00 AM to 11:59 AM in Huawei Cloud. During the alert storm, totally 2751 alerts were generated, among which I observeed both collective anti-patterns as described below.

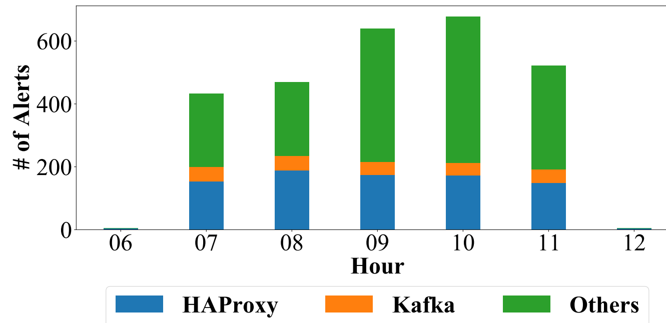



Figure 5.2: Repeating alerts in an alert storm.

[**A5**] *Repeating Alerts.* Repeating alerts means that alerts from the same alert strategy appear repeatedly. Sometimes the repeated alerts may last for several hours. This is usually due to the inappropriate frequency of alert generation. For example, in Figure 5.2, I count the number of alerts per strategy. The total number of alerts is 2751, and the number of effective alert strategies is 200. To make the figure clear, I only show the name of the top two alerts. All other alerts are classified as "Others" in the figure. The alert "haproxy process number warning", abbreviated as `HAProxy` in the figure, takes up around 30% of the total number of alerts in each hour. However, it is only a `WARNING` level alert, i.e., the lowest level. Even though an individual alert is straightforward to process, it is still time-consuming to deal with it when it occurs repeatedly. If one rule continually generates alerts, it will distract OCEs from dealing with the more essential alerts. Most OCEs (94.4%) agree with the impact of *repeating alerts*.

[**A6**] *Cascading Alerts.* Modern cloud systems are composed of many microservices that depend on each other [149]. When a service enters an anomalous state, other services that rely on it will probably suffer from anomalous states as well. Such anomalous states can propagate through the service-calling structure [38]. Despite various fault tolerance mechanisms being introduced, minor anomalies are still common to magnify their impact and eventually affect the entire system. Each of the affected services will generate many anomalous monitoring metrics, resulting in many alerts (e.g., thousands of alerts per hour). As a consequence, the alerts burst and flood to the OCEs. Although the alerts are different, they are implicitly related because they originate from the cascading effect of one single failure. Manually inspecting the alerts is hard without sufficient knowledge of the dependencies in the cloud system. All interviewed OCEs agree with the impact of *cascading alerts*. Table 2.3 shows a simplified sample of cascading alerts. By manually inspecting the alerts, experienced OCEs would infer that the alert 1 possibly cause alert 2 because 1) Alert 2&3 occurred right after alert 1 and 2) The relational database service relies on the block storage service as the backend. If the relational database service failed to commit changes, i.e., write data, one possible reason is that the storage service failed.

**Finding 1**: Individual anti-patterns and collective anti-patterns widely exist. They hinder alert diagnosis to different extent.

| SOP for alert `nginx_cpu_usage_over_80` | |
|---|---|
| **Description** | CPU usage of nginx instance is higher than 80% |
| **Generation Rule** | Continuously check the CPU usage of nginx instance, generate the alert when usage is higher than 80%. |
| **Potential Impact** | Affects the forwarding of all requests. |
| **Possible Causes** | a) The workload is too high. b) ...... |
| **Steps to Diagnose** | Step 1: execute command `top -bn1` in the instance.<br>Step 2: ...... |

Figure 5.3: An example Standard Operation Procedure.

### 5.2.2 RQ2: Standard Alert Processing Procedure

The Standard Operation Procedure (SOP) defines the procedure to process a single alert. For each alert, its SOP includes the alert name, the alert description, the generation rule of the alert (i.e., alert strategy), the potential impact on the cloud system, the possible causes, and the steps to process the alert. Figure 5.3 shows an example SOP of the alert `nginx_cpu_usage_over_80`. The OCEs can follow the SOP to process the alert upon receiving the alert.

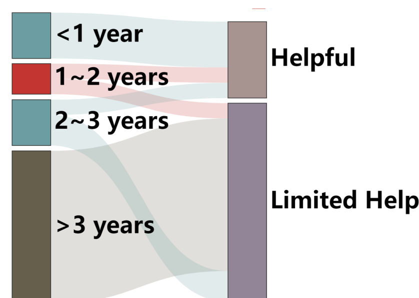



Figure 5.4: Answers to Q1 "Overall Helpfulness" regarding OCEs' working experience.

According to our survey, only 22.2% of OCEs think current SOPs are helpful (Q1, Figure 5.1(c)), and the other 77.8% of

OCEs say the help is limited. The SOPs are deemed to show limited help by all OCEs with over 3 years' experience, taking up 71.4% of all OCEs selected "Limited Help" for Q1 (Figure 5.4). Moreover, SOPs are considered much less helpful for diagnosing collective anti-patterns (Q3, Figure 5.1(c)) than individual anti-patterns (Q2, Figure 5.1(c)).

**Finding 2**: SOPs can help OCEs quickly process alerts, but the help is limited. SOPs are considered less helpful when dealing with collective anti-patterns.

### 5.2.3 RQ3: Reactions to Anti-patterns

Depending on the number of alerts, OCEs react differently. When the number of alerts is relatively small, OCEs will scan through all the reported alerts. Then they will manually rule out alerts that are not of great importance and deal with critical alerts that will affect the whole system.

OCEs react differently when the number of alerts becomes too large. According to our interview with senior OCEs in Huawei Cloud, they typically take four kinds of reactions, i.e., alert blocking, alert aggregation, alert correlation analysis, and emerging alert detection. In practice, I observe that although the reactions are considered effective, they need to be reconfigured after the update of cloud services or alert strategies.

[**R1**] *Alert Blocking.* When OCEs find that transient alerts, toggling alerts, and repeating alerts provide no information about service anomaly, they can treat these alerts as noise and block them with alert blocking rules. As a result, these non-informative alerts will not distract OCEs from quickly identifying the root

causes of service anomalies.

[**R2**] *Alert Aggregation.* When dealing with large amounts of alerts, there may be many duplicate alerts in a time period. For the non-informative alerts, OCEs will employ alert blocking introduced before to facilitate analysis. For the informative ones, they will adopt alert aggregation. To be more specific, OCEs will set rules to aggregate alerts in a period and use the number of alerts as another feature [32]. By doing so, OCEs can quickly identify critical alerts and focus more on the information provided by them.

[**R3**] *Alert Correlation Analysis.* Apart from the information provided by the alerts and their statistical characteristics, OCEs will also leverage other exogenous information to analyze the correlation of alerts. Two kinds of exogenous information are used to correlate alerts. The first is the dependencies of alert strategies, which indicate the spread of alerts in the cloud services [102]. For instance, if a source alert triggers another alert, OCEs will be more interested in the source alert, potentially the root cause of future service failures. They will associate all the derived alerts with their source alerts and diagnose the source alerts only. Another exogenous information is the topology of cloud services. Based on the topology of services, OCEs will set rules to correlate alerts based on the services that generated them. With this kind of correlation, OCEs can quickly pinpoint the root cause of a large number of alerts by following the topological correlation.

[**R4**] *Emerging Alert Detection.* Due to the large scale of cloud services, manually configured dependencies of alert strategies could not cover all the alert strategies. This may lead to the failure of alert correlation analysis. For example, a few alerts

corresponding to a root cause (i.e., emerging alerts) appear first. If they are not dealt with seriously, when the root cause escalates its influence, numerous cascading alerts will be generated. The lack of critical association rules will prevent the OCEs from discovering the correlation and quickly alert diagnosis. This usually happens on gray failures like memory leak and CPU overloading. Hence, it would be helpful to capture the implicit dependencies. I employ the adaptive online Latent Dirichlet Allocation [52, 148] to capture the implicit dependencies. OCEs could detect these emerging alerts as early as possible for faster alert diagnosis with the implicit dependencies.
Figure 5.1(d) shows OCEs' opinions about the effectiveness of the four reactions. In general, the effectiveness of all four reactions is relatively high.

**Finding 3**: Current reactions are considered effective, but the configurations of such reactions still require domain knowledge.

### 5.2.4 RQ4: Avoidance of Anti-patterns

To avoid the alert anti-patterns from occurring, Huawei Cloud also adopts preventative guidelines and conducts periodical reviews on alert strategies. I summarize the generic aspects to consider when designing the guidelines. The guidelines are designed by experienced OCEs and guide from three aspects of alerts.

- *Target* means what to monitor. The performance metrics highly related to the service quality should be monitored.

- *Timing* means when to generate an alert upon the manifes-

tation of anomalies. Sometimes an anomaly does not necessarily mean the service quality will be affected.

- *Presentation* means whether the alerts' attributes are helpful for alert diagnosis.

However, our interview with OCEs shows that the preventative guidelines are not strictly obeyed in practice. Most (88.9%) OCEs agree that strictly following the guidelines will make alert diagnosis easier.

> **Finding 4**: The preventative guidelines could reduce the anti-patterns and assist in alert diagnosis if they are carefully designed and strictly obeyed.

## 5.3 The Practice of Logging

Logging is the task of constructing logging statements with proper description and necessary program variables, and inserting the logging statements to the right positions in the source code. Logging has attracted attention from both academia [107, 123, 128, 146] and industry [14, 26, 47, 123, 154, 158] across a variety of application domains because logging is a fundamental step for all the subsequent log mining tasks.

However, the practice of logging is scarcely documented or regulated by strict standard, such as the logging mechanism and APIs [14]. Essentially, logging is a subjective task that relies heavily on human expertise [47, 62, 118, 123, 132].

In this section, I first define the basic mechanism of logging, then review the challenges and existing solutions. Specifically, I study the following research questions (RQs).

- **RQ5**: What are the general mechanism to generate logs?
- **RQ6**: What are the challenges for logging?
- **RQ7**: What are the current approaches to solve the challenges for logging?

To answer these research questions, I searched several popular online digital libraries (*e.g.*, IEEE Xplore, ACM Digital Library, Springer Online, Elsevier Online, Wiley Online, and ScienceDirect) with the following keywords: "log", "logging". The "log" term is broadly quoted in various domains, such as the query log in database, web search log in recommendation, and the logarithm function in mathematics. Hence, to precisely collect the set of papers of our interest, I mainly focused on regular papers published in top venues (*i.e.*, conferences and journals) of relevant domains, including ICSE, FSE, ASE, TSE, TOSEM, EMSE, SOSP, OSDI, ATC, NSDI, TDSC, DSN, ASPLOS, and TPDS. Then, I manually inspected each reference of these papers to collect additional publications that are related to the survey topics. In this section, I focus on publications studying the reliability issues with logging in software systems. In the end, I get 33 papers in the last 23 years across a variety of topics in logging practice, including *where-to-log*, *what-to-log*, and *how-to-log*. The papers under exploration are mainly from top venues in three related fields: software engineering (*e.g.*, ICSE), system (*e.g.*, SOSP), and networking (*e.g.*, NSDI).

### 5.3.1 RQ5: Logging Mechanism and Libraries

```
1 public void setTemperature(Integer temperature) {
2     // ...
3     logger.debug("Temperature set to {}. Old temperature was
      ↪ {}.", t, oldT);
4     if (temperature.intValue() > 50) {
5     logger.info("Temperature has risen above 50 degrees.");
6     }
7 }
8
```

⇓

```
1 0 [setTemperature] DEBUG Wombat - Temperature set to 61. Old
    ↪ temperature was 42.
2 0 [setTemperature] INFO Wombat - Temperature has risen above 50
    ↪  degrees.
3
```

Figure 5.5: An example of logging statements by SLF4J and the generated logs.

**Logging Mechanism**

Logging mechanism is the set of logging statements and their activation code implemented by developers or a given software platform [123]. Figure 5.5 shows two example logging statements and the collected logs during the execution of the program. The logging statement at line 3 is executed every time the `setTemperature` method is called, with no specific activation code. The logging statement at line 5 is controlled by the activation code `if (temperature.intValue() > 50)` at line 4. According to the data collected by [123], the most widely-adopted coding pattern that is used to activate the logging statements is `if (condition) then log error()`.

**Logging Libraries**

To improve flexibility, industrial developers often utilizes logging libraries [28, 123], which are software components that facilitate

logging and provide advanced features (*e.g.*, thread-safety, log archive configuration, and API separation). Toward this end, a lot of open-source logging libraries have been developed (*e.g.*, Log4j [95], SLF4J [137], AspectJ [10], spdlog [138]). Log4j and SLF4J are popular logging libraries for Java. SLF4J serves as a simple and flexible abstraction for various logging frameworks (*e.g.* Log4j, a fully-fledged logging library from Apache Software Foundation). AspectJ is an aspect-oriented extension of Java. spdlog is a very fast cross-platform logging library written in C++11. It has an asynchronous mode, multi-threaded logging, rich formatting, and a lot more features.

### 5.3.2 RQ6: Challenges for Logging

Logging in a software system is usually decided by an empirical process in the development phase [123]. In general, logging practice, *i.e.*, how developers conduct the task of logging, is scarcely documented or regulated by strict standard, such as the logging mechanism and APIs [14]. Thus, logging relies heavily on human expertise [47,62,118,123,132]. In the following, I summarize three main challenges for automated logging, which also align with the taxonomy mentioned by Chen and Jiang [25]: *where-to-log*, *what-to-log*, and *how-to-log*. Under this categorization, every problem exhibits aspects that represent the primary concerns of the actual practice of logging. Accordingly, I summarize three major aspects, *i.e.*, *diagnosability*, *maintenance*, and *performance*.

**where-to-log**

Where-to-log is about determining the appropriate location of logging statements. Although logging statements provide rich information and can be inserted almost everywhere, excessive logging results in performance degradation [25] and incurs additional maintenance overhead. In addition, it is challenging to diagnose problems by analyzing a large volume of logs as most logs are unrelated to the problematic scenarios [71]. On the other hand, insufficient logging will also impede the logs' diagnosability. For example, an incomplete sequence of logs may hinder the reproduction of precise execution paths [169]. Therefore, developers need to be circumspect in their choices of where-to-log.

**what-to-log**

What-to-log is about providing sufficient and concise information within the three major components of a logging statement, *i.e.*, verbosity level, static text, and dynamic content. Misconfigured verbosity level has similar consequences with inappropriate logging points. As developers typically filter logs according to the verbosity levels, under-valued verbosity levels may result in missing or ignored log messages while over-valued verbosity levels lead to overwhelming log messages [71]. When composing a snippet of logging code, the static text should be concise and the dynamic content should be coherent and up-to-date. Poorly written static text and inconsistent dynamic content could affect the subsequent diagnosis and maintenance activities [73, 93, 104, 161].

#### how-to-log

How-to-log is the ”design pattern” and maintenance of logging statements systematically. Most software testing techniques focus on verifying the quality of feature code, but a few papers [25, 26, 61, 133, 154] pay attention to the quality and anti-patterns in the logging code. Although the Aspect-Oriented Programming (AOP) paradigm [76] provides a systematic approach to modularize the logging statements, most industrial and open source systems choose to scatter logging statements across the entire code base, intermixing with feature code [25], which also hardens the maintenance of logging code.

### 5.3.3 RQ7: Logging Approaches

Numerous solutions have been proposed to address the challenges mentioned in Section § 5.3.2. Table 5.2 summarizes existing studies along with corresponding problems and aspects. Each row represents a challenge. The approaches fall into three categories: *static code analysis*, *machine learning*, and *empirical study*. A bunch of early work utilized static code analysis to analyze logging in a program without executing the source code. Machine learning-based approaches focus on learning from data. By concentrating on the inherent statistical properties of existing logging statements, learning-based approaches automatically give suggestions on improving the logging statements. In the remainder of this section, I discuss solutions by their aspects.

#### Diagnosability

Logs are valuable for investigating and diagnosing failures. However, a logging statement is only as helpful as the information it

Table 5.2: Summary of logging approaches.

| Problems | Aspects | Objectives |
|---|---|---|
| **where-to-log** | Diagnosability | Suggest appropriate placement of logging statements into source code [35, 36, 88, 150, 153, 168, 169, 175] ; Study logging practices in industry [47, 84]. |
| | Performance | Minimize or reduce performance overhead [41, 153]. |
| **what-to-log** | Diagnosability | Enhance existing logging code to aid debugging [155]; Suggest proper variables and text description in log [62, 85, 93]. |
| | Maintenance | Determine whether a logging statement is likely to change in the future [74]; Characterize and detect duplicate logging code [89]. |
| | Performance | Study the performance overhead and energy impact of logging in mobile app [34, 158]; Automatically change the log level of a system in case of anomaly [109]. |
| **how-to-log** | Diagnosability | Characterize the anti-patterns in the logging code [134]; Optimize the implementation of logging mechanism to facilitate failure diagnosis [98]. |
| | Maintenance | Characterize and detect the anti-patterns in the logging code [25, 26, 61, 86, 87, 154]; Characterize and prioritize the maintenance of logging statements [73]; Study the relationship between logging characteristics and the code quality [27, 133]; Propose new abstraction or programming paradigm of logging [76, 94]. |
| | Performance | Optimize the compilation and execution of logging code [147]. |
| | Security | Mitigate the flood attack of access logs [80]; Detect and prevent tampering in system logs for system auditing [122]. |

provides. The research on this aspect aims at (1) understanding the helpfulness of logs for failure diagnosis and (2) making logs informative for diagnosis.

*(1) Understanding the helpfulness of logs for failure diagnosis.* This is of great importance because logs are widely adopted for failure diagnosis. According to a survey [47] involving 54 experienced developers in Microsoft, almost all the participants agreed that "logging statements are important in system development and maintenance" and "logs are a primary source for problem diagnosis". Fu *et al.* [47] also studied the types of logging statements in industrial software systems by source code analysis, and summarized five types of logging snippets, *i.e.*, assertion-check logging, return-value-check logging, exception logging, logic-branch logging, and observing-point logging. Besides, Fu *et al.* [47] further demonstrated the potential feasibility of predicting where to log. Shang *et al.* [134] conducted the first empirical study to provide a taxonomy for user inquiries of logs. By manually going through emails in the mailing list and StackOverflow questions for Hadoop, Cassandra and Zookeeper, they identified five types of information that were often sought from log lines by practitioners, *i.e.*, meaning, cause, context, impact, and solution. Shang *et al.* [134] were also the first to associate the development knowledge at present in various development repositories (*e.g.*, code commits and issues reports) with the log lines and to assist practitioners in resolving real-life log inquiries. In addition, Li *et al.* [84] highlighted the feasibility of guiding developers' logging practice with topic models by investigating six open source systems.

*(2) Making logs informative for failure diagnosis.* As printed logs are often the only run-time information source for debug-

ging and analysis, the quality of log data is critically important. *LogEnhancer* [155] made the first attempt to systematically and automatically augment existing logging statements in order to reduce the number of possible code paths and execution states for developers to pinpoint the root cause of a failure. Zhao *et al.* [168, 169] followed the idea of *LogEnhancer* and proposed an algorithm capable of completely disambiguating the call path of HDFS requests. Yuan *et al.* [153] found that the majority of unreported failures were manifested via a generic set of error patterns (*e.g.*, system call return errors) and proposed the tool *Errlog* to proactively add pattern-specific logging statements by static code analysis.

As modern software becomes more complex, where-to-log has become an important but difficult decision, largely limited to the developer's domain knowledge. Around 60% of failures due to software faults do not leave any trace in logs, and 70% of the logging pattern aims to detect errors via a checking code placed at the end of a block of instructions [35, 36]. Cinque *et al.* [36] concluded that the traditional logging mechanism has limited capacity due to the lack of a systematic error model. They further formalized the placement of the logging instruction and proposed to use system design artifacts to manually define *rule-based logging* which utilizes error models about what cause errors to fail. Zhu *et al.* [175] made an important first step towards the goal of "learning to log". They proposed a logging recommendation tool, *LogAdvisor*, that learns the common logging rules on where-to-log from existing code via training a classifier and further leverages it for informative and viable recommendations to developers. Yao *et al.* [150] leveraged a statistical performance model to suggest the need for updating

logging locations for performance monitoring. Li *et al.* [88] proposed a deep learning framework to suggest where-to-log at the block level. Li *et al.* also concluded that there might be similar rules regarding the implementation of logging mechanism across different systems and development teams, which agreed with the industrial survey by Pecchia *et al.* [123].
The lack of strict logging guidance and domain-specific knowledge makes it difficult for developers to decide what-to-log. To address this need, Li *et al.* [85] employed ordinal regression model to suggest proper verbosity level based in software metrics. He *et al.* [62] conducted the first empirical study on the usage of natural language in logging statements. They showed the global (*i.e.*, in a project) and local (*i.e.*, in a file) repeatability of text descriptions. Furthermore, they demonstrated the potential of automated description text generation for logging statements. Liu *et al.* [93] proposed a deep learning-based approach to recommend variables in logging statements by learning embeddings of program tokens. In order to troubleshoot transiently-recurring problems in cloud-based production systems, Luo *et al.* [98] put forward a new logging mechanism that assigns a blame rank to methods based on their likelihood of being relevant to the root cause of the problem. With the blame rank, logs generated by a method over a period of time are proportional to how often it is blamed for various misbehavior, thus facilitating diagnosis.

**Maintenance**

The maintenance of logging code has also attracted researchers' interest. The research on this aspect aims at (1) characterizing the maintenance and detecting anti-patterns of logging state-

ments and (2) proposing new abstractions of logging.

*(1) Characterizing the maintenance and detecting anti-patterns of logging statements.* Anti-patterns in logging statements are bad coding patterns that undermine the quality and effectiveness of logging statements and increases the maintenance effort of projects. Many papers [26, 27, 61, 133] performed empirical studies to reveal the link between logs and defects. These papers observed a positive correlation between logging characteristics and post-release defects. Therefore, practitioners should allocate more effort to source code files with more logging statements.

Yuan *et al.* [154] made the first attempt to conduct a quantitative characteristic study of how developers log within four pieces of large open-source software. They described common anti-patterns and provided insights into where developers spend most of their efforts in modifying the log messages and how to improve logging practice. They further implemented a prototype checker to verify the feasibility of detecting unknown problematic statements using historical commit data. Chen and Jiang [25] and Hassani *et al.* [61] both studied the problem of how-to-log by characterizing and detecting the anti-patterns in the logging code. The analysis [25] of well-maintained open-source systems revealed six anti-patterns that are endorsed by developers. Chen and Jiang [25] then encoded these anti-patterns into a static code analysis tool to automatically detect anti-patterns in the source code. Li *et al.* [89] developed an automated static analysis tool, *DLFinder*, to detect duplicate logging statements that have the same static text description.

Just like feature code, logging code updates with time [74]. Moreover, logging statements are often changed without con-

sideration for other stakeholders, resulting in sudden failures of log analysis tools and increased maintenance costs for such tools. Pecchia *et al.* [123] reviewed the industrial practice in the reengineering of logging code. Kabinna *et al.* [73] empirically studied the migration of logging libraries and the main reasons for the migration. Li *et al.* [86] derived and used a set of measures to predict whether a code commit requires log changes. Kabinna *et al.* [74] later examined the important metrics for determining the stability of logging statements and further leveraged learning-based models (random forest classifier and Cox proportional hazards) to determine whether a logging statement is likely to remain unchanged in the future. Their findings were helpful to build robust log analysis tools by ensuring that these tools relied on logs generated by more stable logging statements. Li *et al.* [87] designed a tool to learn log revision rules from logging context and modifications and recommend candidate log revisions.

*(2) Proposing new abstractions of logging.* Maintaining logging code along with feature code has proven to be error-prone [26, 61, 133]. Hence, additional logging approaches [76, 94] have been proposed to resolve this issue. Kiczales *et al.* [76] proposed a new programming paradigm that improves the modularity of the logging code. To tackle the ordering problem of logs in distributed systems, Lockerman *et al.* [94] introduced *FuzzyLog* that featured strong consistency, durability, and failure atomicity.

#### Performance

The intermixing nature of the logging code and feature code usually incurs performance overhead, storage cost, and development and maintenance efforts [34, 41, 109, 155, 158]. Tools like *LogEn-*

*hancer* [155], *Errlog* [153], *Log2* [41], and *INFO-logging* [168] all took performance into consideration while dealing with diagnosability and maintenance issues. Mizouchi *et al.* [109] proposed a dynamical adjusting verbosity level to record irregular events while reducing performance overhead. Yang *et al.* [147] proposed *NanoLog*, a nanosecond scale logging system that achieved relatively low latency and high throughput by moving the workload of logging from the runtime hot path to the post-compilation and execution phases of the application. Chowdhury *et al.* [34] were the first to explore the energy impact of logging in mobile apps. Zeng *et al.* [158] conducted a case study that characterized the performance impact of logging on Android apps.

**Security**

Another common and significant research topic about logs is security. This topic focuses on making logging statements invulnerable and mitigating attacks by the logging mechanism. For instance, Lam *et al.* [80] proposed *Carousel* to mitigate the flood attack of hundreds of millions of duplicate log records. Recently, Paccagnella *et al.* [122] introduced a practical framework, *CUSTOS*, for the detection of tampering in system logs. System security is a central concern, but it is beyond the scope of this thesis, so I will not go into this topic too much.

## 5.4 Discussion

### 5.4.1 Detecting Anti-patterns of Alerts

Although several postmortem reactions and preventative guidelines are adopted (Section § 5.2), according to our study, the

problem of alert anti-patterns is still prevailing in industrial cloud monitoring systems because most current measures still require manual configuration. As for the alert blocking, OCEs need to inspect each alert and set rules manually. How to define the blocking rules and when to invalidate these rules become a crucial problem. A similar problem also exists in alert correlation. As for alert correlation analysis, OCEs also need to inspect alert generating rules and service topology documents apart from reading alerts, which incurs a considerable burden to OCEs. Moreover, the effectiveness of the reactions also lacks clear criteria to evaluate. OCEs can only estimate the effectiveness of the reactive measures by their feeling. Therefore, outdated reactive measures is hard to detect. As a result, the whole process of alert governance becomes time-consuming and laborious.

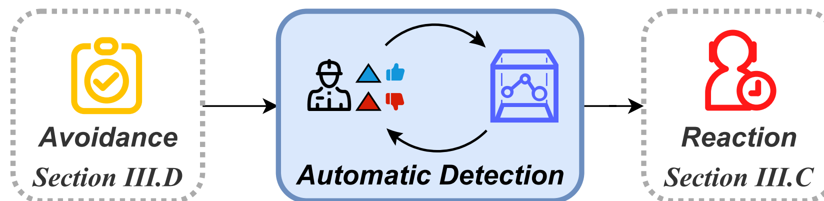



Figure 5.6: Incorporating human knowledge and machine learning to detect anti-patterns of alerts.

In Figure 5.6, I formulate the three stages of the mitigation of alert anti-patterns. I already shared our experience of avoiding and reacting to alert anti-patterns in Section § 5.2. To close the gap between manual alert strategies and cloud system upgrades, I propose to explore the automatic detection of alert anti-patterns. Automatic evaluation of the Quality of Alerts (QoA) will be a promising approach to the automatic detection of alert anti-patterns.

Based on our empirical study, I propose three criteria to measure the quality of alerts (QoA), including *indicativeness*, *precision*, and *handleability*.

- *Indicativeness* measures whether the alert can indicate the failures that will affect the end users' experience.
- *Precision* measures whether the alert can correctly reflect the severity of the anomaly.
- *Handleability* measures whether the alert can be quickly handled. The handleability depends on the target and the presentation of the alert. Improper target or unclear presentation decreases the handleability.

In the future, incorporating human knowledge and machine learning to evaluate the three aspects of alerts deserves more exploration. In particular, OCEs provide their domain knowledge by creating labels like "high/low precision/handleability/indicativeness" for each alerts during alert processing. With the labels, a machine learning model could be trained and continuously updated so that it can automatically absorb the human knowledge for future QoA evaluation.

### 5.4.2 Best Practices for Logging

I discuss the current industrial best practices based on our experience and surveyed papers and articles.

**Practice 1: Always follow the logging standards** It is crucial to follow the standards of logging during development, otherwise the produced logs would be hard to maintain, search and analyze. For example, the following logging standards are shared

by various systems: (1) Timestamp: timestamp helps developers understand the sequential relationship among log events. Using correct timestamps (UTC/timezone adjusted) is necessary for debugging and analytic purposes. (2) Verbosity levels: proper verbosity levels ease log parsing and searching. In addition, aggregating logs by verbosity level is beneficial. (3) Format: log format is highly correlated with the parse and search procedures while most people might ignore. It is recommended to structure logs following an agreed standard (e.g., in JSON format) within the same project group. (4) Log message: meaningful log messages facilitates the identification of the correct root cause for a failure. To construct meaning log messages, it is suggested to avoid duplicate logging descriptions (e.g., by assignment a unique ID to each logging statement).

**Practice 2: Keep proper quantity of log messages** Controlling the number of logging statements in the source code is very tricky. If logging too little, engineers may not have adequate information for problem diagnosis. On the contrary, if logging too much, engineers can easily get overwhelmed by the huge volume of logs and problem diagnosis is like looking for needles in haystacks. Moreover, too many logging statements could lead to unnecessary performance overhead [41,153]. Hence, it is crucial to keep proper quantity of log messages.

## 5.5 Related Work

Many approaches are proposed to manage the alerts and incidents of cloud services. In this section, I introduce related work on improving service reliability using alerts and incidents on

cloud platforms.

#### Alert Management

As for incident management, many works focus on incident management. Incident triage is to assign incidents to the teams that are responsible for repairing the services that cause this incident. For incident assignment, existing approaches mine historical outage triage records with Named-Entity Recognition (NER) [135] and service correlation [145]. Shetty et al. [135] employ Named-Entity Recognition (NER) to extract knowledge from incident tickets and conduct incident triage with the extracted knowledge. Wang et al. [145] focuses on outages and propose to assign outages to teams that are responsible for the failed services using service correlation mined from historical outage triage records. Chen et al. [29] empirically study the possibility of applying multiple bug triage methods to deal with incidents. To understand the challenges of employing AIOps to automatically conduct IT Operations on the cloud, Chen et al. [31] conducted a thorough empirical study over two years of incident management practices at Microsoft. They find that the incomplete service/resource dependencies and imprecise resource health assessment are two critical challenges of AIOps. Zeng et al. [172] resolves incidents by the text information in the incident tickets. Specifically, they adopt machine learning to provide a matching score for each ticket and historical resolution pair, which can be viewed as recommendation of resolution. Finally, Zhao et al. [166] apply multi-instance learning to predict potential future incidents. Machine-learning interpretation methods are adopted to provide an interpretable report to facilitate incident triaging.

#### Cloud Monitoring

Mormul et al. [111] proposes DEAR to automatically distribute alerting rules to the monitored resources to reduce the network traffic while preserving alert precision. Li et al. [90] proposes Gandalf, an end-to-end analytics service for safe deployment in a large-scale system infrastructure. It uses a spatial and temporal correlation algorithm to correlate fault signals with ongoing rollouts to find rollouts that may cause widespread outages. Levy et al. [82] find that reactively dealing with failures in the cloud is inadequate for providing reliable services. They propose Narya, a preventive and adaptive failure mitigation service which predicts imminent host failures and decide proper actions. Gu et al. [56] proposes MID to detect incidents from large-amount, multi-dimensional issue reports.

---

□ **End of chapter.**

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

Recent years have witnessed the wide adoption of microservice architecture in online services. The reliability of online services is crucial to both customers and service providers. However, the increasing complexity and scale in microservice systems make the system reliability harder to ensure, which poses great challenges for microservice reliability engineering. In this thesis, I study the intelligent operations based on the traces, metrics, logs, and alerts in microservice systems, which is crucial to improve the reliability of cloud microservices. Specifically, I explore three types of measures for reliability engineering, i.e., proactive measures, reactive measures, and retrospective measures. I propose a metric-based self-adaptive resilience testing framework as the proactive measure and a trace-based approach for predicting the aggregated intensity of microservice dependency. Furthermore, I study the anti-patterns and best practices for generating logs and alerts for assisting the reliability engineers. The contributions are summarized as follows:

In Chapter 3, I first conduct a comprehensive empirical study on

the maintenance of AWS and Huawei Cloud. I identify the inefficiency in failure diagnosis and recovery with the binary-valued dependencies and define the intensity of dependency for the first time. To facilitate cloud maintenance, I propose AID, the first approach to predict the intensity of dependencies between cloud microservices. AID first generates a set of candidate dependency pairs from the spans. AID then represents the status of each cloud service with a multivariate time series aggregated from the spans and calculates the similarities between the statuses of the caller and callee of each candidate pair. Finally, AID aggregate the similarities to produce a unified value as the intensity of the dependency. For the evaluation, I collect and manually label a new dataset from an open-source microservice benchmark and evaluate AID on it. Furthermore, I evaluate AID using the data of Huawei Cloud and showcase the practical usage of AID. Both the evaluation results and case studies show the efficiency and effectiveness of AID. In the future, I plan to incorporate more information from the traces and other service logs for more accurate predictions.

In Chapter 4, based on our empirical study on the failures' manifestations in resilient and unresilient microservice systems, I propose AVERT for the self-adaptive resilience evaluation of microservice systems to mitigate the scalability and adaptivity issue of current human-dependent resilience testing procedure. I achieve this by measuring the propagation of degradation from system performance metrics to business metrics. The higher the degradation propagation, the lower the resilience of the microservice system. The experimental results on two open-source benchmark microservice systems show that AVERT can efficiently and accurately test the resilience of microservice sys-

tems. AVERT achieves the best performance of 0.1159 in terms of cross entropy, outperforming all baseline methods.

In Chapter 5, I conduct an empirical study on alerting and logging for improving the reliability of cloud systems. For alerting, I study the alert strategies and the alert processing procedure at Huawei Cloud, a leading cloud provider. I conduct the first empirical study to characterize the anti-patterns in cloud alerts. I also summarize the industrial practices of mitigating the anti-patterns by postmortem reactions and preventative guidelines. For logging, I mainly cover four major aspects, i.e., diagnosability, maintenance, performance, security. Additionally, I introduce the available open-source toolkits. Based on the investigation of recent advances, I propose new insights and discuss several future directions. Our study aims at inspiring further research on automatic QoA evaluation and analysis-oriented logging, and benefit the reliability of the cloud services in the long run.

## 6.2 Future Directions

In this section, I would like to discuss promising future directions. Although a number of novel techniques for microservice reliability have been proposed, in the near future, I expect more interesting and inspiring research studies in the microservice reliability field. Aside from the microservice architecture, I also share our thoughts on serverless computing, which is the next-generation of cloud computing [72]. In the following, I introduce the following future directions:

### 6.2.1 Trace Compression based on Service Topology

Traces record the status of each microservice invocation, including the return value, the duration of execution, etc. Owing to the complex service invocations, the number of traces grows exponentially with the number of microservices. Since large-scale microservice systems run on a $24 \times 7$ basis, the size of the generated traces is huge. For example, the open-source microservice benchmark, Social-Network, generates 40 gigabytes of traces per hour when I continuously simulate users' browsing behavior. Long term archiving of such a huge amount of data will bring unimaginable heavy burden to storage space, power and transmission network bandwidth. Although a series of studies have focused on log compression, the compression of traces problem have not attracted much attention yet. As traces are closely related with the topology of services, it is promising to explore trace compression based on service topology, which aims to eliminate the redundancy in traces and reduce their storage consumption.

### 6.2.2 Analysis-Oriented Logging

As the foundation of automated log analysis, current logging practices mostly focus on characterization and recommendation of logging statements (*i.e.*, where, what, and how to log). Meanwhile, a recent industrial study [14] pointed out that the inconsistent presentation format with-in and among teams poses a significant challenge in developing automated log analysis tools. In addition, even an effective log mining algorithm could generate meaningless results if the software runtime information is not well-documented in the collected logs. Thus, a crucial direction

is designing advanced logging mechanisms that can coordinate with the subsequent log analysis steps during the development.

### 6.2.3 Automated Generation of Logging Statements

The automated generation of logging statements can alleviate developers' burden. Recent research at the intersection of machine learning, programming languages, and software engineering has taken an important step toward proposing learnable code-generating models that utilize rich patterns in source code [5]. Automatically generating logging statements is approachable by incorporating code-generating models with existing approaches that suggest where, what, and how to log.

---

□ **End of chapter.**

# Chapter 7

# Publications during Ph.D. Study

(i) **Tianyi Yang**, Jiacheng Shen, Yuxin Su, Xiaoxue Ren, Xiao Ling, Yongqiang Yang, and Michael R. Lyu. *Characterizing and Mitigating Anti-patterns of Alerts in Industrial Cloud Systems.* In Proceedings of the 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'22) June 27-30, 2022, Baltimore, Maryland, USA. IEEE, 2022, pp. 393-401.

(ii) **Tianyi Yang**, Baitong Li, Jiacheng Shen, Yuxin Su, Yongqiang Yang, and Michael R. Lyu. *Managing Service Dependency for Cloud Reliability: The Industrial Practice.* In Proceedings of the 33rd IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'22) October 31 - November 3, 2022, Charlotte, North Carolina, USA. IEEE, 2022. pp. 1-2.

(iii) **Tianyi Yang**, Jiacheng Shen, Yuxin Su, Xiao Ling, Yongqiang Yang, and Michael R. Lyu. *AID: Efficient Prediction of Aggregated Intensity of Dependency in Large-scale Cloud Systems.* In Proceedings of the 36th IEEE/ACM International

Conference on Automated Software Engineering (ASE'21) November 15-19, 2021, Australia. IEEE/ACM, 2021, pp. 653-665.

(iv) **Tianyi Yang**, Cuiyun Gao, Jingya Zang, David Lo, and Michael R. Lyu. *TOUR: Dynamic Topic and Sentiment Analysis of User Reviews for Assisting App Release.* In Companion Proceedings of the Web Conference 2021 (WWW'21), April 19–23, 2021, Ljubljana, Slovenia. ACM, 2021, pp. 708–712.

(v) Jiacheng Shen, **Tianyi Yang**, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. *Defuse: A Dependency-Guided Function Scheduler to Mitigate Cold Starts on FaaS Platforms.* In Proceedings of the 41st IEEE International Conference on Distributed Computing Systems (ICDCS'21) July 7-10, 2021, Washington DC, USA. IEEE, 2021, pp. 194-204.

(vi) Shilin He, Pinjia He, Zhuangbin Chen, **Tianyi Yang**, Yuxin Su, and Michael R. Lyu. *A Survey on Automated Log Analysis for Reliability Engineering.* ACM Computing Survey (CSUR), April, 2021. ACM, New York, NY, USA. ACM, 2021, pp. 1-37.

(vii) (*In submission*) **Tianyi Yang**, Baitong Li, Jiacheng Shen, Yuxin Su, Yongqiang Yang, Michael R. Lyu. *AVERT: A Self-adaptive Resilience Testing Framework for Microservice Systems.*

(viii) (*In submission*) Baitong Li, **Tianyi Yang**, Zhuangbin Chen, Yuxin Su, Shijian Chen, Michael R. Lyu. *Eadro: Integrating Anomaly Detection and Root Cause Localization on Multi-source Monitoring Data for Microservices.*

(ix) (*In submission*) Baitong Li, **Tianyi Yang**, Zhuangbin Chen, Yuxin Su, Yongqiang Yang, Michael R. Lyu. *HADES: Heterogeneous Anomaly Detection for Software Systems via Attentive Multi-modal Learning.*

(x) (*In submission*) Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, **Tianyi Yang**, Yuxin Su, Yangfan Zhou, Michael R. Lyu. *ScaleStore: Scalable and Fault-Tolerant Key-Value Store on Disaggregated Memory.*

---

□ **End of chapter.**

# Bibliography


[1] G. Aceto, A. Botta, W. De Donato, and A. Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.

[2] A. A. Ahmad and P. Andras. Scalability resilience framework using application-level fault injection for cloud-based software services. *J. Cloud Comput.*, 11:1, 2022.

[3] A. B. M. B. Alam, A. Haque, and M. Zulkernine. CREM: A cloud reliability evaluation model. In *IEEE Global Communications Conference, GLOBECOM 2018, Abu Dhabi, United Arab Emirates, December 9-13, 2018*, pages 1–6. IEEE, 2018.

[4] Alibaba. Chaosblade: An easy to use and powerful chaos engineering experiment toolkit, 2022.

[5] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. A. Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Survey*, pages 81:1–81:37, 2018.

[6] P. Alvaro, J. Rosen, and J. M. Hellerstein. Lineage-driven fault injection. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data,*

*Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 331–346. ACM, 2015.

[7] Amazon. What are microservices?, 2022.

[8] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[10] AspectJ. Eclipse aspectj, 2020.

[11] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Softw.*, 33(3):42–52, 2016.

[12] A. E. Bargagliotti and R. N. Greenwell. Combinatorics and statistical issues related to the kruskal-wallis statistic. *Commun. Stat. Simul. Comput.*, 44(2):533–550, 2015.

[13] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*. USENIX Association, May 2003.

[14] T. Barik, R. DeLine, S. Drucker, and D. Fisher. The bones of the system: A case study of logging and telemetry at microsoft. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 92–101. IEEE, 2016.

[15] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[16] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Softw.*, 33(3):35–41, 2016.

[17] G. E. A. P. A. Batista, E. J. Keogh, O. M. Tataw, and V. M. A. de Souza. CID: an efficient complexity-invariant distance for time series. *Data Min. Knowl. Discov.*, 28(3):634–669, 2014.

[18] G. E. A. P. A. Batista, X. Wang, and E. J. Keogh. A complexity-invariant distance measure for time series. In *Proceedings of the Eleventh SIAM International Conference on Data Mining, SDM 2011, April 28-30, 2011, Mesa, Arizona, USA*, pages 699–710. SIAM / Omnipress, 2011.

[19] P. Bermejo, L. de la Ossa, J. A. Gámez, and J. M. Puerta. Fast wrapper feature subset selection in high-dimensional datasets by means of filter re-ranking. *Knowl. Based Syst.*, 25(1):35–44, 2012.

[20] D. J. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. In *Knowledge Discovery in Databases: Papers from the 1994 AAAI Workshop, Seattle, Washington, USA, July 1994. Technical Report WS-94-03*, pages 359–370. AAAI Press, 1994.

[21] D. Blackmore, C. Tornbohm, D. Ackerman, C. Graham, S. Matson, T. Lo, T. Singh, A. Roy, C. Tenneson,

M. Sawai, E. Kim, E. Anderson, S. Nag, N. Barton, N. Sethi, R. Malik, B. Williams, C. Healey, R. Buest, T. Wu, K. Madaan, S. Sahoo, H. Singh, and P. Sullivan. Market share: It services, worldwide, 2020. Technical report, 2021.

[22] A. Blohowiak, A. Basiri, L. Hochstein, and C. Rosenthal. A platform for automating chaos experiments. In *2016 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2016, Ottawa, ON, Canada, October 23-27, 2016*, pages 5–8. IEEE Computer Society, 2016.

[23] L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

[24] H. D. Chadwick and L. Kurz. Rank permutation group codes based on kendall's correlation statistic. *IEEE Trans. Inf. Theory*, 15(2):306–315, 1969.

[25] B. Chen and Z. M. J. Jiang. Characterizing and detecting anti-patterns in the logging code. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pages 71–81, 2017.

[26] B. Chen and Z. M. J. Jiang. Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation. *Empirical Software Engineering*, 22(1):330–374, 2017.

[27] B. Chen and Z. M. J. Jiang. Extracting and studying the logging-code-issue-introducing changes in java-based large-scale open source software systems. *Empirical Software Engineering*, 24(4):2285–2322, 2019.

[28] B. Chen and Z. M. J. Jiang. Studying the use of java logging utilities in the wild. In *Proc. of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, page 397–408, 2020.

[29] J. Chen, X. He, Q. Lin, Y. Xu, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang. An empirical investigation of incident triage for online service systems. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 111–120. IEEE / ACM, 2019.

[30] P. Chen, Y. Qi, P. Zheng, and D. Hou. Causeinfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014*, pages 1887–1895. IEEE, 2014.

[31] Z. Chen, Y. Kang, L. Li, X. Zhang, H. Zhang, H. Xu, Y. Zhou, L. Yang, J. Sun, Z. Xu, Y. Dang, F. Gao, P. Zhao, B. Qiao, Q. Lin, D. Zhang, and M. R. Lyu. Towards intelligent incident management: why we need it and how we make it. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1487–1497. ACM, 2020.

[32] Z. Chen, J. Liu, Y. Su, H. Zhang, X. Wen, X. Ling, Y. Yang, and M. R. Lyu. Graph-based incident aggregation for large-scale online service systems. In *ASE '21:*

*36th IEEE/ACM International Conference on Automated Software Engineering, Virtual Event, Australia, November 15-19, 2021*, pages 1–12. IEEE/ACM, 2021.

[33] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. 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)*, pages 217–231. USENIX Association, Oct. 2014.

[34] S. Chowdhury, S. Di Nardo, A. Hindle, and Z. M. J. Jiang. An exploratory study on assessing the energy impact of logging on android applications. *Empirical Software Engineering*, 23(3):1422–1456, 2018.

[35] M. Cinque, D. Cotroneo, R. Natella, and A. Pecchia. Assessing and improving the effectiveness of logs for the analysis of software faults. In *Proc. of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 457–466, 2010.

[36] M. Cinque, D. Cotroneo, and A. Pecchia. Event logs for the analysis of software failures: A rule-based approach. *IEEE Transactions on Software Engineering*, 39(6):806–821, 2012.

[37] C. Cortes and V. Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, 1995.

[38] R. DeFauw, A. Chigani, and N. Harris. It resilience within aws cloud, part ii: Architecture and patterns, 2021.

[39] C. L. Dickson. A working theory-of-monitoring. Technical report, Google, Inc., 2013.

[40] R. Dillon and Y. Y. Haimes. Risk of extreme events via multiobjective decision trees: application to telecommunications. *IEEE Trans. Syst. Man Cybern. Part A*, 26(2):262–271, 1996.

[41] R. Ding, H. Zhou, J. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie. Log2: A cost-aware logging mechanism for performance diagnosis. In *Proc. of the 2015 USENIX Annual Technical Conference (ATC)*, pages 139–150, 2015.

[42] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[43] K. EIDefrawy, T. Kim, and P. Sylla. Automated inference of dependencies of network services and applications via transfer entropy. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 32–37. IEEE, 2016.

[44] P. Esling and C. Agón. Time-series data mining. *ACM Comput. Surv.*, 45(1):12:1–12:34, 2012.

[45] A. Fakhrazari and H. Vakilzadian. A survey on time series data mining. In *IEEE International Conference on Electro Information Technology, EIT 2017, Lincoln, NE, USA, May 14-17, 2017*, pages 476–481. IEEE, 2017.

[46] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings.* USENIX, 2007.

[47] Q. Fu, J. Zhu, W. Hu, J. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? an empirical study on logging practices in industry. In *Proc. of the 36th International Conference on Software Engineering (ICSE)*, pages 24–33, 2014.

[48] J. Gabrielson. Challenges with distributed systems, 2022.

[49] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou. Sage: practical and scalable ml-driven performance debugging in microservices. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 135–151. ACM, 2021.

[50] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 3–18. ACM, 2019.

[51] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. 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, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 19–33. ACM, 2019.

[52] C. Gao, J. Zeng, M. R. Lyu, and I. King. Online app review analysis for identifying emerging issues. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 48–58. ACM, 2018.

[53] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Mach. Learn.*, 63(1):3–42, 2006.

[54] Google. cadvisor: Analyzes resource usage and performance characteristics of running containers., 2022.

[55] GrubHub. Enabling continuous (food) delivery at grubhub, 2015.

[56] J. Gu, J. Wen, Z. Wang, P. Zhao, C. Luo, Y. Kang, Y. Zhou, L. Yang, J. Sun, Z. Xu, B. Qiao, L. Li, Q. Lin, and D. Zhang. Efficient customer incident triage via linking with system incidents. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1296–1307. ACM, 2020.

[57] Q. Gu, Z. Li, and J. Han. Generalized fisher score for feature selection. In *UAI 2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011*, pages 266–273. AUAI Press, 2011.

[58] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why does the cloud stop computing? lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*, pages 1–16. ACM, 2016.

[59] X. Guo, X. Peng, H. Wang, W. Li, H. Jiang, D. Ding, T. Xie, and L. Su. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1387–1397. ACM, 2020.

[60] J. Han, J. Pei, and M. Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[61] M. Hassani, W. Shang, E. Shihab, and N. Tsantalis. Studying and detecting log-related issues. *Empirical Software Engineering*, pages 3248–3280, 2018.

[62] P. He, Z. Chen, S. He, and M. R. Lyu. Characterizing the natural language descriptions in software logging statements. In *Proc. of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 178–189, 2018.

[63] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu. A survey on automated log analysis for reliability engineering. *ACM Comput. Surv.*, 54(6), July 2021.

[64] S. He, Q. Lin, J. Lou, H. Zhang, M. R. Lyu, and D. Zhang. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 60–70. ACM, 2018.

[65] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar. Gremlin: Systematic resilience testing of microservices. In *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*, pages 57–66. IEEE Computer Society, 2016.

[66] T. K. Ho. The random subspace method for constructing decision forests. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(8):832–844, 1998.

[67] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 1–16. USENIX Association, 2018.

[68] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The achilles' heel of

cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, pages 150–155. ACM, 2017.

[69] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, pages 150–155. ACM, 2017.

[70] L. J. Jagadeesan and V. B. Mendiratta. When failure is (not) an option: Reliability models for microservices architectures. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Coimbra, Portugal, October 12-15, 2020*, pages 19–24. IEEE, 2020.

[71] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *Proc. of the 24th IEEE International Conference on Software Maintenance (ICSM)*, pages 307–316, 2008.

[72] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[73] S. Kabinna, C. Bezemer, W. Shang, and A. E. Hassan. Logging library migrations: a case study for the apache

software foundation projects. In *Proc. of the 13th International Conference on Mining Software Repositories (MSR)*, pages 154–164, 2016.

[74] S. Kabinna, C. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan. Examining the stability of logging statements. *Empirical Software Engineering*, pages 290–333, 2018.

[75] E. J. Keogh. Exact indexing of dynamic time warping. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*, pages 406–417. Morgan Kaufmann, 2002.

[76] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*. Springer, 1997.

[77] Y. Kim and J. Kim. Gradient LASSO for feature selection. In *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004*, volume 69 of *ACM International Conference Proceeding Series*. ACM, 2004.

[78] Kubernetes. Kubernetes documentation: Cluster architecture, 2022.

[79] Kubernetes. Kubernetes documentation: Disruptions, 2022.

[80] V. T. Lam, M. Mitzenmacher, and G. Varghese. Carousel: Scalable logging for intrusion prevention systems. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-*

*30, 2010, San Jose, CA, USA*, pages 361–376. USENIX Association, 2010.

[81] V. Le and H. Zhang. Log-based anomaly detection without log parsing. In *ASE '21: 36th IEEE/ACM International Conference on Automated Software Engineering, Virtual Event, Australia, November 15-19, 2021*, pages 1–12. IEEE/ACM, 2021.

[82] S. Levy, R. Yao, Y. Wu, Y. Dang, P. Huang, Z. Mu, P. Zhao, T. Ramani, N. K. Govindaraju, X. Li, Q. Lin, G. L. Shafriri, and M. Chintalapati. Predictive and adaptive failure mitigation to avert production cloud VM interruptions. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 1155–1170. USENIX Association, 2020.

[83] S. Lewis. Software resilience testing, 2021.

[84] H. Li, T. P. Chen, W. Shang, and A. E. Hassan. Studying software logging using topic models. *Empir. Softw. Eng.*, pages 2655–2694, 2018.

[85] H. Li, W. Shang, and A. E. Hassan. Which log level should developers choose for a new logging statement? *Empirical Software Engineering*, pages 1684–1716, 2017.

[86] H. Li, W. Shang, Y. Zou, and A. E. Hassan. Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, pages 1831–1865, 2017.

[87] S. Li, X. Niu, Z. Jia, X. Liao, J. Wang, and T. Li. Guiding log revisions by learning from software evolution history. *Empir. Softw. Eng.*, pages 2302–2340, 2020.

[88] Z. Li, T.-H. Chen, and W. Shang. Where shall we log? studying and suggesting logging locations in code blocks. In *Proc. of the 35rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[89] Z. Li, T. P. Chen, J. Yang, and W. Shang. Dlfinder: characterizing and detecting duplicate logging code smells. In *Proc. of the 41st International Conference on Software Engineering (ICSE)*, pages 152–163, 2019.

[90] Z. Li, Q. Cheng, K. Hsieh, Y. Dang, P. Huang, P. Singh, X. Yang, Q. Lin, Y. Wu, S. Levy, and M. Chintalapati. Gandalf: An intelligent, end-to-end analytics service for safe deployment in large-scale cloud infrastructure. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 389–402. USENIX Association, 2020.

[91] H. Liu, S. Lu, M. Musuvathi, and S. Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*, pages 155–162. ACM, 2019.

[92] P. Liu, Y. Chen, X. Nie, J. Zhu, S. Zhang, K. Sui, M. Zhang, and D. Pei. Fluxrank: A widely-deployable framework to automatically localizing root cause machines

for software service failure mitigation. In *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019*, pages 35–46. IEEE, 2019.

[93] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li. Which variables should i log? *IEEE Transactions on Software Engineering*, pages 308–317, 2019.

[94] J. Lockerman, J. M. Faleiro, J. Kim, S. Sankaran, D. J. Abadi, J. Aspnes, S. Sen, and M. Balakrishnan. The fuzzylog: a partially ordered shared log. In *Proc. of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 357–372, 2018.

[95] Log4j. Apache log4j, 2020.

[96] Z. Long, G. Wu, X. Chen, C. Cui, W. Chen, and J. Wei. Fitness-guided resilience testing of microservice-based applications. In *2020 IEEE International Conference on Web Services, ICWS 2020, Beijing, China, October 19-23, 2020*, pages 151–158. IEEE, 2020.

[97] J.-G. Lou, Q. Fu, Y. Wang, and J. Li. Mining dependency in distributed systems through unstructured logs analysis. *ACM SIGOPS Operating Systems Review*, 44(1):91–96, 2010.

[98] L. Luo, S. Nath, L. R. Sivalingam, M. Musuvathi, and L. Ceze. Troubleshooting transiently-recurring errors in production systems with blame-proportional logging. In *Proc. of the 2018 USENIX Annual Technical Conference (ATC)*, pages 321–334, 2018.

[99] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh. Using service dependency graph to analyze and test microservices. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 02, pages 81–86, 2018.

[100] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 378–393, New York, NY, USA, 2015. Association for Computing Machinery.

[101] S. Maldonado and J. López. Dealing with high-dimensional class-imbalanced datasets: Embedded feature selection for SVM classification. *Appl. Soft Comput.*, 67:94–105, 2018.

[102] R. Melo and D. Macedo. A cloud immune security model based on alert correlation and software defined network. In *28th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2019, Naples, Italy, June 12-14, 2019*, pages 52–57. IEEE, 2019.

[103] N. C. Mendonça, C. M. Aderaldo, J. Cámara, and D. Garlan. Model-based analysis of microservice resiliency patterns. In *2020 IEEE International Conference on Software Architecture, ICSA 2020, Salvador, Brazil, March 16-20, 2020*, pages 114–124. IEEE, 2020.

[104] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, et al. Loganomaly: unsupervised detection of sequential and quantitative anoma-

lies in unstructured logs. In *Proc. of the 2019 International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 4739–4745, 2019.

[105] 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 *28th IEEE/ACM International Symposium on Quality of Service, IWQoS 2020, Hangzhou, China, June 15-17, 2020*, pages 1–10. IEEE, 2020.

[106] I. G. Methodology. Test software resiliency, 2022.

[107] Microsoft. Event logging, 2018.

[108] Microsoft. Microservices architecture style, 2019.

[109] T. Mizouchi, K. Shimari, T. Ishio, and K. Inoue. PADLA: a dynamic log level adapter using online phase detection. In *Proc. of the 27th International Conference on Program Comprehension (ICPC)*, pages 135–138, 2019.

[110] S. Morishita and J. Sese. Transversing itemset lattices with statistical metric pruning. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 226–236, 2000.

[111] M. Mormul, P. Hirmer, C. Stach, and B. Mitschang. DEAR: distributed evaluation of alerting rules. In *13th IEEE International Conference on Cloud Computing, CLOUD 2020, Virtual Event, 18-24 October 2020*, pages 158–165. IEEE, 2020.

[112] W. Mostert, K. M. Malan, and A. P. Engelbrecht. Filter versus wrapper feature selection based on problem landscape features. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2018, Kyoto, Japan, July 15-19, 2018*, pages 1489–1496. ACM, 2018.

[113] A. Mueen, H. Hamooni, and T. Estrada. Time series join on subsequence correlation. In *2014 IEEE International Conference on Data Mining, ICDM 2014, Shenzhen, China, December 14-17, 2014*, pages 450–459. IEEE Computer Society, 2014.

[114] K. Neshatian and M. Zhang. Pareto front feature selection: using genetic programming to explore feature space. In *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, pages 1027–1034. ACM, 2009.

[115] S. Newman. *Building microservices - designing fine-grained systems, 1st Edition*. O'Reilly, 2015.

[116] V. Niennattrakul and C. A. Ratanamahatana. On clustering multimedia time series data using k-means and dynamic time warping. In *2007 International Conference on Multimedia and Ubiquitous Engineering (MUE 2007), 26-28 April 2007, Seoul, Korea*, pages 733–738. IEEE Computer Society, 2007.

[117] P. Novotny, B. J. Ko, and A. L. Wolf. On-demand discovery of software service dependencies in manets. *IEEE Transactions on Network and Service Management*, 12(2):278–292, 2015.

[118] A. J. Oliner, A. Ganapathi, and W. Xu. Advances and challenges in log analysis. *ACM Communication*, pages 55–61, 2012.

[119] OpenTracing. The opentracing semantic specification, 2021.

[120] OpenTracing. Opentracing spring cloud, 2021.

[121] D. L. Oppenheimer and D. A. Patterson. Architecture and dependability of large-scale internet services. *IEEE Internet Comput.*, 6(5):41–49, 2002.

[122] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. W. Fletcher, A. Miller, and D. Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[123] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo. Industry practices and event logging: assessment of a critical software development process. In *Proc. of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 169–178, 2015.

[124] C. A. Ratanamahatana, J. Lin, D. Gunopulos, E. J. Keogh, M. Vlachos, and G. Das. Mining time series data. In *Data Mining and Knowledge Discovery Handbook, 2nd ed*, pages 1049–1077. Springer, 2010.

[125] M. S. Raza and U. Qamar. A hybrid feature selection approach based on heuristic and exhaustive algorithms using

rough set theory. In *Proceedings of the International Conference on Internet of Things and Cloud Computing, Cambridge, UK, March 22-23, 2016*, pages 47:1–47:7. ACM, 2016.

[126] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE transactions on acoustics, speech, and signal processing*, 26(1):43–49, 1978.

[127] N. Samir and B. Kyle. Production software application performance and resiliency testing, 2020.

[128] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proc. of the 2006 International Conference on Dependable Systems and Networks (DSN)*, pages 249–258, 2006.

[129] A. W. Services. Aws cloud products, 2021.

[130] A. W. Services. Aws post-event summaries, 2022.

[131] S. Y. Shah, Z. Yuan, S. Lu, and P. Zerfos. Dependency analysis of cloud applications for performance monitoring using recurrent neural networks. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 1534–1543. IEEE, 2017.

[132] W. Shang. Bridging the divide between software developers and operators using logs. In *Proc. of the 34th International Conference on Software Engineering (ICSE)*, pages 1583–1586, 2012.

[133] W. Shang, M. Nagappan, and A. E. Hassan. Studying the relationship between logging characteristics and the code

quality of platform software. *Empirical Software Engineering*, pages 1–27, 2015.

[134] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang. Understanding log lines using development knowledge. In *Proc. of the 30th International Conference on Software Maintenance and Evolution (ICSME)*, 2014.

[135] M. Shetty, C. Bansal, S. Kumar, N. Rao, N. Nagappan, and T. Zimmermann. Neural knowledge extraction from cloud service incidents. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*, pages 218–227. IEEE, 2021.

[136] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[137] SLF4J. Simple logging facade for java (slf4j), 2020.

[138] spdlog. Spdlog, 2020.

[139] M. E. Tipping and C. M. Bishop. Mixtures of probabilistic principal component analysers. *Neural Comput.*, 11(2):443–482, 1999.

[140] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590. IEEE, 2015.

[141] H. Wang, C. Shah, P. Sathaye, A. Nahata, and S. Katariya. Service application knowledge graph and dependency system. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 134–136. IEEE, 2019.

[142] H. Wang, Z. Wu, H. Jiang, Y. Huang, J. Wang, S. Köprü, and T. Xie. Groot: An event-graph-based approach for root cause analysis in industrial settings. In *ASE '21: 36th IEEE/ACM International Conference on Automated Software Engineering, Virtual Event, Australia, November 15-19, 2021*, pages 1–12. IEEE/ACM, 2021.

[143] P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, and P. Chen. Cloudranger: Root cause identification for cloud native systems. In *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*, pages 492–502. IEEE Computer Society, 2018.

[144] Y. Wang, G. Li, Z. Wang, Y. Kang, Y. Zhou, H. Zhang, F. Gao, J. Sun, L. Yang, P. Lee, Z. Xu, P. Zhao, B. Qiao, L. Li, X. Zhang, and Q. Lin. Fast outage analysis of large-scale production clouds with service correlation mining. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 885–896. IEEE, 2021.

[145] Y. Wang, G. Li, Z. Wang, Y. Kang, Y. Zhou, H. Zhang, F. Gao, J. Sun, L. Yang, P. Lee, Z. Xu, P. Zhao, B. Qiao, L. Li, X. Zhang, and Q. Lin. Fast outage analysis of large-scale production clouds with service correlation mining.

In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 885–896. IEEE, 2021.

[146] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[147] S. Yang, S. J. Park, and J. K. Ousterhout. Nanolog: A nanosecond scale logging system. In *Proc. of the 2018 USENIX Annual Technical Conference (ATC)*, pages 335–350, 2018.

[148] T. Yang, C. Gao, J. Zang, D. Lo, and M. R. Lyu. TOUR: dynamic topic and sentiment analysis of user reviews for assisting app release. In *Companion of The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, pages 708–712. ACM / IW3C2, 2021.

[149] T. Yang, J. Shen, Y. Su, X. Ling, Y. Yang, and M. R. Lyu. AID: efficient prediction of aggregated intensity of dependency in large-scale cloud systems. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 653–665. IEEE, 2021.

[150] K. Yao, G. B. de Pádua, W. Shang, C. Sporea, A. Toma, and S. Sajedi. Log4perf: suggesting and updating logging locations for web-based systems' performance monitoring. *Empir. Softw. Eng.*, 25(1):488–531, 2020.

[151] J. Yin, X. Zhao, Y. Tang, C. Zhi, Z. Chen, and Z. Wu. Cloudscout: A non-intrusive approach to service depen-

dency discovery. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1271–1284, 2016.

[152] K. Yin and Q. Du. On representing resilience requirements of microservice architecture systems. *Int. J. Softw. Eng. Knowl. Eng.*, 31(6):863–888, 2021.

[153] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be conservative: enhancing failure diagnosis with proactive logging. In *Proc. of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 293–306, 2012.

[154] D. Yuan, S. Park, and Y. Zhou. Characterizing logging practices in open-source software. In *Proc. of 34th International Conference on Software Engineering (ICSE)*, pages 102–112, 2012.

[155] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 3–14, 2011.

[156] A. Zand, A. Houmansadr, G. Vigna, R. Kemmerer, and C. Kruegel. Know your achilles' heel: Automatic detection of network critical services. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 41–50, 2015.

[157] A. Zand, G. Vigna, R. Kemmerer, and C. Kruegel. Rippler: Delay injection for service dependency detection. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 2157–2165. IEEE, 2014.

[158] Y. Zeng, J. Chen, W. Shang, and T. P. Chen. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. *Empirical Software Engineering*, pages 3394–3434, 2019.

[159] E. Zhai, A. Chen, R. Piskac, M. Balakrishnan, B. Tian, B. Song, and H. Zhang. Check before you change: Preventing correlated failures in service updates. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 575–589. USENIX Association, 2020.

[160] X. Zhang, Q. Lin, Y. Xu, S. Qin, H. Zhang, B. Qiao, Y. Dang, X. Yang, Q. Cheng, M. Chintalapati, Y. Wu, K. Hsieh, K. Sui, X. Meng, Y. Xu, W. Zhang, F. Shen, and D. Zhang. Cross-dataset time series anomaly detection for cloud systems. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 1063–1076. USENIX Association, 2019.

[161] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, et al. Robust log-based anomaly detection on unstable log data. In *Proc. of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[162] X. Zhang, Y. Xu, S. Qin, S. He, B. Qiao, Z. Li, H. Zhang, X. Li, Y. Dang, Q. Lin, M. Chintalapati, S. Rajmohan, and D. Zhang. Onion: identifying incident-indicating logs for cloud systems. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on*

*the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 1253–1263. ACM, 2021.

[163] Y. Zhang, S. Makarov, X. Ren, D. Lion, and D. Yuan. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 19–33, 2017.

[164] G. Zhao, S. Hassan, Y. Zou, D. Truong, and T. Corbin. Predicting performance anomalies in software systems at run-time. *ACM Trans. Softw. Eng. Methodol.*, 30(3):33:1–33:33, 2021.

[165] N. Zhao, J. Chen, X. Peng, H. Wang, X. Wu, Y. Zhang, Z. Chen, X. Zheng, X. Nie, G. Wang, Y. Wu, F. Zhou, W. Zhang, K. Sui, and D. Pei. Understanding and handling alert storm for online service systems. In *ICSE-SEIP 2020: 42nd International Conference on Software Engineering, Software Engineering in Practice, Seoul, South Korea, 27 June - 19 July, 2020*, pages 162–171. ACM, 2020.

[166] N. Zhao, J. Chen, Z. Wang, X. Peng, G. Wang, Y. Wu, F. Zhou, Z. Feng, X. Nie, W. Zhang, K. Sui, and D. Pei. Real-time incident prediction for online service systems. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 315–326. ACM, 2020.

[167] N. Zhao, H. Wang, Z. Li, X. Peng, G. Wang, Z. Pan, Y. Wu, Z. Feng, X. Wen, W. Zhang, K. Sui, and D. Pei.

An empirical investigation of practical log anomaly detection for online service systems. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 1404–1415. ACM, 2021.

[168] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou. The game of twenty questions: do you know where to log? In *Proc. of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[169] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou. Log20: fully automated optimal placement of log printing statements under specified overhead threshold. In *Proc. of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 565–581, 2017.

[170] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 603–618. USENIX Association, Nov. 2016.

[171] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 629–644. USENIX Association, Oct. 2014.

[172] W. Zhou, W. Xue, R. Baral, Q. Wang, C. Zeng, T. Li, J. Xu, Z. Liu, L. Shwartz, and G. Y. Grabarnik. STAR:

A system for ticket analysis and resolution. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, pages 2181–2190. ACM, 2017.

[173] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Trans. Software Eng.*, 47(2):243–260, 2021.

[174] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 683–694. ACM, 2019.

[175] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang. Learning to log: helping developers make informed logging decisions. In *Proc. of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 415–425, 2015.